UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFORMATICS 1 - OBJECT-ORIENTED PROGRAMMING

Wednesday 2 nd May 2012

09:30 to 12:30

Convener: J Bradfield
External Examiner: A Preece

INSTRUCTIONS TO CANDIDATES

1. **Note that all questions are compulsory.**
2. **Different questions may have different numbers of total marks. Take note of this in allocating time to questions.**
3. **This is an Open Book exam.**

Question 1

In each of parts (a)–(c) below, you will be asked to supply the body to a method inside a class. There will be a separate class for each part, named `OneA`, `OneB` and `OneC` respectively. You will be given a skeleton file for each of these classes, and the skeleton will contain the appropriate method declaration. You should add your definitions of the methods at the points marked as follows:

```
// ADD CODE HERE
```

(a) In the class `OneA`, implement the static method

```
int prodOfPairs(int[] nums)
```

to compute the sum of the products of successive pairs of elements in a list of even length, as shown below. If the input to the method is empty, the method should return zero, and if the input is of an odd length, it should return -1.

Expected behaviour:

```
prodOfPairs(new int[] {1, 2, 3, 4}) = 1*2 + 3*4 -> 14
prodOfPairs(new int[] {3, 5, 7, 5, -2, 4}}) = 3*5 + 7*5 + (-2)*4
                                               -> 42
prodOfPairs(new int[] {}) -> 0
prodOfPairs(new int[] {1, 2, 3}) -> -1
```

[12 marks]

(b) In the class `OneB`, implement the following static method:

```
double meanColSums(int[][] matrix)
```

This takes as input an $N \times M$ matrix of integers, and calculates the mean of the sum of each column. It returns the mean as a `double`. You can assume that `matrix` is non-empty and is encoded as `matrix[row][col]`.

For example, given the 3×3 table of integers shown below, the column sums are 22, 7, 21, and the mean of the column sums comes out as $50/3 = 16.67$ to two d.p.

12	1	14
2	5	5
8	1	2
22	7	21

Expected behaviour:

```
meanColSums(new int[][] {{12, 1}, {2, 5}, {8, 1}}) -> 14.5
meanColSums(new int[][] {{12, 1, 14}, {2, 5, 5}, {8, 1, 2}}) ->
                                            16.666666666666668
meanColSums(new int[][] {{1, 0, 0, 0, 0}}) -> 0.2
meanColSums(new int[][] {{0}}) -> 0.0
```

[*15 marks*]

(c) Clustering involves inspecting a collection of 'points' and grouping them according to some distance measure. In this question, the points are alphabetic strings of length 5, and the measure is Hamming distance, which simply counts the number of characters in which two strings of equal length differ. Given a list of words, we can create two clusters by the following method. Pick an initial point w_0, say the first word in the list, and then find all words within a certain distance of w_0, say Hamming distance of 2. Then pick as a second point the word w_1 with the greatest Hamming distance from w_0, and find all the words within Hamming distance 2 of w_1. This gives two initial clusters. Note that in this case, there are still some points which remain unclustered.

The task is broken down into subparts, each of which involves implementing the body of a static method within the class `OneC`.

(i) Implement the static method

`int hammingDist(String left, String right)`

You can assume that the input strings `left` and `right` are of equal length. The method should return an integer n which is the number of positions at which the corresponding symbols are different. Alternatively, we can think of n as being the number of substitutions required to convert one string to the other.

Expected behaviour:

```
hammingDist("abaca", "abaca") -> 0
hammingDist("abaca", "aback") -> 1
hammingDist("abaca", "abaft") -> 2
hammingDist("abaca", "adapt") -> 3
hammingDist("abaca", "accoy") -> 4
hammingDist("abaca", "actor") -> 4
```

[*4 marks*]

(ii) Implement the static method

`String findFarthest(String s, String[] targets)`

Given an input array `targets` of type `String[]`, this should return the string in the array which has the greatest Hamming distance from `s`. If there is more than one such string, return the first one found.

If `targets` is bound to the array {`"abaca", "aback", "abaft", "adapt", "accoy", "actor"`}, then expected behaviour is as follows:

```
findFarthest("abaca", targets) -> "accoy"
```

Notice that in this case, `"accoy"` and `"actor"` are equally far from `"abaca"`, and the former is returned only because it appears earlier in the list `targets`.

[*4 marks*]

(iii) Implement the static method

```
ArrayList<String> findNearestK(String s, String[] targets, int k)
```

which returns an `ArrayList<String>` of all strings whose Hamming distance from `s` is at most `k`. The return value of this method should contain `s` itself.

Expected behaviour (with `targets` defined as before):

```
findNearestK("abaca", targets, 2) -> ["abaca", "aback", "abaft"]
```

[*4 marks*]

(iv) Hamming distance is limited by requiring the two strings being compared to have equal length. Implement a more general method

```
int stringDist(String left, String right)
```

which removes this restriction. Your implementation of `stringDist()` should involve two stages. Suppose we are given two strings `s` and `t` of unequal length, where `s` is longer than `t`. First, truncate the extra n characters from the *end* of `s`, resulting in a shorter string `s'`. Compute the Hamming distance between `s'` and `t`. Second, count how many extra characters were removed to convert `s` to `s'`, and add this number to the Hamming distance. Return the resulting sum.

Note that `stringDist()` should return the same value regardless of the order of its arguments.

Expecte behaviour:

```
stringDist("heat", "heater") -> 2
stringDist("heater", "heat") -> 2
stringDist("heat", "hatter") -> 4
```

[*5 marks*]

(d) Create a class `QuestionOneTester` with a single `main()` method. Inside `main()`, add calls to the static methods

```
OneA.prodOfPairs()
OneB.meanColSums()
OneC.<methodOfYourChoice>
```

that you implemented for parts (a)–(c) above, in order to test that your implementations produce the correct results. (Remember that for a client program to call a static method from an external class, the method name must be qualified by the class name, as shown.) You are recommended to have your tests simply print out the value of the methods for some appropriate input arguments. Write at least one such test for each of the three classes specified. In the case of `OneC`, you can choose which method to test. [*6 marks*]

The files that you must submit for this question are the following:

(a) `OneA.java`

(b) `OneB.java`

(c) `OneC.java`

(d) `QuestionOneTester.java`

Question 2

This question focusses on implementing data types for expressions built from variables, sums and products. Here is a grammar for the type:

```
Expr ::= Var(<string>)  | Expr "*" Expr | Expr "+" Expr
```

We will create variables using the `Var` constructor:

```
Var x = new Var("x");
Var y = new Var("y");
Var z = new Var("z");
```

Complex expressions will be created as instances of `BinaryExpr`:

```
BinaryExpr e0 = new BinaryExpr(x, Op.PRODUCT, y);
BinaryExpr e1 = new BinaryExpr(x, Op.SUM, y);
BinaryExpr e2 = new BinaryExpr(e0, Op.SUM, z);
BinaryExpr e3 = new BinaryExpr(e1, Op.PRODUCT, z);
BinaryExpr e4 = new BinaryExpr(e0, Op.PRODUCT, z);
BinaryExpr e5 = new BinaryExpr(e1, Op.PRODUCT, e1);
```

The sum and product operators are assumed to be enumerated types (i.e., `Op.SUM` and `Op.PRODUCT`); these are implemented in the file `Op.java`, which is supplied to you.

The classes `Var` and `BinaryExpr` will both extend the superclass `Expr`, and will both implement the `toString` method.

```
x.toString()  -> "x"
e0.toString() -> "(x * y)"
e1.toString() -> "(x + y)"
e2.toString() -> "((x * y) + z)"
e3.toString() -> "((x + y) * z)"
e4.toString() -> "((x * y) * z)"
e5.toString() -> " ((x + y) * (x + y))"
```

An expression is *a term* if it is a variable, or is the product of two expressions that are terms. For example:

```
x.isTerm()  -> true
e0.isTerm() -> true
e1.isTerm() -> false
e2.isTerm() -> false
e3.isTerm() -> false
e4.isTerm() -> true
e5.isTerm() -> false
```

An expression is *normal* if it is a term, or the sum of two expressions that are normal.

```
x.isNorm()  -> true
e0.isNorm() -> true
e1.isNorm() -> true
e2.isNorm() -> true
e3.isNorm() -> false
e4.isNorm() -> true
e5.isNorm() -> false
```

(a) `Expr` will be an abstract class and should meet the following API:

`public class Expr`

`boolean isTerm()`	*abstract method*
`boolean isNorm()`	*abstract method*
`Expr normalize()`	*normalize this expression*
`Expr getLeft()`	*get the left subexpression of this expression, if it exists*
`Expr getRight()`	*get the right subexpression of this expression, if it exists*
`Op getOp()`	*get the main operator of this expression, if it exists*

Implement `normalize()` so that it returns the expression itself (i.e., using `this`). Implement `getLeft()`, `getRight()` and `getOp()` to return `null`. [*8 marks*]

(b) Implement the class `Var` to extend `Expr`.

Here is the API for the `Var` data type:

`public class Var`

`Var(String symbol)`	*constructor*
`boolean isTerm()`	*return* ***true***
`boolean isNorm()`	*return* ***true***
`String toString()`	*return the variable's symbol*

[*4 marks*]

(c) Products and sums should both be implemented as instances of `BinaryExpr`, which extends `Expr`.

Here is the API for the `BinaryExpr` data type:

public class BinaryExpr	
BinaryExpr(Expr l, Op op, Expr r)	*constructor*
boolean isTerm()	`true` *if the expression is a term*
boolean isNorm()	`true` *if the expression is normal*
Expr getLeft()	*get the left subexpression of this expression*
Expr getRight()	*get the right subexpression of this expression*
Op getOp()	*get the main operator of this expression*
String toString()	*return a string representing the expression*
Expr normalize()	*normalize this expression*

The task of implementing `BinaryExpr` is broken down into sub-tasks.

(i) Define instance variables for the class `BinaryExpr`, declare the constructor given in the API above, and define the instance methods `getLeft()`, `getRight()` and `getOp()`. [*4 marks*]

(ii) Define the instance methods `isTerm()` and `isNorm()`. The first of these returns `true` if and only if the `Expr`'s left and right subexpressions are both terms and the operator is product. The method `isNorm()` returns `true` when **either** of the following two conditions is met: (i) the `Expr` is a term, or (ii) the `Expr`'s left and right subexpressions are both normal and the operator is sum. [*12 marks*]

(iii) Define the instance method `toString()` so that it produces output of the kind indicated at the start of this question. For example:

```
e0.toString() -> "(x * y)"
e1.toString() -> "(x + y)"
e2.toString() -> "((x * y) + z)"
e3.toString() -> "((x + y) * z)"
```

Note that the string representation of the operators has been implemented for you in `Op.java`. [*4 marks*]

(iv) Define the instance method `normalize()`. This converts an expression to an equivalent expression in normal form. An expression not in normal form may be converted to normal form by repeated application of the distributive laws:

$$
\begin{aligned}
(a+b)\times c &= (a\times c)+(b\times c)\\
a\times(b+c) &= (a\times b)+(a\times c)
\end{aligned}
$$

For example:

```
x.normalize()  -> "x"
e0.normalize() -> "(x * y)"
e3.normalize() -> "((x * z) + (y * z))"
e5.normalize() -> "(((x * x) + (x * y)) + ((y * x) + (y * y)))"
```

[12 marks]

(d) Implement a class `ExprClient` containing a `main()` method. Credit will be given for implementing code to test `isTerm()`, `isNorm()` and `normalize()`. *[6 marks]*

The files that you must submit for this question are the following:

- `Expr.java`
- `Var.java`
- `BinaryExpr.java`
- `ExprClient.java`

Final Checklist

Here is a complete list of all the files required for this exam:

```
OneA.java
OneB.java
OneC.java
QuestionOneTester.java
Expr.java
Var.java
BinaryExpr.java
ExprClient.java
```