```java
/**
 * This tests roughly the functions of the class Matrix. {@link Matrix}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       13/11/2023
 */

public class Main {
    public static void main(String[] args) {
        int modulo = 5;
        System.out.println("The modulus is " + modulo);

        Matrix one = new Matrix(new int[][]{{1,3,1,1},
                                            {3,2,4,2},
                                            {1,0,1,0}}, modulo);

        Matrix two = new Matrix(new int[][]{{1,4,2,3,2},
                                            {0,1,0,4,2},
                                            {0,0,2,0,2}}, modulo);

        System.out.println("one");
        System.out.println(one);

        System.out.println("two");
        System.out.println(two);

        System.out.println("one + two");
        System.out.println(one.add(two));

        System.out.println("one - two");
        System.out.println(one.sub(two));

        System.out.println("one x two");
        System.out.println(one.mult(two));

        System.out.println("Test randomness: ");
        System.out.println(new Matrix(4, 4, 7));
        System.out.println(new Matrix(4, 4, 7));
        System.out.println(new Matrix(4, 4, 7));
    }
}
```

```java
/**
 * Class Matrix used to store a matrix, display this matrix and make operations between two matrices.
 * The matrix is made of integers
 * This uses another class named "Operation" to operate each element of the matrix : {@link Operation}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       13/11/2023
 */

public class Matrix {
    private final int m;
    private int n;
    private final int modValue;
    private final int[][] matrix;

    /**
     * Create a randomly filled Matrix by giving its dimensions.
     * @param m number of row
     * @param n number of columns
     * @param modValue value of the modulo
     */
    public Matrix(int m, int n, int modValue) {
        verifyDim(m, n);
        modPos(modValue);
        this.n = n;
        this.m = m;
        this.modValue = modValue;

        this.matrix = new int[m][n];

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                matrix[i][j] = (int) Math.floor(Math.random()
                                * modValue);
            }
        }
    }

    /**
     * Create a Matrix from a given 2d array of int.
     * @param matrix 2d array for the matrix
     * @param modValue value of the modulo
     */
    public Matrix(int[][] matrix, int modValue){

        arrayCheck(matrix);
        modPos(modValue);


        this.m = matrix.length;
        this.n = matrix[0].length;

        //Searching for the longest array in the given matrix
        for(int i = 0; i < m;++i){
            if (this.n < matrix[i].length)
                this.n = matrix[i].length;
        }

        this.matrix = new int[m][n];
        this.modValue = modValue;

        for(int i = 0; i < m; ++i){
            for(int j = 0; j < n; ++j){
                if(j >= matrix[i].length) continue;
                this.matrix[i][j] = Math.floorMod(matrix[i][j],
                                                  modValue);
            }
        }
    }

    public String toString(){
        StringBuilder out = new StringBuilder();
        for(int i = 0; i < m; ++i){
            for(int j = 0; j < n; ++j){
                out.append(matrix[i][j]).append(" ");
            }
            out.append('\n');
        }
        return out.toString();
    }

    /**
     * Applies an operation to all the elements of 2 matrix
     * @param b Other Matrix for the operation
     * @param operation Operation to apply
     * @return  A new Matrix from the application ot
     *          the chosen operation.
     */
    private Matrix matrixOp(Matrix b, Operation operation) {

        verifyMod(b);
        int newM = Math.max(this.m,b.m);
        int newN = Math.max(this.n,b.n);

        int operandA;
        int operandB;

        int[][] newMatrix = new int[newM][newN];

```

```java
        for(int i = 0; i < newM; ++i){
            for(int j = 0; j < newN; ++j){
                //If a matrix is bigger in the n or m dimension
                // than the other one, the smaller matrix sends
                // 0 to avoid out of range error.
                operandA = (i >= this.m || j >= this.n) ?
                            0 : this.matrix[i][j];

                operandB = (i >= b.m|| j >= b.n) ?
                            0 : b.matrix[i][j];

                newMatrix[i][j] =
                        Math.floorMod(operation.operate(operandA,
                                      operandB),modValue);
            }
        }

        return new Matrix(newMatrix, modValue);
    }

    /**
     * Add 2 Matrix together.
     * @param b The other Matrix
     * @return A new Matrix created from the addition
     */
    public Matrix add(Matrix b){
        return matrixOp(b, new Addition());
    }

    /**
     * Subtract 1 Matrix from another.
     * @param b The other Matrix
     * @return  A new Matrix created from the subtraction
     */
    public Matrix sub(Matrix b){
        return matrixOp(b, new Subtraction());
    }

    /**
     * Multiply 2 matrix together.
     * @param b The other Matrix
     * @return A new Matrix created from the multiplication
     */
    public Matrix mult(Matrix b){
        return matrixOp(b, new Multiplication());
    }

    /**
     * Verify that the moduli of this Matrix and another
     * are equals, else it throws and exception.
     * @param b The other Matrix
     * @throws RuntimeException if the moduli of the 2 matrices
     *                           are not equal.
     */
    private void verifyMod(Matrix b) {
        if(modValue != b.modValue)
            throw new RuntimeException("The moduli of the 2 " +
                                       "matrices are not equal");
    }

    /**
     * Verify that a 2d array of int isn't null or empty,
     * else it throws an exception.
     * @param a The 2d array to check
     * @throws RuntimeException if the array passed as parameter
     *                           isn't valid.
     */
    private void arrayCheck(int[][] a){
        if(a == null || a.length == 0 || a[0].length == 0)
            throw new RuntimeException("The array passed as parameter" +
                                       " isn't valid");
    }

    /**
     * Verify that a modulo isn't negative or equal to 0,
     * else it throws an exception.
     * @param mod The modulo to check
     * @throws RuntimeException if the modulo cannot be negative
     *                           or equal to 0.
     */
    private void modPos(int mod){
        if(mod <= 0)
            throw new RuntimeException("The modulo cannot be negative" +
                                       " or equal to 0");
    }

    /**
     * Verify that two int aren't negative or equal to 0,
     * else it throws an exception.
     * @param m One of the int to check
     * @param n The second int to check
     * @throws RuntimeException if one of the dimension
     *                           is lower or equal to 0.
     */
    private void verifyDim(int m, int n){
        if (n <= 0 || m <= 0)
            throw new RuntimeException("One of the dimension is" +
                                       " lower or equal to 0");
    }
}
```

```java
/**
 * This adds operand A and operand B.
 * Children of the abstract class Operation : {@link Operation}
 * Operand A and operand B must be integers
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       13/11/2023
 */

public class Addition extends Operation{

    /**
     * Add the second operand to the first
     * @param opA first operand
     * @param opB second operand
     * @return the first plus the second operand
     */
    @Override
    public int operate(int opA, int opB) {
        return opA + opB;
    }
}

```

```java
/**
 * Abstract class Operation used to operate two operands.
 * Operand A and operand B must be integers
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       13/11/2023
 */
public abstract class Operation {

    /**
     * Applies the operation on the two operands.
     * @param opA first operand
     * @param opB second operand
     * @return the result of the operation
     */
    abstract int operate(int opA,int opB);
}

```

```java
/**
 * This tests the functions of the class Matrix using junit. {@link
Matrix}
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       13/11/2023
 */


import junit.extensions.RepeatedTest;
import org.junit.Test;

import java.lang.annotation.Repeatable;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertThrows;
public class MatrixTest {

    private static final int[][] MATRIX_NOT_FULL = {{1},
                                                     {3,2,4},
                                                     {},
                                                     {1,0,2,3,1,2,1}};

    private static final int[][] MATRIX_NOT_FULL_RES = {{1,0,0,0,0,0,0},
                                                         {3,2,4,0,0,0,0},
                                                         {0,0,0,0,0,0,0},
                                                         {1,0,2,3,1,2,1}};

    private static final int[][] MATRIX_ONE = {{1,3,1,1},
                                                {3,2,4,2},
                                                {1,0,1,0}};

    private static final int[][] MATRIX_TWO = {{1,4,2,3,2},
                                                {0,1,0,4,2},
                                                {0,0,2,0,2}};

    private static final int[][] MATRIX_RESULT_ADD = {{2,2,3,4,2},
                                                       {3,3,4,1,2},
                                                       {1,0,3,0,2}};

    private static final int[][] MATRIX_RESULT_SUB = {{0,4,4,3,3},
                                                           {3,1,4,3,3},
                                                           {1,0,4,0,3}};
    private static final int[][] MATRIX_RESULT_MULT = {{1,2,2,3,0},
                                                       {0,2,0,3,0},
                                                       {0,0,2,0,0}};
    private static final int[][] MATRIX_MOD = {{-1,2,-2,3,0},
                                           {0,2,0,3,-465},
                                           {-9,0,2,0,0},
                                           {-7,-8,345,0,1}};
    private static final int[][] MATRIX_RESULT_MOD =  {{7,2,6,3,0},
                                                        {0,2,0,3,7},
                                                        {7,0,2,0,0},
                                                        {1,0,1,0,1}};
    @Test
    public void testAdd(){

        Matrix matrixA = new Matrix(MATRIX_ONE,5);
        Matrix matrixB = new Matrix(MATRIX_TWO,5);

        Matrix matrixResAdd = new Matrix(MATRIX_RESULT_ADD,5);

        assertEquals("Test: add Matrix FAILED",
           matrixResAdd.toString(), matrixA.add(matrixB).toString());
    }

    @Test
    public void testSub(){

        Matrix matrixA = new Matrix(MATRIX_ONE,5);
        Matrix matrixB = new Matrix(MATRIX_TWO,5);

        Matrix matrixResSub = new Matrix(MATRIX_RESULT_SUB,5);

        assertEquals("Test: sub Matrix FAILED",
           matrixResSub.toString(),matrixA.sub(matrixB).toString());
    }

    @Test
    public void testMult(){

        Matrix matrixA = new Matrix(MATRIX_ONE,5);
        Matrix matrixB = new Matrix(MATRIX_TWO,5);

        Matrix matrixResMult = new Matrix(MATRIX_RESULT_MULT,5);

        assertEquals("Test: mult Matrix FAILED",
           matrixResMult.toString(),matrixA.mult(matrixB).toString());
    }
    @Test
    public void testMod(){
        Matrix matrix = new Matrix(MATRIX_MOD,8);
        Matrix matrixResMod = new Matrix(MATRIX_RESULT_MOD,8);

        assertEquals("Test: mod negative Matrix FAILED",
                matrixResMod.toString(),matrix.toString());
    }

    @Test
    public void testMatrixNotFull(){
        Matrix matrix = new Matrix(MATRIX_NOT_FULL,5);
```

```java
        Matrix matrixRes = new Matrix(MATRIX_NOT_FULL_RES,5);

        assertEquals("Test: matrix not full FAILED",
                matrixRes.toString(),matrix.toString());
    }


    @Test
    public void testInitDimMod(){
        Exception e = assertThrows(RuntimeException.class,
                ()-> new Matrix(1, 3, 0));
        assertEquals("Test: mod equal to zero when init with dim FAILED"
                , "The modulo cannot be negative or equal to 0",
                e.getMessage());
    }

    @Test
    public void testInitMatrixMod(){
        Exception e = assertThrows(RuntimeException.class,
                ()->new Matrix(new int[][] {{1,3}, {1,4,5,6}}, 0));
        assertEquals("Test: mod equal to zero when init" +
                        " with matrix FAILED",
                    "The modulo cannot be negative or equal to 0",
                    e.getMessage());
    }

    @Test
    public void testVerifyMod(){
        Matrix matrixA = new Matrix(MATRIX_ONE,5);
        Matrix matrixB = new Matrix(MATRIX_TWO,8);

        Exception e = assertThrows(RuntimeException.class,
                ()->matrixA.add(matrixB));
        assertEquals("Test: same mod when operating FAILED",
                "The moduli of the 2 matrices are not equal",
                e.getMessage());
    }

    @Test
    public void testInitDimZero(){
        Exception em = assertThrows(RuntimeException.class,
                ()->new Matrix(0, 3, 4));
        assertEquals("Test: dim m not equal to zero FAILED",
                "One of the dimension is lower or equal to 0",
                em.getMessage());
        Exception en = assertThrows(RuntimeException.class,
                ()->new Matrix(4, 0, 4));
        assertEquals("Test: dim n not equal to zero FAILED",
                "One of the dimension is lower or equal to 0",
                en.getMessage());
    }

    @Test
    public void testInitEmptyMatrix(){
        Exception e = assertThrows(RuntimeException.class,
                ()->new Matrix(new int[][]{{}}, 9));
        assertEquals("Test: init with an empty matrix FAILED",
                "The array passed as parameter isn't valid",
                e.getMessage());

    }
}

```

```java
/**
 * This subtracts operand B from operand A.
 * Children of the abstract class Operation : {@link Operation}
 * Operand A and operand B must be integers
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       13/11/2023
 */
public class Subtraction extends Operation {

    /**
     * Subtract the second operand to the first.
     * @param opA first operand
     * @param opB second operand
     * @return the first operand minus the second
     */
    @Override
    public int operate(int opA, int opB) {
        return opA - opB;
    }
}

```

```java
/**
 * This multiplies operand A and operand B.
 * Children of the abstract class Operation : {@link Operation}
 * Operand A and operand B must be integers
 *
 * @author      Arthur Junod
 * @author      Edwin Haeffner
 * Date :       13/11/2023
 */
public class Multiplication extends Operation{

    /**
     * Multiply the second and first operands.
     * @param opA first operand
     * @param opB second operand
     * @return first * second
     */
    @Override
    public int operate(int opA, int opB) {
        return opA * opB;
    }
}

```