# Task No. 1: Matrices

Ver. 1.01 (10 pts.)

Task issued: May 1, 2022

Matrices of real numbers of size n*m (n, m are positive integers) can be represented in different ways. The most obvious one is a two-dimensional array of numbers of type double. However, if we know something more about the content of the matrix, we can choose other solutions. This is especially true when we do not have enough memory to store every element of the matrix. For example, we may not remember those elements that occur most often. Typically, the most frequently repeating value is zero. Matrices in which values different from this most common one rarely occur are called sparse.

Sparse matrices can have a very regular form. Examples of such matrices are the zero matrix (contains only zeros), constant matrix (is filled entirely with only one value, e.g. the zero matrix is completely filled with the value zero), square unit matrix (has ones on the main diagonal, let's assume that on the one where the indices are equal to each other, otherwise zeros), square diagonal matrix (has values only on the main diagonal, again let's assume that on the one where the indices are equal to each other), column matrix (all values in each row are equal to each other, rows are probably different), row matrix (all values in each column are equal to each other, columns are probably different).

There are also sparse matrices with irregular structures. We can represent them in various ways. A rather obvious one is an array of rows represented by lists - the next sparse (i.e., different from the most frequently repeating, e.g., non-zero) values linked by references. There are of course other possibilities (column list array, list of lists, dictionary, ...). For irregular matrices, you should choose one of the implementations presented in this paragraph (excluding the dictionary) and include only this one in the solution.

Various operations can be performed on matrices. The basic one is of course the provision of a single element. Others are multiplication (as well as addition and subtraction) by another matrix or by a scalar (a number of type double). Another group of operations is the calculation of norms (e.g. maximum norms or Frobenius norms). Other operations include representing the matrix as a two-dimensional array of double numbers, or as a string (toString). Of course, the matrix should be able to provide its dimensions.

The aim of this task is to create a set of classes implementing matrices in various (as above) ways. You should ensure that a uniform interface is implemented for all matrices and try to include repeated operations in the abstract class/classes.

Error situations (e.g., mismatched matrix sizes) should be detected and handled using the assert instruction (remember that by default, assert checking is disabled, you need to enable it with the -ea option). Soon in class we will discuss another mechanism that fits here - exceptions, but we do not use it in this task yet.

Matrix operations (e.g. norm calculation) should use the matrix type to limit operations (e.g. to reduce the number of operations from n^2 to n). For matrix multiplication, you can use a standard algorithm that works for n*n matrices in n^3 time.

Attached to this task is a zip archive containing:

● IDoubleMatrix interface,

● skeleton of the DoubleMatrixFactory class (without the contents of the methods included there),

● a set of tests using the JUnit library.

The matrices from your solution should implement the IDoubleMatrix interface. The content of the DoubleMatrixFactory class and tests should be supplemented with those classes of matrices that are mentioned in the task, but are not present in the factory/tests. Your solution should meet the given tests. You cannot change the content of the given classes/interfaces, but you can add your own additional methods to the presented classes/interfaces. You can also, if necessary, create auxiliary classes/interfaces.

You can add other (than the ones mentioned above) own tests (of course without modifying or deleting the given ones).

You should implement at least one optimization of the creation of result matrices (i.e., the creation of a matrix that is not a full matrix, e.g., adding two diagonal matrices will result in a diagonal matrix, not a regular one). This optimization should take place before (and not after) the creation of the matrix (so it's not about creating a full matrix, and then checking if it doesn't have a special form). This optimization (or these optimizations) can use additional (more) operations added to the matrix interface or the double dispatch technique or checking whether two objects are objects of the same class (comparing the results of the getClass operation).

Please note that in this task the matrices are immutable (this has various, positive and negative, consequences - in this task we chose immutable ones). Please therefore consider whether this will not allow you to reduce the execution time of some of the operations given in the matrix interface and - if so - you should do so.

When implementing individual matrices, you should use their structure so as not to remember unnecessary values and - if possible - to calculate results faster.

In the main function, you should include printing each type of implemented matrices. Matrices should be of size 10 by 10. Their content can be arbitrary (e.g., random or set in code) consistent with the specifics of a given type of matrix. Each matrix should be printed using the toString operation. Matrices should be printed one row per line (the text generated by toString should contain \n sequences). During printing, matrices (according to their specifics) should replace a series of at least three repetitions in a row of one value w with a record with three dots: w ... w. Note: we are talking about compressing (replacing with a record with dots) repetitions resulting from the type of matrix, if the data stored in the matrix will contain data with repetitions at adjacent positions in the row, you do not need to detect and compress this. You should also calculate and print the results (for example data) of operations performed by your program (one printout for each implemented method).

You cannot use any libraries other than the standard Java library (provided with JDK).

As a solution, you should send a complete (compiling and passing tests) project from the IDE you use. If your solution used some unusual (i.e., non-standard in the IDE you use) project architecture (which we strongly discourage), you should describe exactly how to run your tests. Please also state in a comment in the code which IDE you used. If you have any doubts about the form of the solution you are sending, please contact the person conducting the classes in your group (homework is checked by people conducting classes in laboratory groups).

The solution should be sent as a zipped archive to Moodle (to the task)

Good