CIS 110 — Introduction To Computer Programming

December 19th, 2011 — Final

Answer key for review problems

CIS 110 final instructions

- You have 120 minutes to finish this exam. Time will begin when called by a proctor and end precisely 120 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:

  System.out.println ⟶ S.O.PLN

  System.out.print ⟶ S.O.P

  System.out.printf ⟶ S.O.PF

  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 of 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 final 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.
```

The critter class

```
// The Critter class is the base class for all critters in the simulation.
public abstract class Critter {
    // Defines the types of food a critter can eat: grass or meat.
    public static enum FoodType { GRASS, MEAT }

    // Defines the possible directions a critter can move: the four cardinal
    // directions along with "no move" (CENTER).
    public static enum Direction { NORTH, EAST, SOUTH, WEST, CENTER }

    // The speed at which a critter moves: fast, medium, or slow.
    public static enum Speed { FAST, MEDIUM, SLOW }

    // Returns the direction this critter should move for the next step of the
    // simulation.
    public abstract Direction getMove();

    // Returns the type of food this critter eats.
    public abstract FoodType getFoodType();

    // Returns the color of this critter.
    public abstract Color getColor();

    // Returns the speed of this critter.
    public abstract Speed getSpeed();

    // Returns the String representation of this critter.  This should be a
    // a single character which will be rendered onto the simulation board.
    public String toString() {
        throw new UnsupportedOperationException(
                "Implementors must provide their own toString() methods!");
    }
}
```

Notes about this exam:

- The final is cumulative covers all the content covered in the course (chapters 1–9). It is similar to exam 2 in terms of content but with the addition of inhertance and polymorphism (i.e., chapters 8–9).
- The purpose of this exam is to *test your algorithmic thinking skills* rather than have you regurgitate facts about computer programming. To prepare for this exam, you should practice those skills by reviewing homeworks and doing practice problems discussed in section or found in the text.
- The exam is *closed book*, *closed notes*, and *closed electronic devices.* You may bring a single 8.5×11" sheet of double-sided notes to help jog your memory.
- In addition to your note sheet, we will include a sheet of commonly-used APIs so that you do not have to write them down.
- This practice exam introduces you to the types of problems that will be on the real exam. That way you can spend more time solving problems instead of trying to understand what the problem asks of you. That being said, while my goal is not to stray far from this format, the format of the actual exam may change slightly if necessary.
- I strongly recommend that you attempt this practice exam without looking at the answers. Afterwards, check your work, and then use the usual channels (e.g., Piazza, your TA, or myself) to resolve any lingering questions that you may have.

Express yourself

1. (5 points) Evalute these Java expressions to their final values.

   (a) `10 + 10 / 10 - 10 * 10` $\longrightarrow$

   $\boxed{\texttt{-89}}$

   (b) `5 + 41091 % 10 / 2.0` $\longrightarrow$

   $\boxed{\texttt{5.5}}$

   (c) `5 > 10 || (13 != 5 && 6 <= 6)` $\longrightarrow$

   $\boxed{\texttt{true}}$

   (d) `!(true && !(5 >= 3) && 13 % 2 == 1)` $\longrightarrow$

   $\boxed{\texttt{true}}$

   (e)
   ```
   String s = "Hello world!";
   System.out.println("" + s.charAt(1) + s.charAt(11) + s.charAt(4));
   ```
   $\longrightarrow$

   $\boxed{\texttt{e!o}}$

Method mysteries: the next generation

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

```
public static int[] mystery1(int[] arr1, int[] arr2) {
    if (arr1.length > arr2.length) {
        int[] t = arr1;
        arr1 = arr2;
        arr2 = t;
    }
    int[] arr3 = new int[arr1.length + arr2.length];
    for (int i = 0; i < arr1.length; i++) {
        if (arr1[i] < arr2[i]) {
            arr3[i*2] = arr1[i];
            arr3[i*2+1] = arr2[i];
        } else {
            arr3[i*2] = arr2[i];
            arr3[i*2+1] = arr1[i];
        }
    }
    for (int i = arr1.length; i < arr2.length; i++) {
        int x = arr1.length * 2 + i - arr1.length;
        arr3[arr1.length * 2 + i - arr1.length] = arr2[i];
    }
    return arr3;
}
```

(a) `mystery1(new int[]{1, 3, 5, 7}, new int[]{0, 2, 4, 6})` $\longrightarrow$

`[0, 1, 2, 3, 4, 5, 6, 7]`

(b) `mystery1(new int[]{0, 2, 4, 6}, new int[]{1, 3, 5, 7})` $\longrightarrow$

`[0, 1, 2, 3, 4, 5, 6, 7]`

(c) `mystery1(new int[]{3, 9, 6}, new int[]{10, 7, 9})` $\longrightarrow$

`[3, 10, 7, 9, 6, 9]`

(d) `mystery1(new int[]{5, 3, 1}, new int[]{12, 1, 2, 5, 9})` $\longrightarrow$

`[5, 12, 1, 3, 1, 2, 5, 9]`

(e) `mystery1(new int[]{7, 6, 3, 9, 4}, new int[]{3, 8, 4})` $\longrightarrow$

`[3, 7, 6, 8, 3, 4, 9, 4]`

Be assertive

3. (15 points) For 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 or sometimes false. You may abbreviate always, never, and sometimes as A, T, and S respectively.

```
public static int mystery(int n) {
    Random r = new Random();
    int a = r.nextInt(2)+1;
    int b = 2;
    // POINT A
    while (n > b) {
        // POINT B
        b = b + a;
        if (a > 1) {
            n--;
            // POINT C
            a = r.nextInt(2)+1;
        } else {
            a = b + 2;
            // POINT D
        }
    }
    // POINT E
    return n;
```

	`n > b`	`a > 1`	`a > b`
A	Sometimes	Sometimes	Never
B	Always	Sometimes	Sometimes
C	Sometimes	Always	Never
D	Sometimes	Always	Always
E	Never	Sometimes	Sometimes

Polymorph

4. (15 points) For the class hierarchy below, fill out the table with the results of method calls to `noise()` on objects in that class hierarchy. For example, the top-left corner of the table is the result of the call:

```
Instrument i = new Instrument();
i.noise();
```

Several of the combinations of variables and assignments below cause compiler errors. For those that cause compiler errors, write "Error" in place of the output.

```
class Instrument {
    public String noise() { return "Honk"; }
}
class Flute extends Instrument {
    public String noise() { return "Whoosh"; }
}
class Piccalo extends Flute {
    public String noise() { return "Tweet"; }
}
class Oboe extends Instrument { }
class EnglishHorn extends Oboe {
    public String noise() { return "Hooooonk"; }
}
```

	`= new Piccalo()`	`= new Oboe()`	`= new EnglishHorn()`
`Instrument i`	`"Tweet"`	`"Honk"`	`"Hooooonk"`
`Flute f`	`"Tweet"`	Error	Error
`Piccalo p`	`"Tweet"`	Error	Error
`Oboe o`	Error	`"Honk`	`"Hooooonk"`
`EnglishHorn e`	Error	Error	`"Hooooonk"`

Flying one more time

5. (20 points) Write a method `printPyramid` that takes an integer `n` and prints a pyramid of height `n`. The pyramid itself consists of the numbers of 0 through 9 in increasing order from top-to-bottom, left-to-right. After printing 9, we wrap around back to 0 and continue. Here are some example invocations of `printPyramid` and their output.

`printPyramid(3)`	`0`
	`1 2`
	`3 4 5`

`printPyramid(1)`	`0`

`printPyramid(5)`	`0`
	`1 2`
	`3 4 5`
	`6 7 8 9`
	`0 1 2 3 4`

```
public static void printPyramid(int n) {
    int val = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i+1; j++) {
            System.out.print(val++ % 10 + " ");
        }
        System.out.println();
    }
}
```

Nom nom nom

6. (20 points) Write a method `worstPlayer` that takes a `Scanner` that is reading from a text file in a particular format, and returns a String that contains the full name of the the worst basketball player found in the database. The worst player is defined as the player with the lowest points-per-game (ppg). The file contains a series of player entries for a basketball team where each record represents a player's stats, one record per line, in the following format:

```
<Lastname> <Firstname> <weight> <year> <ppg>
```

In the event that multiple players share the lowest ppg, you may return any of them. The String that `worstPlayer` returns should be in the format `<Firstname> <Lastname>`. You may assume the file is well-formatted and contains at least one record. For example, the following is a possible text file in the correct format:

```
Allen Lavoy 225 R 0.0
Brackins Craig 230 R 2.7
Brand Elton 254 11 15.0
Daniels Antonio 205 12 1.5
Hawes Spencer 245 3 7.2
Holiday Jrue 180 1 14.0
Iguodala Andre 207 6 14.1
```

If `worstplayer` is called with a `Scanner` pointed to this file, then it would return `"Lavoy Allen"` since he has a ppg of 0.0.

Hint: You may find the `double` constant `Double.MAX_VALUE` useful here.

```
public static String worstPlayer(Scanner in) {
    String worstPlayer = null;
    double lowestPPG = Double.MAX_VALUE;
    while (in.hasNextLine()) {
        Scanner line = new Scanner(in.nextLine());
        String last = line.next();
        String first = line.next();
        line.next();
        line.next();
        double ppg = line.nextDouble();
        if (ppg < lowestPPG) {
            lowestPPG = ppg;
            worstPlayer = first + " " + last;
        }
    }
    return worstPlayer;
}
```

I've got 99 problems and an array ain't one

7. (20 points) Write a method `isSurrounded` that takes a square two-dimensional grid of integers and integer `row` and `col` values and returns true if the element at `(col, row)` in the grid is surrounded by 1s. An element is surrounded by 1s if it's neighbors in all the cardinal directions — north, south, east, and west — are 1s. That is, the array locations

```
arr[col-1][row]
arr[col+1][row]
arr[col][row-1]
arr[col][row+1]
```

all have value 1. If one of indices goes over the edge of the grid, then we consider that side of the element as surrounded.

For example, consider the array declaration:

```
int[][] arr = new int[][] { { 0, 0, 1, 0, 1 },
                            { 1, 0, 1, 0, 1 },
                            { 1, 1, 1, 1, 0 },
                            { 0, 1, 0, 1, 0 },
                            { 1, 1, 1, 1, 0 } };
```

where the 0th row refers to the sub-array `[0, 0, 1, 0, 1]`. Here are example invocations of `isSurrounded` on that array.

`isSurrounded(arr, 3, 1)`	false
`isSurrounded(arr, 2, 3)`	true
`isSurrounded(arr, 0, 3)`	true

```
public static boolean isSurrounded(int[][] arr, int row, int col) {
    // Note: the expression boolean b = e1 ? e2 : e3 is equivalent to:
    // boolean b = false
    // if (e1) { b = e2 }
    // else { b = e3 }
    // You don't need to know this for the exam.  It's just shorthand syntax.
    if (col >= arr.length || row >= arr[0].length) { return false; }
    boolean left = col-1 >= 0 ? arr[col-1][row] == 1 : true;
    boolean right = col+1 < arr.length ? arr[col+1][row] == 1 : true;
    boolean up = row-1 >= 0 ? arr[col][row-1] == 1 : true;
    boolean down = row+1 < arr[col].length ? arr[col][row+1] == 1 : true;
    return left && right && up && down;
}
```

Didn't we just do this?

8. (20 points) Write a subclass of `Critter` called `Giraffe` that meets the following specification:

Construct	`Giraffe()`
Food type	Grass
Movement	Repeatedly Moves in a L shape randomly either to the north the south. The L shape itself is composed of two lines, both of length 3. When moving south, the giraffe goes south 3 steps then east 3 steps. When moving north, the giraffe goes north 3 steps then west 3 steps.
Color	Yellow
Speed	Slow
toString	`"G"`

For reference, the essential pieces of the critter class are reproduced as part of your cheat sheet.

```
public class Giraffe extends Critter {
    private boolean isMovingNorth;
    private int counter;
    private Random rand;
    public Giraffe() {
        this.rand = new Random();
        this.isMovingNorth = rand.nextBoolean();
    }
    public Direction getMove() {
        Direction ret = Direction.NORTH;
        if (counter == 6) {
            isMovingNorth = rand.nextBoolean();
            counter = 0;
        }
        if (isMovingNorth) {
            if (counter < 3) {
                ret = Direction.NORTH;
            } else {
                ret = Direction.WEST;
            }
        } else {
            if (counter < 3) {
                ret = Direction.SOUTH;
            } else {
                ret = Direction.EAST;
            }
        }
        counter++;
        return ret;
    }
    public FoodType getFoodType() { return FoodType.GRASS; }
    public Color getColor() { return Color.YELLOW; }
    public Speed getSpeed() { return Speed.SLOW; }
    public String toString() { return "G"; }
}
```

That's a wrap!

9. (25 points)

   (a) Write a method `swapLowestToBottom` that takes an integer array `arr` and an index `begin` that finds the smallest element found in `arr` starting from index `begin` to the end of the array. `swapLowestToBottom` then swaps that smallest element with the element found at index `begin`. Note that if `begin = 0` then we are swapping the smallest element in the array with the first element of the array. Also note that `swapLowestToBottom` does not return a value or output anything to the console.

   For the following arrays, here are example method invocations of `swapLowestToBottom` and the resulting changes to those arrays.

```
int[] arr1 = new int[] { 3, 5, 4, 9, 1 };
int[] arr2 = new int[] { 2, 8, 6, 4 };
int[] arr3 = new int[] { 5, 4, 3, 2, 1, 0 };
```

`swapLowestToBottom(arr1, 0)`	[1, 5, 4, 9, 3]
`swapLowestToBottom(arr2, 1)`	[2, 4, 6, 8]
`swapLowestToBottom(arr3, 3)`	[5, 4, 3, 0, 1, 2]

```
public static void swapLowestToBottom(int[] arr, int begin) {
    int index = begin;
    for (int i = begin; i < arr.length; i++) {
        if (arr[i] < arr[index]) {
            index = i;
        }
    }
    int temp = arr[begin];
    arr[begin] = arr[index];
    arr[index] = temp;
}
```

(b) Using `swapLowestToBottom`, write a method `sort` that takes an integer array and sorts the elements of that array in ascending order. You may only `swapLowestToBottom` in your implementation of `sort`. In particular, you may not use any other sorting methods in your `sort` methods. Note that `sort` does not return a value or output anything to the console.

For the following arrays, here are example method invocations of `sort` and the resulting changes to those arrays.

```
int[] arr4 = new int[] { 3, 5, 4, 9, 1 };
int[] arr5 = new int[] { 0, 1, 2 };
int[] arr6 = new int[] { 5, 4, 3, 2, 1, 0 };
```

`sort(arr4)`	[1, 3, 4, 5, 9]
`sort(arr5)`	[0, 1, 2]
`sort(arr6)`	[0, 1, 2, 3, 4, 5]

Hint: The code for this method — now that you've written `swapLowestToBottom` — turns out to not be that complicated. Consider how you can decompose the problem of sorting an array by repeatedly calling this helper method.

```
public static void sort(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        swapLowestToBottom(arr, i);
    }
}
```

Scratch paper