22c:022 Object-Oriented Software Development
Fall 2013

Homework 2

Due: Thursday, Sep 26 by 9pm

This is a programming assignment in Scala to be done *individually.* The homework is in two parts. Download from ICON the accompanying files `hw2a.scala` and `hw2b.scala`, complete as instructed below, and submit them through ICON.

Your files *must generate no syntax or type errors.* You may receive no credit for each subproblem whose code contains such errors.
Pay close attention to the problem specification and the restrictions imposed on its solution. Solutions ignoring the restrictions may receive no credit.

This assignment is meant to exercise your general programming skills, your understanding of basic OO programming concepts, your newly acquired knowledge of Scala, and your ability to read and understand specifications. It is in your best interest to you do all the assigned Scala readings so far before working on the problems below.

Part A

In this part you are to develop some Scala classes implementing *dictionaries.* These are mutable data structures that generalize the notion of a physical dictionary. They are essentially a set of key/value pairs where keys are Scala strings (the words in the dictionary) and the associated values are arbitrary Scala objects, of any type (the word definitions). As with real dictionaries, where a given word can have zero, one or more definitions, it is possible for a key to be associated with zero, one or more values in the dictionary. Operations include creating an empty dictionary, adding or removing a key/value pair, getting the value (or one of the values) associated with a key, and so on.

1. A more precise specification of the operations is provided in the file `hw2a.scala` in an abstract class called `Dictionary`. All the methods in that class are abstract except for `getAll` and `removeAll`. You are to provide an implementation of the latter directly in `Dictionary`.

2. The file also contains an incomplete definition of a concrete subclass of `Dictionary` called `ListDictionary`. You are to complete this definition by implementing there all the abstract methods of `Dictionary`. Use a list of type `List[(String,Any)]` to store the key/value pairs of the dictionary.

You are free to implement the various operations as you want as long as (i) you satisfy the restrictions above and the method specifications provided in `hw2a.scala`, and (ii) you do not modify the public interface of either class.[1]

Note: The operation `get` is supposed to return values of type `Option[Any]`. Option types are described In Section 15.6 of the Scala textbook.

Part B

In this part you are to develop a concrete, immutable Scala class called `Matrix` implementing matrices of real numbers are used in mathematics. Intuitively, a matrix is a two-dimensional $m \times n$ array of real numbers for some $m, n > 0$ such as the following:

$$\begin{vmatrix} 4.2 & 0.2 & 9.3 & 4.1 \\ 1.2 & 8.0 & 0.5 & 0.5 \\ 0.7 & 2.2 & 4.6 & 4.5 \end{vmatrix}$$

Abstractly, a matrix is a collection of real numbers, each with an associated position given by two coordinates: a *row* i and a *column* j. In a $m \times n$ matrix, the rows range from 0 to $m-1$ and the columns from 0 to $n-1$. For convenience, we define the *height* an $m \times n$ matrix to be m, its number of rows, and the *width* to to be n, its number of columns.

Typical matrix operations, yielding another matrix, are the transpose of a matrix, the addition of two matrices with the same height and width, the concatenation of two matrices with the same height, the stacking of two matrices with the same width, the multiplication of an $m \times n$ matrix by an $n \times p$ matrix.

A similar but simpler concept is that of a *vector*, which is essentially a n-tuple of real numbers associated with the positions $0, 1, \ldots, n-1$ for $n > 0$. Typical vector operations are addition and concatenation, which yields a new vector, and scalar product, which yields a real number.

The file `hw2b.scala` contains an implementation of a class `Vec`, modeling vectors in the sense above, and an incomplete implementation of a class `Matrix` modeling matrices. The latter class has methods for creating matrices, concatenating (`++`), stacking (`/`), and adding (`+`) them together, checking that they are equal (`equals`), generating their transpose (`transpose`), and providing a string representation for them (`toString`).

1. Study the implementation of `Vec`, which is devoid of comments purpose, and try to understand what each method does and how. For that it might be helpful to generate singleton vectors (e.g., with `new Vec(3.2)`) and experiment with adding and concatenating vectors together.[2]

2. Complete the definition of `Matrix` as instructed in `hw2b.scala` by implementing those fields and methods that now have just a dummy implementation. For uniformity and convenience we require that `Matrix` store the matrix elements as an array of rows, where each row is implemented as an instance of `Vec`. The vector at position i in the array collects of all the matrix elements in row i, from column 0 onward.

[1]Which means, in particular, that any auxiliary method or field you use should be hidden in one the two classes.
[2]This is a homework activity you do not need to report anything about.

Three constructors are already defined for you. The default one, which is private, and so accessible only by the other methods in `Matrix`, takes as input an array of m vectors of the same length n and constructs an $m \times n$ matrix with elements from the input array. The first of the two auxiliary constructors, which are both public, takes a vector v of length n and produces a $1 \times n$ matrix with elements from v. The other auxiliary constructor takes a `Double` x and produces the 1×1 matrix storing x.

Note that while external users of `Matrix` do not have access to the general matrix constructor, they can build matrices with multiple rows and columns by stacking or concatenating together smaller matrices.[3] For instance,

- `new Matrix(5.4)` constructs the matrix $\begin{vmatrix} 5.4 \end{vmatrix}$
- `new Matrix(1.0) ++ new Matrix(2.0)` constructs the matrix $\begin{vmatrix} 1.0 & 2.0 \end{vmatrix}$
- if `m1` is $\begin{vmatrix} 1.0 & 2.0 \end{vmatrix}$ and `m2` is $\begin{vmatrix} 3.0 & 4.0 \\ 5.0 & 6.0 \end{vmatrix}$ then `m1 / m2` constructs the matrix $\begin{vmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{vmatrix}$
- and so on.

As for Part 1, you are free to implement the various operations as you want as long as (i) you satisfy the restrictions above and the method specifications provided in `hw2b.scala`, and (ii) you do not modify the public interface of either class.

3. **Optional, Extra Credit.** Storing an $m \times n$ matrix internally as an array of vectors requires an amount of space proportional to $m \cdot n$. This is rather inefficient for *sparse matrices*, matrices most of whose elements are zero, because then the vector array is used mostly to store zeros.

   Define a subclass `SparseMatrix` of `Matrix` which has the *same interface and behavior* as `Matrix` but needs an amount of space proportional only to the number of non-zero values in the matrix.

   For the purposes of this problem it is better to relax the privacy requirement for the main constructor of Matrix by making it protected instead of private. In other words, for this problem you may replace

   ```
   class Matrix private (a:Array[Vec]) { ... }
   ```

   with

   ```
   class Matrix protected (a:Array[Vec]) { ... } .
   ```

   Note: If you are careful about your implementation of the methods of `Matrix`, you will not need to re-implement all of them in `SparseMatrix` as a consequence of the different way you store matrix elements in that subclass.

[3] Note that the first auxiliary constructor is actually redundant, since a $1 \times n$ matrix can be obtained as the concatenation of n 1×1 matrices.