COMP103P Coursework

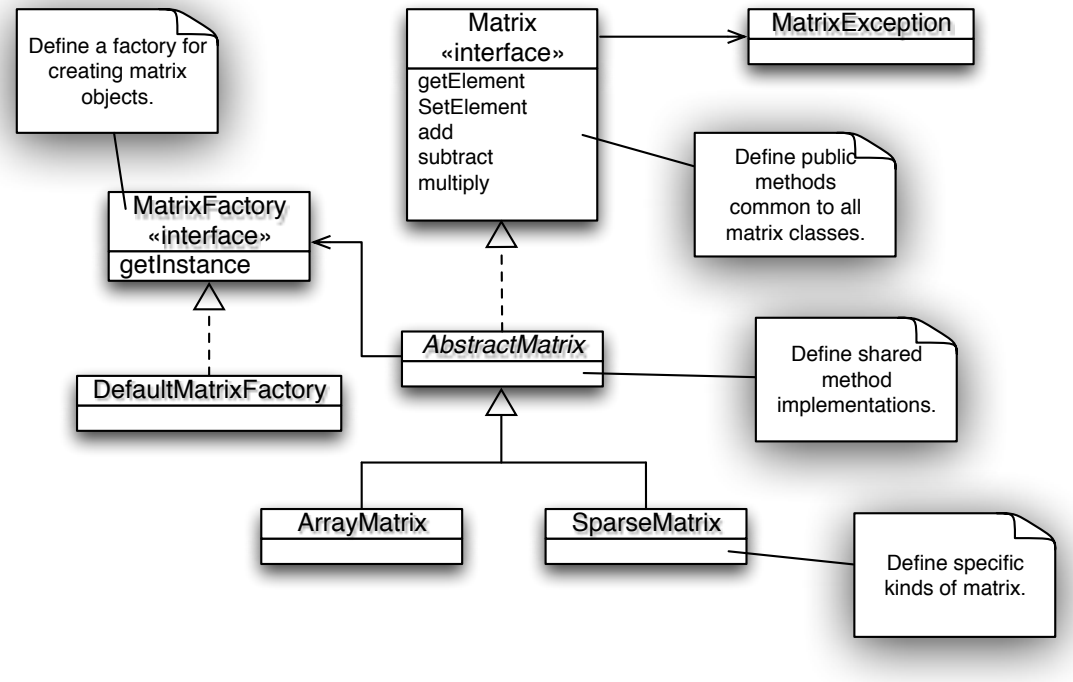
Purpose: A more complex Java programming task to make sure you understand interfaces, abstract classes, inheritance, overriding methods and using exceptions. There are a number of important concepts covered here, make sure you understand them all.

Submission: Complete this coursework by 11.55pm Monday 27th April and submit online via Moodle (instructions will be sent by email). You can submit anytime before then.

Marking: This is an individual graded coursework, with grades A-F.

This coursework is about implementing Matrix classes, making use of interfaces, an abstract class, inheritance, overridden methods, a factory and exceptions. A matrix class represents the familiar rectangular array of numbers, with the size specified by the number of rows and columns. The contents of a matrix are represented by double values. The Java language features needed to implement these classes were all covered in the first half of term.

The goal is to set up a simple framework that will support several classes providing different concrete Matrix implementations. The framework structure is shown below:



This framework is based on the principles used by the data structure classes in the standard Java libraries but is intended to illustrate a range of programming concepts rather than be an actual design for a real framework.

Interface Matrix defines the common methods for all matrix classes. A variable of type Matrix can reference any Matrix object of a class that implements Matrix (or inherits the implements relationship). This enables 'Programming to an Interface', allowing code to be written using the Matrix type without needing to be concerned about the exact type of Matrix of objects it is working with at runtime.

The Matrix interface, documented using Javadoc documentation comments, is defined as follows:

```
/*
```

```
 * Specification of the public methods that all matrix
 * implementations should provide.
 * Copyright (c) 2015
 * UCL Computer Science
 * @author Graham Roberts
 * @version 1.7 Sept 2015
 */

public interface Matrix
{
  /**
   * Return the number of rows in the matrix.
   * @return the number of rows in the matrix.
   */
  public int getNumberOfRows();

  /**
   * Return the number of columns in the matrix.
   * @return the number of columns in the matrix.
   */
  public int getNumberOfColumns();

  /**
   * Return the element at position (row,column).
   *
   * @param row row of element to return.
   * @param column column of element to return.
   * @return element value of element.
   * @throws MatrixException if the element position is not within the
   * matrix.
   */
  public double getElement(final int row, final int column)
    throws MatrixException;

  /**
   * Set element at position (row,column). If the element is not within
   * the matrix, the matrix is left unchanged.
   *
   * @param row row of element to set.
   * @param column column of element to set.
   * @param value value to store at (row,column).
   * @throws MatrixException if the element position is not within the
   * matrix.
   */
  public void setElement(final int row, final int column, final double value)
    throws MatrixException;

  /**
   * Add the argument matrix to <i>this</i> and return a new matrix containing
   * the result.
   * The matrices must match in size for the addition to succeed.
   * This is a generic method that can add matrices of any concrete
   * implementation class.
   * @param m matrix to add to <i>this</i>.
   * @return new matrix containing the result or null.
   * @throws MatrixException if parameter matrix is not the same size
   * as the matrix the method is called for.
   */
  public Matrix add(final Matrix m) throws MatrixException;
```

```
  /**
   * Subtract the argument matrix from <i>this</i> and return a new matrix
   * containing the result.
   * The matrices must match in size for the subtraction to succeed.
   * This is a generic method that can subtract matrices of any concrete
   * implementation class.
   * @param m matrix to subtract from <i>this</i>.
   * @return new matrix containing the result or null.
   * @throws MatrixException if parameter matrix is not the same size
   * as the matrix the method is called for.
   */
  public Matrix subtract(final Matrix m) throws MatrixException;

  /**
   * Multiply the argument matrix with <i>this</i> and return a new matrix
   * containing the result.
   * The matrices must have valid sizes for the multiplication to succeed.
   * This is a generic method that can multiply matrices of any concrete
   * implementation class.
   * @param m matrix to multiply with <i>this</i>.
   * @return new matrix containing the result or null.
   * @throws MatrixException if matrices are not of compatible size
   * to perform multiplication.
   */
  public Matrix multiply(final Matrix m) throws MatrixException;
}
```

Note that a number of methods are declared as throwing exceptions of type MatrixException, if an attempt is made to access an invalid matrix element (i.e., the row,column specified is not in the matrix) or if an invalid operation applied to a matrix. A single exception class is used to represent all errors but the constructors have parameters allowing the message stored in the exception to provide more information. The MatrixException class is:

```
/**
 * Exception subclass to represent exceptions thrown when working
 * with matrix objects.
 * Copyright (c) 2015
 * Dept. of Computer Science, UCL
 * @author Graham Roberts
 * @version 1.1
 */

public class MatrixException extends Exception
{
  /**
   * Provide a custom message.
   * @param message The message to store in the exception object.
   */
  public MatrixException(String message)
  {
    super(message);
  }

  /**
   * Attempt made to access an invalid matrix element.
   * @param row Index of row
   * @param column Index of column
   */
  public MatrixException(int row, int column)
  {
    super("Attempt to access invalid element ("+row+","+column+")");
```

```
  }
}
```

The AbstractMatrix class is an abstract class that provides common methods for concrete subclasses. This avoids duplicating the methods in the subclasses as they can just be inherited. Remember that an abstract class cannot have instance objects, so attempting to create a new AbstractMatrix object will result in a compilation error.

The listing below includes an example add method, which adds two matrices together returning a new matrix as the result. The new matrix object returned is created using a *Factory*. A factory is a class that specialises in creating objects. The use of a factory avoids code having to name a specific concrete set class and decouples object creation from class naming. Factory is an example of a Design Pattern.

The AbstractMatrix class follows:

```
/*
 * An abstract matrix class providing the shared implementation
 * for concrete subclasses.
 * The add, subtract and multiply methods are declared final, so cannot
 * be overridden by subclasses.
 * Copyright (c) 2015
 * UCL Computer Science
 * @author Graham Roberts
 * @version 1.7 Sept 2015
 */

public abstract class AbstractMatrix implements Matrix
{
  private static MatrixFactory factory = new DefaultMatrixFactory();

  private Matrix getNewMatrixInstance(Matrix kind,
    int numberOfRows, int numberOfColumns)
    throws MatrixException
  {
    return factory.getInstance(kind.getClass(), numberOfRows, numberOfColumns);
  }

  /**
   * Set the factory to be used for matrix creation. Note this is a static
   * method so <em>all</em> matrix objects created from now on
   * will be created by the new factory.
   * @param aFactory The factory to use.
   */
  public static void setFactory(MatrixFactory aFactory)
  {
    factory = aFactory;
  }

  public final boolean isSameSize(final Matrix m)
  {
    return (getNumberOfRows() == m.getNumberOfRows())
      && (getNumberOfColumns() == m.getNumberOfColumns());
  }

  public final Matrix add(final Matrix m) throws MatrixException
  {
    if (!isSameSize(m))
    {
      throw new MatrixException("Trying to add matrices of different sizes");
    }
```

```
    final Matrix result =
      getNewMatrixInstance(this, getNumberOfRows(), getNumberOfColumns());
    for (int row = 0; row < getNumberOfRows(); row++)
    {
      for (int column = 0; column < getNumberOfColumns(); column++)
      {
        final double value =
          getElement(row, column) + m.getElement(row, column);
        result.setElement(row, column, value);
      }
    }
    return result;
  }

  public final Matrix subtract(final Matrix m) throws MatrixException
  {
   // The method body is not provided - you should write your own implementation
  }

  public final Matrix multiply(final Matrix m) throws MatrixException
  {
   // The method body is not provided - you should write your own implementation
  }
}
```

Class AbstractMatrix class uses class DefaultMatrixFactory to create new matrix objects but this can be changed at runtime by providing a different factory. As the factory is referenced via a static variable the same factory is used for creating all matrix objects. Allowing this variability is a common strategy to facilitate testing, as test code can provide a special factory for testing purposes.

Interface MatrixFactory defines the public methods needed by a factory. This again allows the Programming to an Interface concept to used in code that uses a factory.

```
/**
 * This interface defines the public methods that a Matrix factory must
 * implement.
 * Copyright (c) 2015
 * UCL Computer Science, UCL
 * @author Graham Roberts
 * @version 1.1
 */

public interface MatrixFactory
{
 /**
   * Create a matrix that is an instance of the same class as the
   * parameter object, with the given matrix size.
   * @param  matrixClass A class object used to determine the class
   * of the matrix to instantiate
   * @param numberOfRows The number of rows in the new matrix.
   * @param numberOfColumns The number of columns in the new matrix.
   * @return The new matrix.
   * @throws MatrixException If the class is not recognised as one from
   * which a matrix object can be created or if an attempt is made to
   * create a matrix with an invalid size.
   */
  public Matrix getInstance(Class<? extends Matrix> matrixClass,
                            int numberOfRows, int numberOfColumns)
   throws MatrixException;
```

```
 /**
   * Create a matrix that is an instance of the same class as the
   * parameter object, with the given content. The matrix size is
   * determined from the content provided.
   * @param  matrixClass A class object used to determine the
   * class of the matrix to instantiate.
   * @param contents The contents of the new matrix.
   * @return The new matrix.
   * @throws MatrixException If the class is not recognised as one from
   * which a matrix object can be created or if an attempt is made to
   * create a matrix with an invalid size.
   */
  public Matrix getInstance(Class<? extends Matrix> matrixClass,
                            double[][] contents)
   throws MatrixException;
}
```

Note that the getInstance factory methods take a value of type Class, a generic class, which is used to determine the class of the matrix object to be created. Java represents classes at runtime with objects of type Class and an object can be asked for its class.

The DefaultMatrixFactory class looks like this:

```
/**
 * Factory to create Matrix objects.
 * Copyright (c) 2015
 * UCL Computer Science
 * @author Graham Roberts
 * @version 1.1
 */

public class DefaultMatrixFactory implements MatrixFactory
{

  public Matrix getInstance(Class<? extends Matrix> matrixClass,
                            int numberOfRows, int numberOfColumns)
    throws MatrixException
  {
    if (matrixClass.equals(ArrayMatrix.class))
    {
      return new ArrayMatrix(numberOfRows,numberOfColumns);
    }
    if (matrixClass.equals(SparseMatrix.class))
    {
      return new SparseMatrix(numberOfRows,numberOfColumns);
    }
    throw new MatrixException
      ("getNewInstance: Class is not a recognised matrix class");
  }

  public Matrix getInstance(Class<? extends Matrix> matrixClass,
                            double[][] contents)
    throws MatrixException
  {
    if (matrixClass.equals(ArrayMatrix.class))
    {
      return new ArrayMatrix(contents);
    }
    if (matrixClass.equals(SparseMatrix.class))
    {
      return new SparseMatrix(contents);
```

```
    }
    throw new MatrixException
      ("getNewInstance: Class is not a recognised matrix class");
  }
}
```

To illustrate how to implement a concrete matrix class as a subclass of AbstractMatrix, here is an outline template of class ArrayMatrix that uses a 2D array of doubles to store the values held in the matrix.

```
public class ArrayMatrix extends AbstractMatrix
{
  private double[][] elements;

  public ArrayMatrix(final int rows, final int columns)
    throws MatrixException
  {
    // Initialise a new matrix with all the elements set to 0.0
  }

  public ArrayMatrix(double[][] content) throws MatrixException
  {
    // Initialise a new matrix storing the data provided by the
    // double[][] parameter. Note the data should be copied.
  }

  public int getNumberOfRows()
  {
    // Number of rows in matrix
  }

  public int getNumberOfColumns()
  {
    // Number of columns in matrix
  }

  public double getElement(final int row, final int column)
    throws MatrixException
  {
    // Return the element at the specified position or throw an exception
  }

  public void setElement(final int row, final int column, final double value)
    throws MatrixException
  {
    // Set the element at the specified position or throw an exception
  }
}
```

Note that class ArrayMatrix throws exceptions *but does not catch any of its own exceptions*. A matrix throws exceptions, while code using a matrix catches any exceptions using try/catch blocks. Make sure you understand the point being made here.

Coursework Questions

Note - all your matrix classes *must* conform to interface Matrix, which *cannot* be changed. The only public methods should be the constructors and the methods declared in interface Matrix that are overridden. DO NOT ADD ANY OTHER PUBLIC METHODS!

Q1. A complete and working answer to this question is worth 50% of the coursework marks.

a) Fill in the template for the ArrayMatrix class shown above to create a complete working class.

b) Write the subtract and multiply method bodies for class AbstractMatrix. These methods will be inherited by all matrix subclasses and should work with any matrix implementation, so should only make use of the methods declared in the Matrix interface. The add method body is already provided as an example. You should not define any add, subtract or multiply methods in the ArrayMatrix class.

c) Implement a sparse matrix subclass called SparseMatrix, following the same basic template as the ArrayMatrix class but with a different data structure.

A sparse matrix behaves exactly like a normal matrix but only values that are not equal to 0.0 are actually stored, the rest are assumed to be 0.0. Hence, the matrix object only stores the value of a matrix element in its data structure if that element is explicitly set to a value other than 0.0 by a call to setElement. If getElement is called to access an element whose value has not been explicitly set then the default value of 0.0 is returned, and no space in the data structure is needed.

A sparse matrix allows a large matrix to be represented in a relatively small amount of memory, providing the majority of elements have the value 0.0. This is useful for certain types of calculation.

To implement a sparse matrix consider using a HashMap as the data structure (but investigate sets as well). This is a kind of hash table and is documented in the online Javadoc documentation. An element in a hash table is accessed using a key value. For the sparse matrix the key value can be the (x,y) coordinates of an element in matrix.

To represent a key value, look at class Index below, which should be *nested* in the scope of the sparse matrix class. The hash table can then be declared as a HashMap<Index,Double>, i.e., a map from an index to a double value, which has to be stored as a Double object as maps cannot store values of primitive types directly (but find out how autoboxing/unboxing helps).

```
private static class Index
{
  private int x = 0;
  private int y = 0;
  private int hashvalue = 0;

  public Index(final int x, final int y)
  {
    this.x = x;
    this.y = y;
    hashvalue =
      ((x +"") + (y + "")).hashCode();
  }

  // Override equals and hashcode to ensure Index objects
  // behave correctly when used as keys in a hash table.
  @Override
  public boolean equals(final Object obj)
  {
    if (obj instanceof Index)
    {
      Index index = (Index) obj;
      return ((x == index.x) && (y == index.y));
    }
    else
    {
```

```
        return false;
      }
    }

    @Override
    public int hashCode()
    {
      return hashvalue;
    }
  }
```

Study this class carefully and use the Javadoc documentation, books and other sources to work out what the equals and hashcode methods are doing. Make sure that the setElement method of the matrix class does not end up adding two values with the same index.

d) Write an example program to demonstrate your matrix implementations work. The program should use all public methods of the matrix classes, including the error handling (e.g., that an exception is thrown when matrices of incompatible sizes are added, subtracted or multiplied).

It should be possible to use both ArrayMatrix and SparseMatrix objects interchangeably in the same program, for example to add an ArrayMatrix to a SparseMatrix. Include code to confirm this does actually work.

Q2. A complete and working answer to this question is worth 25% of the coursework marks.

Write a JUnit test class (or classes) to test your matrix implementations. All public methods should be tested.

Advanced Questions

Complete and working answer to these questions are worth 25% of the coursework marks.

Q3. Implement another sparse matrix subclass using a tree data structure. Add or extend JUnit test class(es) to test it.

Q4. Using Java generics, modify all your classes (and the interface) to create generic classes. This would allow the type of number stored to be determined using declarations like:

```
Matrix<Double> myMatrix = new ArrayMatrix<Double>(10,10);
Matrix<Integer> intMatrix = new SparseMatrix<Integer>(5,6);
```

and for a challenge:

```
Matrix<BigDecimal> big = new SparseMatrix<BigDecimal>(100,100);
```