Structure & Performance

In this exercise we will examine the performance of various implementations of the same interface. The exercise has three goals:

- To help you relate the structure of the code to the performance of the program. For example, what is the performance cost (or benefit) of immutability, what is the performance costs of using interfaces, and so on.
- To help you understand non-structural issues that affect performance. There are other things besides the structure of the program that affect performance, and the experiments in this exercise should help you understand at least some of them.
- To demonstrate a robust experimental methodology for performance experiments. To be able to draw conclusions from a performance experiment, you must be confident that it indeed measures what you think it does and that the measurements represent some useful metric, such as average running time. It takes some care to ensure that this is indeed the case. The methodology that we use in this exercise is not the best possible, but it is reasonable.

In the exercise, you will implement several classes that model rectangular real matrices. All of the classes are subclasses of the abstract base class `Matrix`, with the following contract:

```
public abstract class Matrix {
  Abstract state: an m-by-n matrix A
  public final int m; number of rows
  public final int n; number of columns

  public Matrix(int m, int n) {...} sets m and n; requires m>0 and n>0

  abstract public void set(int i, int j, double v); sets A_ij
    requires: 0 <= i < m and 0 <= j < n
    ensures: A_ij == v

  abstract public double get(int i, int j); returns A_ij
    requires: 0 <= i < m and 0 <= j < n
    ensures: returns A_ij

  public void random() {...} sets all the elements to random values
    requires: nothing
    ensures: for all i and j sets A_ij to a random value

  public String multiply(Matrix B, Matrix C) {...} matrix multiply add
    requires: A.m == B.m and A.n == C.n and B.n == C.m
    ensures: A = A + B*C  and

  returns the name of the class that actually computed the product
}
```

You will get this abstract base class, as well as two complete working implementations and a testing class. The testing class is called `MatrixTest`. Its `main` procedure creates triplets of three 350-by-350 matrices and calls a testing procedure (`test`) three times on each triplet. The testing procedures fills the matrices with random numbers, saves a copy of one matrix (A), and then multiplies A=A+A*B. Finally, the testing procedure verifies that the matrix multiply-add was computed correctly. If the product is incorrect (more precisely, if it is very inaccurate), the testing routine throws an arithmetic exception. This should help you test you code. The `main` procedure also prints out the vendor and version of the Java virtual machine and of the operating system, to help identify the platform where the experiments were carried out.

The testing procedures measures the running time of the matrix multiply-add and prints out the name of the class that performed the multiplication, the running time in milliseconds, and the computational rate in millions of floating-point operations per second.

We provide you with two subclasses of the abstract base class. One is called `DoubleMatrix`. It represents the matrix using a two-dimensional array (really an array of arrays) of references to `java.lang.Double` objects. These `Double` objects contain a single `double` primitive value, and are immutable. The class overrides the multiply method: if the two arguments are also of the type `DoubleMatrix`, it casts them and multiplies them directly and returns the name of the class (`DoubleMatrix`) by calling `getClass().getName()`. Otherwise, it calls `super.multiply`. All your implementations should follow this pattern.

The second subclass that we provide is called `NativeMatrix`. It represents the matrix using a one dimensional array of `double` primitive values (stored column by column), and it performs the matrix multiply-add operation by calling a so-called native procedure, a procedure that is implemented in C, not in Java. We will provide you with the C code, that you have to compile into a shared library (`.dll` on Windows, `.so` on Linux and Unix), and with a compiled shared library for windows.

The Assignment

1. Study carefully the code in `Matrix`, `MatrixTest`, and `DoubleMatrix`. Why do you think we print the value that `multiply` returns rather than print fixed labels? Why do we print the version of the Java virtual machine directly from the program?
2. We have also provided you with an interface `Real` that models real numbers. It declares three methods, `double get()`, `void set(double)`, and `void mutiplyAdd(Real,Real)`. Calling `x.multiplyAdd(y,z)` should set `x` to the value `x+y*z`. Implement a final class `DoubleReal` that implements `Real` using a single `double`. Objects from this class should be mutable, but the class itself should be declared `final`. In this class, add a method `multiplyAddDoubleReal( DoubleReal,DoubleReal)`, which should be more efficient than `multiplyAdd`.
3. Now implement a class `FinalMatrix` that extends `Matrix`. It should be fairly similar to `DoubleMatrix`, except that it should use an array of references to `DoubleReal`, not `Double`, and it should exploit the method

multiplyAddDoubleReal. To test it, comment out in MatrixTest.main the creation and testing calls to the yet-unimplemented classes InterfaceMatrix and PrimitiveMatrix.

4. Next, implement a class InterfaceMatrix that is similar to FinalMatrix except that it should use references to the interface Real, not to the class DoubleReal. The constructor of InterfaceMatrix should construct DoubleReal directly (we could have used a factory, but let's keep things simple).
5. Finally, implement a class PrimitiveMatrix that also extends extends Matrix. Make it as fast as you can, but implement it in Java (no C code). We will give you two hints. First, a representation like the one we used in NativeMatrix, a one-dimensional array of double's, is probably the most effective. Second, suppose that you multiply the matrices using three nested loops of the form
```
for i=0..m-1
  for j=1..n-1
    for k=1..l-1
     Aij = Aij + Bik * Ckj
    end
  end
end
```
then any ordering of the loops is valid. Reordering the loops affects two issues: the distance, in elements, between consecutive elements of A, B, and C that you use, and the number of times Aij is mutated. If you store the elements of matrices column after column in a one-dimensional array, then the j-k-I ordering accesses all three arrays sequentially (A and C are accessed once, C is accessed B.n times).
6. Now run the program to collect results. Use the results of the experiments to tune the code, especially to tune class PrimitiveMatrix. Try to run the program under more than one Java virtual machine, and perhaps under more than one machine. Submit all the outputs, and document the machine that ran each experiment (processor and processor speed).
7. Try to explain the results as best you can. In particular, try to explain why the different implementations perform differently, why different JVMs behave differently (if they do), why different runs of exactly the same code behave differently (if they do), and so on. Try to draw useful conclusions, but do not draw more conclusions that the data supports; if you need to, collect more data.
8. (Extra Credit) Make the C implementation faster. You almost certainly can. Remember that compiler optimization options also play a big role, not just the code that you write.
9. (Extra Credit) Extend the program to multiply matrices of different sizes to inspect size-dependent behavior, and explain the behaviors that you find. Graphing the results can probably help.