CSE 142, Spring 2010
Final Exam
Wednesday, June 9, 2010

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

1. Array Mystery

Consider the following method:

```
public static void arrayMystery(int[] a) {
    for (int i = 1; i < a.length - 1; i++) {
        if (a[i] > a[i + 1]) {
            a[i] = a[i + 1] + 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 = {42, 99, 42};` `arrayMystery(a1);`	_____
`int[] a2 = {6, 8, 4, 2};` `arrayMystery(a2);`	_____
`int[] a3 = {7, 7, 20, 8, 1};` `arrayMystery(a3);`	_____
`int[] a4 = {4, 5, 3, 2, 1, 0};` `arrayMystery(a4);`	_____
`int[] a5 = {6, 0, -1, 80, 5, 0, -3};` `arrayMystery(a5);`	_____

2. Reference Semantics Mystery

Write the output of the following program, as it would appear on the console.

```
import java.util.*;   // for Arrays class

public class BasicPoint {
    int x;
    int y;

    public BasicPoint(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }
}

public class ReferenceMystery {
    public static void main(String[] args) {
        BasicPoint p = new BasicPoint(11, 22);
        int[] a = {33, 44};
        int n = 55;

        System.out.println(p.x + "," + p.y + " " + Arrays.toString(a) + " " + n);
        mystery(p, a, n);
        System.out.println(p.x + "," + p.y + " " + Arrays.toString(a) + " " + n);

        a[0] = a[1];
        p.x = p.y;

        mystery(p, a, n);
        System.out.println(p.x + "," + p.y + " " + Arrays.toString(a) + " " + n);
    }

    public static int mystery(BasicPoint p, int[] a, int n) {
        n = 0;
        a[0] = a[0] + 11;
        a[1] = 77;
        p.x = p.x + 33;
        System.out.println(p.x + "," + p.y + " " + Arrays.toString(a) + " " + n);
        return n;
    }
}
```

---

3. Inheritance Mystery

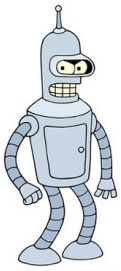
Assume that the following four classes have been defined:

```
public class Leela extends Fry {
    public void method1() {
        System.out.print("Leela1 ");
    }

    public void method2() {
        System.out.print("Leela2 ");
        super.method2();
    }
}

public class Farnsworth extends Bender {
    public void method1() {
        System.out.print("Farnsworth1 ");
    }

    public String toString() {
        return "Good news everyone!";
    }
}
```

```
public class Fry extends Bender {
    public void method2() {
        System.out.print("Fry2 ");
        super.method2();
    }
}

public class Bender {
    public void method1() {
        System.out.print("Bender1 ");
    }

    public void method2() {
        System.out.print("Bender2 ");
        method1();
    }

    public String toString() {
        return "We're doomed!";
    }
}
```

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

```
Bender[] rodriguez = {new Leela(), new Bender(), new Farnsworth(), new Fry()};
for (int i = 0; i < rodriguez.length; i++) {
    rodriguez[i].method2();
    System.out.println();
    System.out.println(rodriguez[i]);
    rodriguez[i].method1();
    System.out.println();
    System.out.println();
}
```



4. File Processing

Write a static method named **`mostCommonNames`** that accepts as its parameter a `Scanner` for an input file whose data is a sequence of lines, where each line contains a set of first names separated by spaces. Your method should print the name that occurs the most frequently in each line of the file. The method should also return the total number of names that were seen in the file. You may assume that **no name appears on more than one line** of the file.

Each line should be considered separately from the others. On a given line, some names are repeated; all occurrences of a given name will appear consecutively in the file. If two or more names occur the same number of times, return the one that appears earlier in the file. If every single name on a given line is different, every name will have 1 occurrence, so you should just return the first name in the file.

For example, if the input file contains the following text:

```
Benson Eric   Eric  Marty Kim  Kim Kim   Jenny  Nancy Nancy  Nancy  Paul  Paul
Stuart Stuart Stuart Ethan Alyssa Alyssa Helene Jessica Jessica Jessica Jessica
Jared  Alisa Yuki   Catriona  Cody   Coral   Trent Kevin  Ben Stefanie Kenneth
```

On the first line, there is one occurrence of the name Benson, two occurrences of Eric, one occurrence of Marty, three occurrences of Kim, one of Jenny, three of Nancy, and two of Paul. Kim and Nancy appear the most times (3), and Kim appears first in the file. So for that line, your method should print that the most common is `Kim`. The complete output would be the following. The method would also return `23`, since there are 23 unique names in the entire file.

```
Most common: Kim
Most common: Jessica
Most common: Jared
```

You may assume that there is at least one line of data in the file and that each line will contain at least one name.

---

5. Array Programming

Write a static method named **`swapPairs`** that accepts an array of strings as a parameter and switches the order of values in a pairwise fashion. Your method should switch the order of the first two values, then switch the order of the next two, and so on. For example, if the array initially stores these values:

```
String[] a = {"four", "score", "and", "seven", "years", "ago"};
swapPairs(a);
```

Your method should switch the first pair ("four", "score"), the second pair ("and", "seven") and the third pair ("years", "ago"), to yield this array:

```
{"score", "four", "seven", "and", "ago", "years"}
```

If there are an odd number of values, the final element is not moved. For example, if the original list had been:

```
{"to", "be", "or", "not", "to", "be", "hamlet"}
```

It would again switch pairs of values, but the final value ("hamlet") would not be moved, yielding this list:

```
{"be", "to", "not", "or", "be", "to", "hamlet"}
```

You may assume that the array is not `null` and that no element of the array is `null`.

---

6. Array Programming

Write a static method named **`banish`** that accepts two arrays of integers *a1* and *a2* as parameters and removes all occurrences of *a2*'s values from *a1*. An element is "removed" by shifting all subsequent elements one index to the left to cover it up, placing a 0 into the last index. The original relative ordering of *a1*'s elements should be retained.

For example, suppose the following two arrays are declared and the following call is made:

```
int[] a1 = {42, 3, 9, 42, 42, 0, 42, 9, 42, 42, 17, 8, 2222, 4, 9, 0, 1};
int[] a2 = {42, 2222, 9};
banish(a1, a2);
```

After the call has finished, the contents of `a1` should become:

```
{3, 0, 17, 8, 4, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Notice that all occurrences of the values `42`, `2222`, and `9` have been removed and replaced by 0s at the end of the array, and the remaining values have shifted left to compensate.

Do not make any assumptions about the length of the arrays or the ranges of values each might contain. For example, each array might contain no elements or just one element, or very many elements. (If *a2* is an empty array that contains no elements, *a1* should not be modified by the call to your method.) You may assume that the arrays passed are not `null`. You may assume that the values stored in *a2* are unique and that *a2* does not contain the value `0`.

You may not use any temporary arrays to help you solve this problem. (But you may declare as many simple variables as you like, such as `int`s.) You also may not use any other data structures or complex types such as `String`s, or other data structures that were not taught in CSE 142 such as the `ArrayList` class from Chapter 10.

---

7. Critters

Write a critter class **`Raptor`** along with its movement and eating behavior. All unspecified aspects of `Raptor` use the default behavior. Write the complete class with any fields, constructors, etc. necessary to implement the behavior.

In the absence of food, `Raptor`s move horizontally. When a `Raptor` is constructed, he is passed a `boolean` value that indicates whether he will initially walk west (`false`) or east (`true`). The `Raptor` should remember this value and should continue walking in that direction until he finds food. If a `Raptor` finds food, he should eat it, and then "stomp" up and down 10 times by moving north-south 10 times. In other words, his next twenty moves after finding food should be N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S. After finishing his ten "stomps" the `Raptor` should resume moving horizontally, but reverse the direction from west to east or vice versa. For example, if the `Raptor` had been moving west and then finds food, he will move east once he is done stomping.

(If a `Raptor` finds another food while stomping, he will start over his stomp behavior with another 10 N/S moves.)

The following would be a possible sequence of moves for a `new Raptor(false)`:

- W,W,W,W,W,W *(eats food)*, N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S E, E, E, *(eats food)*, N,S,...

The following would be a possible sequence of moves for a `new Raptor(true)`:

- E,E,E,E,E,E,E,E,E,E,E,E *(eats food)*, N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S,
  W,W,W,W,W,W,W,W *(eats food)*, N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S,N,S, E, E, E, ...

8. Classes and Objects

Suppose that you are provided with a pre-written class `BankAccount` as described at right. (The headings are shown, but not the method bodies, to save space.) Assume that the fields, constructor, and methods shown are already implemented. You may refer to them or use them in solving this problem if necessary.

```
// A BankAccount keeps track of a
// user's money balance and ID (name),
// and counts how many transactions
// (deposits/withdrawals) are made.

public class BankAccount {
    private String id;
    private double balance;
    private int transactions;

    // Constructs a BankAccount
    // object with the given id, and
    // $0 balance and 0 transactions.
    public BankAccount(String id)

    // returns the field values
    public double getBalance()
    public String getID()
    public int getTransactions()

    // Adds the amount to the balance
    // if it is between 0-500.
    // Also counts as 1 transaction.
    public void deposit(double amount)

    // Subtracts the amount from
    // the balance if the user has
    // enough money.
    // Also counts as 1 transaction.
    public void withdraw(double amount)


    // your method would go here

}
```

Write an instance method named **`transfer`** that will be placed inside the `BankAccount` class to become a part of each `BankAccount` object's behavior. The `transfer` method moves money from this bank account to another account. The method accepts **two parameters**: a second `BankAccount` to accept the money, and a real number for the amount of money to transfer.

There is a **$5.00 fee** for transferring money, so this much must be deducted from the current account's balance before any transfer.

The method should modify the two `BankAccount` objects such that "this" current object has its balance decreased by the given amount plus the $5 fee, and the other `BankAccount` object's balance is increased by the given amount. A transfer also counts as a transaction on both accounts.

If this account object does not have enough money to make the full transfer, transfer whatever money is left after the $5 fee is deducted. If this account has under $5 or the amount is 0 or less, no transfer should occur and neither account's state should be modified.

For example, given the following `BankAccount` objects:

```
BankAccount benson = new BankAccount("Benson");
benson.deposit(90.00);
BankAccount martin = new BankAccount("Marty");
martin.deposit(25.00);
```

Assuming that the following calls were made, the balances afterward are shown in comments to the right of each call:

```
benson.transfer(martin, 20.00);   // B $65, M $45   (B loses $25, M gains $20)
benson.transfer(martin, 10.00);   // B $50, M $55   (B loses $15, M gains $10)
benson.transfer(martin, -1);      // B $50, M $55   (no effect; negative amount)
martin.transfer(benson, 39.00);   // B $89, M $11   (M loses $44, B gains $39)
martin.transfer(benson, 50.00);   // B $95, M $ 0   (M loses $11, B gains $ 6)
martin.transfer(benson,  1.00);   // B $95, M $ 0   (no effect; no money in acct)
benson.transfer(martin, 88.00);   // B $ 2, M $88   (B loses $93, M gains $88)
benson.transfer(martin,  1.00);   // B $ 2, M $88   (no effect; can't afford fee)
```

Write your answer on the **next page**.

Problem 8 additional writing space

X. Extra Credit (+1 point)

Write a short poem, sonnet, haiku, limerick, etc. about your TA.
Make sure to write your TA's name in it so we know who it is about!

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