CSE 142, Autumn 2008
Final Exam
Wednesday, December 10, 2008

Name: _____

Section: _____ **TA:** _____

Student ID #: _____

Rules:

- You have 110 minutes to complete this exam.
  You may receive a deduction if you keep working after the instructor calls for papers.
- This test is open-book/notes.
- You may not use any computing devices of any kind including calculators.
- Unless otherwise indicated, your code will be graded on proper behavior/output, not on style.
- You do not need to write any `import` statements in your code.
- Please do not abbreviate code, such as writing ditto marks ("") or dot-dot-dot marks (...). The only abbreviations allowed are `S.o.p`, `S.o.pln`, and `S.o.pf` for `System.out.print`, `println`, and `printf`.
- If you enter the room, you must turn in an exam and will not be permitted to leave without doing so.
- You must show your Student ID to a TA or instructor for your submitted exam to be accepted.

Good luck!

Problem	Description	Earned	Max
1	Expressions		10
2	Array Mystery		15
3	Inheritance Mystery		15
4	File Processing		15
5	Array Programming		15
6	Critters		10
7	Classes and Objects		10
8	Array Programming		10
X	Extra Credit		+1
TOTAL	**Total Points**		**100**

1. Expressions

For each expression in the left-hand column, indicate its value in the right-hand column.
Be sure to list a constant of appropriate type and capitalization, and do not abbreviate.
e.g., write `7` for an `int`, `7.0` for a `double`, `"hello"` for a `String`, or `true` or `false` for a `boolean`.

Expression	Value
`-(1 + 2 * 3 + (1 + 2) * 3)`	_____
`1 + 2 + "(3 + 4)" + 5 * 6 + 7`	_____
`30 % 9 + 5 % 8 - 11 % 4 % 2`	_____
`4 < 10 != (5 == 6 || 9 >= 9)`	_____
`(double) 45 / 10.0 / 2 + 7 / 4.0 * 10`	_____

2. Array Mystery

Consider the following method:

```
public static void arrayMystery(int[] a) {
    for (int i = a.length - 2; i > 0; i--) {
        if (a[i - 1] < a[i + 1]) {
            a[i] += a[i - 1];
        } else {
            a[i] += a[i + 1];
        }
    }
}
```

Indicate in the right-hand column what values would be stored in the array after the method `arrayMystery` executes if the array in the left-hand column is passed as its parameter.

Original Contents of Array	Final Contents of Array
`int[] a1 = {1, 2, 3};` `arrayMystery(a1);`	_____
`int[] a2 = {8, 2, 3, 1, 6};` `arrayMystery(a2);`	_____
`int[] a3 = {1, 1, 1, 1, 1, 1};` `arrayMystery(a3);`	_____
`int[] a4 = {40, 10, 25, 5, 10, 30};` `arrayMystery(a4);`	_____
`int[] a5 = {15, 6, -1, 4, 8, -2, 7, 4};` `arrayMystery(a5);`	_____

3. Inheritance Mystery

Assume that the following four classes have been defined:

```
public class McCain extends Biden {
    public void republican() {
        System.out.print("mccain-R   ");
    }
}

public class Palin {
    public void republican() {
        System.out.print("palin-R   ");
    }

    public void democrat() {
        republican();
        System.out.print("palin-D   ");
    }

    public String toString() {
        return "palin";
    }
}
```

```
public class Obama extends Palin {
    public void republican() {
        super.republican();
        System.out.print("obama-R   ");
    }
}

public class Biden extends Palin {
    public String toString() {
        return "biden";
    }

    public void democrat() {
        System.out.print("biden-D   ");
        super.democrat();
    }
}
```

Given the classes above, what output is produced by the following code?

```
Palin[] politicians = {new Biden(), new Palin(), new McCain(), new Obama()};
for (int i = 0; i < politicians.length; i++) {
    System.out.println(politicians[i]);
    politicians[i].republican();
    System.out.println();
    politicians[i].democrat();
    System.out.println();
    System.out.println();
}
```

4. File Processing

Write a static method named `printDuplicates` that accepts as its parameter a `Scanner` for an input file containing a series of lines. Your method should examine each line looking for consecutive occurrences of the same token on the same line, and print each duplicated token along how many times it appears consecutively. Non-repeated tokens are not printed. Repetition across multiple lines (such as if a line ends with a given token and the next line starts with the same token) is not considered in this problem.

For example, if the input file contains the following text (sequences of duplicated tokens are underlined for emphasis):

```
hello how how are you you you you
I I I am Jack's Jack's smirking smirking smirking smirking smirking revenge
   bow  wow wow yippee yippee   yo yippee   yippee yay  yay yay
one fish two fish red fish blue fish
It's the Muppet Show, wakka wakka wakka
```

Your method would produce the following output for the preceding input file:

```
how*2 you*4
I*3 Jack's*2 smirking*5
wow*2 yippee*2 yippee*2 yay*3

wakka*3
```

Your code prints only the repeated tokens; the ones that appear only once in a row are not shown. Your code should place a single space between each reported duplicate token and should respect the line breaks in the original file. This is why a blank line appears in the expected output, corresponding to the fourth line of the file that did not contain any consecutively duplicated tokens. You may assume that each line of the file contains at least 1 token of input.

5. Array Programming

Write a static method named `arraySum` that accepts two arrays of real numbers *a1* and *a2* as parameters and returns a new array *a3* such that each element of *a3* at each index *i* is the sum of the elements at that same index *i* in *a1* and *a2*. For example, if *a1* stores `{4.5, 5.0, 6.6}` and *a2* stores `{1.1, 3.4, 0.5}`, your method should return `{5.6, 8.4, 7.1}`, which is obtained by adding 4.5 + 1.1, 5.0 + 3.4, and 6.6 + 0.5.

If the arrays *a1* and *a2* are not the same length, the result returned by your method should have as many elements as the larger of the two arrays. If a given index *i* is in bounds of *a1* but not *a2* (or vice versa), your result array's element at index *i* should be equal to the value of the element at index *i* in the longer of *a1* or *a2*. For example, if *a1* stores `{1.8, 2.9, 9.4, 5.5}` and *a2* stores `{2.4, 5.0}`, your method should return `{4.2, 7.9, 9.4, 5.5}`.

The table below shows some additional calls to your method and the expected values returned:

Arrays	Call and Value Returned
`double[] a1 = {4.5, 2.8,  3.4, 0.8};` `double[] a2 = {1.4, 8.9, -1.0, 2.3};`	`arraySum(a1, a2)` returns `{5.9, 11.7, 2.4, 3.1}`
`double[] ax = {2.4, 3.8};` `double[] ay = {0.2, 9.2, 4.3, 2.8, 1.4};`	`arraySum(ax, ay)` returns `{2.6, 13.0, 4.3, 2.8, 1.4}`
`double[] aa = {1.0, 2.0, 3.0};` `double[] ab = {4.0, 5.0};`	`arraySum(aa, ab)` returns `{5.0, 7.0, 3.0}`
`double[] ai = {};` `double[] aj = {42.0};`	`arraySum(ai, aj)` returns `{42.0}`

For full credit, you should not modify the elements of *a1* or *a2*. You may not use a `String` to solve this problem.

6. Critters

Write a class `Hyena` that extends the `Critter` class, along with its **movement and eating** behavior. All unspecified aspects of `Hyena` use the default behavior. Write the complete class with any fields, constructors, etc. necessary.

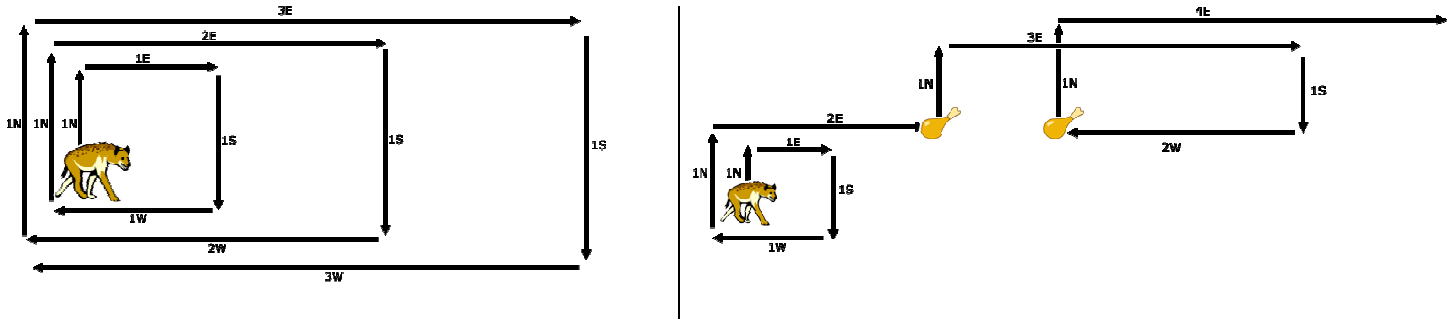
`Hyena` objects move in a rectangular pattern looking for food, walking NORTH, then EAST, then SOUTH, then WEST. Each time the Hyena walks an entire rectangle or eats food, it starts the rectangle pattern over again but with a rectangle 1 step wider than before. The general pattern is as follows, if the Hyena doesn't find any food:

- N, E, S, W, N, E, E, S, W, W, N, E, E, E, S, W, W, W, N, E, E, E, E, S, W, W, W, W, ...

If the `Hyena` encounters food at any point during its movement pattern, it eats the food and starts the pattern over, lengthening the rectangular pattern by 1 in the process. For example:

- N, E, S, W, N, <u>E, E</u> *(eats food)*, N, <u>E, E, E</u>, S, <u>W, W</u> *(eats food)*,
  N, <u>E, E, E, E</u>, S, <u>W, W, W, W</u>, N, <u>E, E, E, E, E</u>, S *(eats food)*, N, <u>E, E, E, E, E, E</u>, S, W, W, ...

The following diagrams summarize this behavior, the left without food and the right when food is encountered:



7. Classes and Objects

Write a method named `absoluteDay` that will be placed in the `Date` class from the Birthday/Date assignment, as shown at right.

The `absoluteDay` method should return the "absolute day of the year" between 1 and 365 represented by the `Date` object. January 1 is absolute day #1, January 2 is day #2, ..., and December 31st is absolute day #365. For example, calling this method on a date representing February 13th should return 44, and calling it on a `Date` representing September 19th should return 262.

The table below summarizes the absolute days for various dates throughout the year.

```
public class Date {
    private int month;
    private int day;

    public Date(int m, int d)

    public int getDay()
    public int getMonth()
    public int daysInMonth()
    public void nextDay()

    // your method would go here
}
```

Date (month/day)	1/1	1/2	1/3	...	1/30	1/31	2/1	2/2	...	2/28	3/1	3/2	...	12/30	12/31
Absolute day	1	2	3	...	30	31	32	33	...	59	60	61	...	364	365

The following table shows the results of calling your method on several `Date` objects.

Object	Call	Returns
`Date jan1 = new Date(1, 1);`	`jan1.absoluteDay()`	1
`Date jan4 = new Date(1, 4);`	`jan4.absoluteDay()`	4
`Date feb1 = new Date(2, 1);`	`feb1.absoluteDay()`	32
`Date mar10 = new Date(3, 10);`	`mar10.absoluteDay()`	69
`Date sep19 = new Date(9, 19);`	`sep19.absoluteDay()`	262
`Date dec31 = new Date(12, 31);`	`dec31.absoluteDay()`	365

You should not solve this problem by writing 12 `if` statements, one for each month; this is redundant and poor style. If you solve the problem that way, you will receive at most half credit. Also, your method should not have the side effect of modifying the `Date` object on which it was called. In other words, when your method is done executing, the values of the fields of the `Date` object on which it was called should be unchanged. A solution that does so will receive at most half credit. (It's okay for you to modify the object's state temporarily if it helps you solve this problem, but when your method is done executing, the state should be the same as when you started.)

Recall that your method is allowed to call other methods on `Date` objects or construct other objects if you like.

8. Array Programming

Write a static method named `partition` that accepts an array of integers *a* and an integer element value *v* as its parameters, and rearranges ("partitions") the array's elements so that all its elements of *a* that are less than *v* occur before all elements that are greater than *v*. The exact order of the elements is unimportant so long as all elements less than v appear before all elements greater than *v*. For example, if your method were passed the following array:

```
int[] a = {15, 1, 6, 12, -3, 4, 8, -7, 21, 2, 30, -1, 9};
partition(a, 5);
```

One acceptable ordering of the elements after the call would be: (elements < 5 and > 5 are underlined for emphasis)

{-1, 1, 2, -7, -3, 4, 8, 12, 21, 6, 30, 15, 9}

Hint: Your method will need to rearrange the elements of the array, which will involve swapping various elements from less desirable indexes to more desirable ones.

You may assume that the array contains no duplicates and does not contain the element value *v* itself. You may not use `Arrays.sort`, `Collections.sort`, or any other pre-defined sorting algorithm from the Java Class Libraries or textbook to solve this problem. You also may not use a `String` to solve this problem.

X. Extra Credit (+1 point)

Draw a picture of what your TA would look like if he/she chose to fully embrace hip-hop culture.
Make sure to write your TA's name (and/or hip-hop nickname) on your picture so we know who it is!

(Any picture that appears to reflect a nontrivial effort will receive the bonus point.)