Problem 1

Create a class containing static functions:

- `static int maxOfThree(int a, int b, int c)` returning the value of the greatest of three numbers passed to the function;
- `static int greatestDivisor(int n)` returning the greatest divisor of `n` smaller than `n` (this will be 1 for prime numbers);
- `static boolean areRelativelyPrime(int a, int b)` returning **true** if and only if `a` and `b` are relatively prime;
- `static boolean isPerfect(int n)` returning **true** if and only if `n` is a perfect number, i.e., is the sum of all its divisors (including 1 but excluding `n` itself).

In the **main** function test all these functions. Do not use any classes from packages other than basic *java.lang* (in particular, no collections are allowed). You can assume that the functions will be invoked with positive arguments.

For example, the following program

download *StatFuns.java*

```
public class StatFuns {
    static int maxOfThree(int a, int b, int c) {
        // ...
    }
    static int greatestDivisor(int n) {
        // ...
    }
    static boolean areRelativelyPrime(int a, int b) {
        // ...
    }
    static boolean isPerfect(int n) {
        // ...
    }
    public static void main(String[] args) {
        int x = 2, y = 3, z = 1;
        System.out.println("Max of " + x + ", " + y + ", " +
                z + " is " + maxOfThree(x, y, z));

        for (int a = 12; a < 16; ++a)
                System.out.println("Greatest divisor of " +
                    a + " is " + greatestDivisor(a));

        for (int m = 11, n = 5; m >= 3; m -= 2, n += 2)
            if (areRelativelyPrime(m, n))
                System.out.println(m + " and " + n +
```

```
                    " are relatively prime");

        for (int m = 2; m <= 100; ++m)
            if (isPerfect(m))
                System.out.println(m + " is perfect");
    }
}
```

should print

```
Max of 2, 3, 1 is 3
Greatest divisor of 12 is 6
Greatest divisor of 13 is 1
Greatest divisor of 14 is 7
Greatest divisor of 15 is 5
11 and 5 are relatively prime
9 and 7 are relatively prime
7 and 9 are relatively prime
5 and 11 are relatively prime
3 and 13 are relatively prime
6 is perfect
28 is perfect
```

Problem 2

Create a class **OneLiners** with four static functions described below. Try to implement them as 'one-liners', i.e. in such a way that each of them consists only of just one **return** statement. Do not use any functions from the standard library.

- `public static boolean hasTwoRoots(double a, double b, double c)` returns **true** if, and only if, the equation $ax^2 + bx + c$ has exactly two different solutions, and **false** otherwise;
- `public static boolean monot(double a, double b, double c)` returns **true** if, and only if, the numbers `a`, `b` and `c` constitute a strictly increasing sequence (each subsequent term is bigger than the previous) or a strictly decreasing one, and **false** otherwise;
- `public static double maxEl(double a, double b, double c))` returns the maximum value of the numbers `a`, `b` and `c`;
- `public static int numPos(double a, double b, double c)` returns the number of positive values among `a`, `b` and `c`.

For example, the following program,

download *OneLiners.java*

```
public class OneLiners {
    public static boolean hasTwoRoots(double a,
                                      double b, double c) {
        // ...
    }
```

```
public static boolean monot(double a,
                            double b, double c) {
    // ...
}
public static double maxEl(double a,
                           double b, double c) {
    // ...
}
public static int numPos(double a,
                         double b, double c) {
    // ...
}

public static void main(String[] args) {
    double a = 2, b = 3, c = 1;
    System.out.println("(a, b, c,)=(" + a + ", " +
            b + ", " + c + "): hasTwoRoots? " +
            hasTwoRoots(a,b,c));
    a = 0; b = 2; c = 1;
    System.out.println("(a, b, c,)=(" + a + ", " +
            b + ", " + c + "): hasTwoRoots? " +
            hasTwoRoots(a,b,c));

    a = 2; b = 1; c = -1;
    System.out.println("(a, b, c,)=(" + a + ", " +
            b + ", " + c + "): monot? " +
            monot(a,b,c));
    a = 2; b = 1; c = 2;
    System.out.println("(a, b, c,)=(" + a + ", " +
            b + ", " + c + "): monot? " +
            monot(a,b,c));

    a = 2; b = 1; c = 2;
    System.out.println("(a, b, c,)=(" + a + ", " +
            b + ", " + c + "): maxEl = " +
            maxEl(a,b,c));
    a = 2; b = 2; c = 4;
    System.out.println("(a, b, c,)=(" + a + ", " +
            b + ", " + c + "): maxEl = " +
            maxEl(a,b,c));

    a = -2; b = 1; c = -3;
    System.out.println("(a, b, c,)=(" + a + ", " +
            b + ", " + c + "): numPos = " +
            numPos(a,b,c));
```

```
        a = -2; b = 1; c = 3;
        System.out.println("(a, b, c,)=(" + a + ", " +
                b + ", " + c + "): numPos = " +
                numPos(a,b,c));
    }
}
```

after implementing the functions, should print

```
(a, b, c,)=(2.0, 3.0, 1.0): hasTwoRoots? true
(a, b, c,)=(0.0, 2.0, 1.0): hasTwoRoots? false
(a, b, c,)=(2.0, 1.0, -1.0): monot? true
(a, b, c,)=(2.0, 1.0, 2.0): monot? false
(a, b, c,)=(2.0, 1.0, 2.0): maxEl = 2.0
(a, b, c,)=(2.0, 2.0, 4.0): maxEl = 4.0
(a, b, c,)=(-2.0, 1.0, -3.0): numPos = 1
(a, b, c,)=(-2.0, 1.0, 3.0): numPos = 2
```

Problem 3

Write a static function which takes an array of **int**s and swaps (exchanges the values) of its greatest and smallest elements. Create a program which prints the array (in one line) before and after this operation.

Problem 4

Write a static function which takes a two-dimensional array of **int**s, for example

```
int[][] arr = { {1,3}, {3,4,5,8}, {6,8}, {1,9,6} };
```

and then creates and returns an array of size `arr.length` the elements of which are equal to maximum elements of subsequent 'rows' of the array **arr** (of course, the program should work with any definition of this array).

Problem 5

Write a static function **inner** which takes a two-dimensional *rectangular* (by assumption) array of **int**s (we assume that both number of rows and number of columns are not smaller than three). The function creates and returns a new two-dimensional array which contains the "inner part" of the original array, i.e., without the first and the last row and without the first and the last column. It doesn't print anything!

For example, the following program

download *Arr2DInn.java*

```
import java.util.Arrays;
public class Arr2DInn {
    // ...
    public static void main (String[] args) {
        int[][] a = { {1,2,3,4,5,6},
                      {2,3,4,5,6,7},
```

```
                        {3,4,5,6,7,8},
                        {4,5,6,7,8,9} };
            for (int[] r : a)
                System.out.println(Arrays.toString(r));
            System.out.println();
            for (int[] r : inner(a))
                System.out.println(Arrays.toString(r));
        }
    }
```

should print

```
[1, 2, 3, 4, 5, 6]
[2, 3, 4, 5, 6, 7]
[3, 4, 5, 6, 7, 8]
[4, 5, 6, 7, 8, 9]

[3, 4, 5, 6]
[4, 5, 6, 7]
```

Problem 6

Create a static function

```
private static int roll() {
    // ...
}
```

which simulates throws of the two dice and repeats rolls until two sixes come out; the function returns the number of throws.
In **main** function call **roll** 3 million times and answer the questions

- in what percentage of cases two sixes came out in the first roll;
- in what percentage of cases two sixes came out for the first time after 128 or more rolls.

The printout of the program could look something like

```
In first roll    : 2.77%
128 or more rolls: 2,81%
```

[As can be easily calculated, expectation values for these two numbers are approximately 2.78% and 2.79%]

Problem 7

A producer of cereal inserts a coupon into every box; with equal probability, of one out of N kinds. Customers can win a prize after collecting at least one of every kind. We assume that collectors do not cooperate (by exchanging their coupons).

Write a function **boxesBought** which simulates, using a pseudo-random number generator, the process of buying boxes of cereal until at least one coupon of every kind is collected. The function takes the number of kinds of coupons and returns the number of boxes bought. Run many (e.g., 100000) such simulations and calculate the average number of boxes bought (for various values of N).

Compare the average obtained with the theoretically expected value which is, as can be calculated, NH_N, where H_N is the n-th harmonic number

$$H_N = \sum_{i=1}^{N} \frac{1}{i}$$

(write an auxiliary function **harmo** calculating harmonic numbers.)
The program below

download *Coupons.java*

```
public class Coupons {
    public static void main(String[] args) {
        final int N = 90;
        final int NUM_OF_SIMULATIONS = 100000;
        int totalBoxes = 0;
        for (int  i = 0; i < NUM_OF_SIMULATIONS; ++i) {
            totalBoxes += boxesBought(N);
        }
        double aver = totalBoxes/(double)NUM_OF_SIMULATIONS;
        System.out.println("***** N = " + N);
        System.out.println("Average : " + aver);
        System.out.println("Expected: " + N*harmo(N));
    }

    static int boxesBought(int coupons) {
        // ...
    }

    static double harmo(int n) {
        // ...
    }
}
```

should print (of course, the numbers obtained from simulation can be a little different)
For $N = 5$

```
***** N = 5
Average : 11.37195
Expected: 11.416666666666666
```

For $N = 30$

```
***** N = 30
Average : 119.86697
Expected: 119.84961392761174
```

For $N = 90$

```
***** N = 90
Average : 457.93183
Expected: 457.4313542563665
```

The task could be accomplished more effectively by using streams and collections, but plain loops and arrays can also be sufficient.
