Exercises Java

Øyvind Ryan

February 19, 2013

1. What code blocks are executed, and which are not?

```
int smallNumber = 13;
if ((smallNumber > 5) && (smallNumber < 10)){
    System.out.println("Between 5 and 10");
}
if ((smallNumber > 10) && (smallNumber < 15)){
    System.out.println("Between 10 and 15");
}
if ((smallNumber > 15) && (smallNumber <20)){
    System.out.println("Between 15 and 20");
}
```

What would happen if the first line was changed to

```
int smallNumber = 10;
// the rest unchanged
```

Solution: Since 13 is both larger than 10 and smaller than 15, the middle block is executed. For the other two blocks, only one of the parts is true, so the code isn't executed. If we use 10 instead, nothing is executed. This is because $10 < 10$ is false.

2. This code block

```
boolean[] truthValues = {false, true};
for (boolean x : truthValues){
    for (boolean y : truthValues){
    System.out.printf("%b and %b is %b\n", x, y, x && y);
    }
}
```

produces a truth table for $\wedge$. Modify it to make truth tables for eaxh statement.
Solution: In each case, we write the whole code. Note that the first two parts simply have very minor changes, whereas the third part needs an additional nested layer.

a. $\vee$
Solution: $\vee$ can be used directly.

```
boolean[] truthValues = {false, true};
for (boolean x : truthValues){
        for (boolean y : truthValues){
                System.out.printf("%b or %b is %b\n", x, y, x || y);
        }
}
```

b. $\rightarrow$

Solution: Truth table for implication, $\rightarrow$, using $x \rightarrow y = \neg x \vee y$:

```
boolean[] truthValues = {false, true};
for (boolean x : truthValues){
for (boolean y : truthValues){
System.out.printf("%b implies %b is %b\n", x, y, !x || y);
        }
}
```

c. $z \rightarrow (x \vee y)$

Solution: We use that $z \rightarrow (x \vee y) = \neg z \vee x \vee y$

```
boolean[] truthValues = {false, true};
for (boolean x : truthValues){
        for (boolean y : truthValues){
                for (boolean z : truthValues){
System.out.printf("%b implies %b or %b is %b\n",
z, x, y, !z ||x || y);
                }
        }
}
```

3. Compute the values of the smallest and the largest `double` as a decimal number.

Solution: Using the calculator, we find that the largest is

$$(2-2^{-52}) \cdot 2^{1023} = \text{Math ERROR}$$

Oops! The number is too large for my calculator. What now?

Trying Java with the same formula, as in

```
double maxDouble = (2-Math.pow(2, -52)) * Math.pow(2,1023);
System.out.println(maxDouble);
```

returns

```
1.7976931348623157E308
```

So the maximal double is $\approx 1.8 \cdot 10^{308}$.

Note that expanding the parantheses would lead to an error:

```
double maxDouble = 2 * Math.pow(2,1023) -
    Math.pow(2, -52) * Math.pow(2,1023);
System.out.println(maxDouble);
```

yields `Infinity` since $2 \cdot 2^{1023} = 2^{1024}$ is too large for a double.

Similarly, we find the smallest double as

```
double minDouble = Math.pow(2, -1074);
System.out.println(minDouble);
```

returns

```
4.9E-324
```

so the minimal double is $\approx 4.9 \cdot 10^{-324}$.

4. See how Java handles $0/0$, $1/0$ and $-1/0$ for integers (use `int`). Also check how Java handles $\sqrt{-1}$ (use `Math.sqrt`).
Solution: Each of the expressions $0/0$, $1/0$ and $-1/0$ for integers lead to compilation errors. Running

```
double rootOfNegative = Math.sqrt(-1);
System.out.println(rootOfNegative);
```

returns NaN.

5. Convert the `float` $124.234f$ to `long`.
Solution:

```
long converted = (long) 124.234f;
```

6. The associative law for multiplication states that $(ab)c = a(bc)$. Run this code:

```
float a=0.25e-12f;
float b=4e12f;
float c=5e35f;
System.out.println((a*b)*c);
System.out.println(a*(b*c));
```

What happens? Change all the `floats` into `doubles`, and try again. What has changed? Can you create a similar problem with `doubles`?
Solution: The output from running the code is

```
5.0E35
Infinity
```

Changing to doubles, we get $4.99\cdots 10^{35}$ from both computations. The problem is that bc is too large for a `float`, so that infinity is returned when this computation comes first. To get the same problem for `double`, we need to input larger numbers. Trial and error, and remembering the result from Exercise 3, we get the same type of result from running

```
double a=0.25e-12;
double b=4e12;
double c=5e305;
System.out.println((a*b)*c);
System.out.println(a*(b*c));
```

7. In this exercise, we will use Java to solve quadratic equations. The formula for solving the quadratic equation

$$ax^2 + bx + c = 0$$

is

$$x_0 = \frac{-b+\sqrt{b^2-4ac}}{2a} \quad x_1 = \frac{-b-\sqrt{b^2-4ac}}{2a}.$$

Choose values for a, b, c as `floats`, and write a programme that computes these two solutions. Note that the square root is found in Java as `Math.sqrt`.

Then add a test to the programme, checking whether the numbers you have calculated actually solves the original problem. Try out your programme on these four sets of coefficients.

a	b	c
1	2	-3
1	2	1
1	1	1
1	-20000	1

What happens? Why?

If you get a “possible loss of precision" error, explicitly cast to `float`.
Solution: Here is code for the programme, initalized with the first set of coefficients from the table. It would be better to print a more informative statement.

```
float a =1.0f;
float b = 2.0f;
float c = -3.0f;
float x_0 = (float) (-b + Math.sqrt(b*b - 4*a*c))/(2*a);
float x_1 = (float) (-b - Math.sqrt(b*b - 4*a*c))/(2*a);
System.out.println(x_0);
System.out.println(x_1);
//test. This should return 0
System.out.println(a*x_0*x_0 + b*x_0+c);
System.out.println(a*x_1*x_1 + b*x_1+c);
```

This returns

```
1.0
-3.0
0.0
0.0
```

In the next set of values for a, b, c, we get

```
-1.0
-1.0
0.0
0.0
```

The third:

```
NaN
NaN
NaN
NaN
```

Looking at the computation, we see that we try to compute the square root of a negative number, and this returns NaN as in Exercise 4.

The last:

```
20000.0
0.0
1.0
1.0
```

Here the two solutions give 1 instead of 0 when plugged into the original equation. The reason is that there are errors coming from rounding. If all the variables are changed into `doubles` (and we delete the casting to `float`), the output will be

```
19999.999949999998
5.000000055588316E-5
-5.9604644775390625E-8
-8.617663249665952E-9
```

We see that the approximate solution 20000 is actually a bit too large, and 0 a bit to small. Also, the test doesn't return 0, but numbers quite close to 0, which is to be expected because of rounding errors.

8. One can use `floats` in loops, which is practical if we want for instance points on a graph. Try to use a `for` loop of the form

```
for (double x = startValue, x < stopValue, x = x + increment){...}
```

to compute points on the graph of $f(x) = \sin x$ in the interval $[-7, 7]$. To make sure that the point are close enough to get a nice graph from plotting them, compute the sine for each of the points $-7, -6.99$ and so forth.

Then try to do the same computation using an `int` as index in the loop.

Solution: Here is the code for writing the points to the terminal window:

```
for (double x = -7.0; x<7.0; x+= 0.01){
    double y = Math.sin(x);
    System.out.printf("Point (%f, %f)\n ", x, y);
}
```

To find the easiest formulation for the range of an integer variable, note that if we multiply with a hundred, the increment is changed to +1. So we can write

```
for(int i = -700; i< 700; i++){
    double x = ((double) i)/100;
    double y = Math.sin(x);
    System.out.printf("Point (%f, %f)\n ", x, y);
}
```

Note that we also need to compute x, and we need to make sure that $i/100$ is not considered integer division.

9. In this exercise, we will compute as many powers of two that we can, using the primitive numerical types in Java. Start by defining `int power = 2`, and multiply repeatedly by 2. Check (how?) if the computation is correct using `int`, and if it isn't, convert to `long`. When this also fails, convert to `float`. When this fails, convert to `double`. Finally, write a statement about the largest powers handled by the different types.

Can you modify the programme to work with powers of 3?

Solution: There are many ways to solve this exercise, so if your solution doesn't resemble this one, don't worry.

```java
int power = 1;
int powerOf2 = 2;
int oneLower = 1;
while (powerOf2 > oneLower){
    System.out.println(powerOf2);
    powerOf2 *= 2;
    oneLower *= 2;
    power ++;
}
System.out.printf("Longs from power %d\n", power);
long powerOf2L = 2*(long) oneLower;
long oneLowerL = (long) oneLower;
while (powerOf2L > oneLowerL){
    System.out.println(powerOf2L);
    powerOf2L *= 2;
    oneLowerL *= 2;
    power ++;
}
System.out.printf("Floats from power %d\n", power);
float powerOf2f = 2* (float) oneLowerL;
float oneLowerf = (float) oneLowerL;
Float infinity = new Float(Float.POSITIVE_INFINITY);
while (powerOf2f < infinity){
    System.out.println(powerOf2f);
    powerOf2f *= 2;
    oneLowerf *= 2;
    power ++;
}
System.out.printf("Floats from power %d\n", power);
double powerOf2Double = 2* (double) oneLowerf;
double infinityDouble = (double) infinity;
```

```
while (powerOf2Double < infinityDouble){
    System.out.println(powerOf2Double);
    powerOf2Double *= 2;
    power ++;
}
System.out.printf("Final power is %d\n", power);
```

Changing all instances of 2 into 3s actually works, as a test it stops at 3^{647}.

10. Create multiplication and addition tables for $\pm\infty$, NaN, -0 and $+0$. To get the values, use for instance the wrapper class `Float`, as in

```
Float minusInfinity = new Float(Float.NEGATIVE_INFINITY);
```

Solution: This code produces the tables. It is not the simplest construction for the problem, and it might be better to format the output differently.

```
Float minusInfinity = new Float(Float.NEGATIVE_INFINITY);
Float plusInfinity = new Float(Float.POSITIVE_INFINITY);
Float nan = new Float(Float.NaN);
Float zero = new Float(0.0f);
Float minusZero = new Float(-0.0f);

Float[] specialFloats = new Float[5];
specialFloats[0] = minusInfinity;
specialFloats[1] = plusInfinity;
specialFloats[2] = nan;
specialFloats[3] = zero;
specialFloats[4] = minusZero;

//multiplication table:
for (Float special : specialFloats){
    for (Float special2 : specialFloats){
        System.out.printf("%f * %f = %f\n", special, special2,
           special * special2);
    }
}
//addition table:
for (Float special : specialFloats){
    for (Float special2 : specialFloats){
        System.out.printf("%f + %f = %f\n", special, special2,
            special + special2);
    }
}
```

11. Add another method to the `Combinatorics` class called `computeCatalan`. This method takes one input and gives one output. The mathematical formula for the nth *Catalan number* is

$$C_n = \frac{1}{n+1}\binom{2n}{n} \text{ for } n \geq 0.$$

The input should be n, the output C_n.

To check your programming, produce the first few Catalan numbers, and compare with `"http://oeis.org/A000108"` from the online encyclopedia of integer sequences.

Solution: We can use the previously defined method `computeBinomialCoeff`. It is a fact that the Catalan numbers are integers, so we will use the `int` type.

```
public static int computeCatalan(int n){
    return computeBinomialCoeff(2*n,n)/(n+1);
}
```

The first few values are 1, 1 ,2, 5, 14, 42.

12. We will now implement the functions $f : A \to B$, where

$$A = \{0, 1, 2, \cdots, n-1\} \text{ and } B = \{0, 1, 2, \cdots, m-1\}$$

and the function f is defined by

$$f(a) = k \cdot a \mod m.$$

See Example **??**. If you want a more challenging exercise, look only at the Example, and produce a Java programme for these tasks. Otherwise, read the following hints.

To initialize A, first give a value to n, define A as an array of this length, and define $A[i] = i$ in a loop. Do the same for B.

Then declare the function f as an array of the same length as A, and define $f[i]$ as in the pseudocode in the example.

The most involved part of this exercise is to compute the number `numberOfValues` correctly. Do this!

Finally, write a statement if f is injective or surjective.

If you want to check your programme, use it for some of the values in the main text.

Solution: Here is a possible programme. Note that the `break` statement prevents repetitions to be counted; each value is counted only once.

```
public class FunctionFromRemainders{
    public static void main(String[] args){
        int n = 7;
        int m = 8;
        int k = 3;
        int[] setA = new int[n];
        int[] setB = new int[m];
        int[] functionf = new int[n];
        int numberOfValues = 0;

        for (int i = 0; i < n; i++){
            setA[i] = i;
        }
        for (int i = 0; i < m; i++){
            setB[i] = i;
        }
        for (int i = 0; i < n; i++){
            functionf[i] = (i*k) % m;
        }

        for (int b : setB){
            for (int i = 0; i < n; i++){
                if (b == functionf[i]){
                    numberOfValues ++;
                    break;
                }
            }
        }
        if (numberOfValues == n){
            System.out.println("Injective!");
        }
        if (numberOfValues == m){
            System.out.println("Surjective!");
        }
    }
}
```

13. Check how Java deals with division involving one negative and one positive number.
Solution: We run the two possibilities we have (using ±7/3), first with negative numerator:

```
System.out.printf("-7/3 = %d\n", (-7)/3);//quotient
System.out.printf("-7 %% 3 = %d\n",(- 7) % 3); //remainder
```

returns

```
-7/3 = -2
-7 % 3 = -1
```

Then with negative denominator:

```
System.out.printf("7/-3 = %d\n", 7/(-3));//quotient
System.out.printf("7 %% -3 = %d\n", 7 % (-3)); //remainder
```

returns

```
7/-3 = -2
7 % -3 = 1
```

In both cases, the result of dividing yields the quotient rounded towards 0, and the remainder is given such that

$$numerator = quotient \cdot denominator + remainder$$

14. Check that the symbol ^ works as "exclusive or", by displaying `43 ^ 3245` in binary. The number you'll get is 3206.
Solution: Running

```
String binaryXor = String.format("%32s",
   Integer.toBinaryString(43 ^ 3245)).replace(' ', '0');
System.out.println(binaryXor);
```

yields

```
00000000000000000000110010000110
```

We can check the answer; running

```
String binary3206 = String.format("%32s",
   Integer.toBinaryString(3206)).replace(' ', '0');
System.out.println(binary3206);
```

yields the same result.

15. It is not allowed to use modulo 0, %0, in Java (because of division by zero). Check how Java treats modulo 1.

Solution: We can run, for instance

```
for (int i = -5; i<5; i++){
    System.out.println(i % 1);
}
```

Apparently, `%1` always returns 0.

16. Bitwise operations can be used to set flags. This means that the states of some system are described using several switches, that could be turned on or off. We will model this using bitwise operations using the integer type `byte`, which has 8 bits. To get the binary representation as a binary `String`, use

```
byte aByte = 57;
int theInt = 0;
if (aByte < 0){
    theInt = 256 + aByte;
} else{
    theInt = aByte;
}
String binaryByte = String.format("%8s",
    Integer.toBinaryString(theInt)).replace(' ', '0');
```

The +256 is a trick to alleviate the problems from casting to `int`. The bit in the 16-place is set (the switch is turned on) if its value is 1.

Solution:

a. Explain why the bit in the 4-place is set by this code (`stateByte` represents the current value for the states). Print the binary strings for different stateBytes so that you see both what happens if this bit was set and if it wasn't set before running the code.

```
// stateByte is already defined
stateByte = (byte) (stateByte |  4);
```

The bitwise operations are performed as `int`, so we need to cast to `byte` in the end.

Solution: In other places than the four-bit, 4 has 0. Now $0 \vee 0 = 0$ and $1 \vee 0 = 1$, so in either case, nothing is changed. In the four-bit, 4 has 1. Since $1 \vee 1 = 1$ and $0 \vee 1 = 1$, in either case the four-bit is set.

b. Explain why the bit in the 4-place is turned off by this code. Again, give examples both with the 4-bit set and not set before running this code block.

```
// stateByte is already defined
stateByte = (byte) (stateByte &  -5);
```

Solution: -5 is the number that has a 0 in the four-bit, and 1 everywhere else. In places other than the four-bit, using $0 \wedge 1 = 0$ and $1 \wedge 1 = 1$, we see that nothing is changed. In the four-bit, $0 \wedge 0 = 0$ and $1 \wedge 0 = 0$, so in either case the flag is turned off.

c. What does exclusive or (`^`) with 0 do? What does exclusive or with 4 do? What does exclusive or with -1 do?
Solution: 0 has 0 in all bits. $1 \oplus 0 = 1$ and $0 \oplus 0 = 0$, so nothing happens.

4 has 0 in all places different from the four-bit, so nothing happens there. In the four-bit, $0 \oplus 1 = 1$ and $1 \oplus 1 = 0$, so the value in this place is switched from on to off or from off to on.

-1 has 1 in all bits, so as in the previous step, we see that all bits are changed. Thus `a ^ -1 = ~a`.

d. Given two `integers` `intX` and `intY`, what is the value of each of them after running this code:

```
intX = intX ^ intY;
intY = intX ^ intY;
intX = intX ^ intY;
```

If it is difficult to see what happens, run a couple of examples.
Solution: After the first step, `intX` is changed in all bits where `intY` has a 1. After the second step, `intY` gets the value computed from `intX` by changing all the bits were `intY` had a one *again*. Thus `intY` now has the original value of `intX`. Another way to think about `intX` after the first step, is to say that it comes from `intY` by changing all the bits were `intX` has a one. Since `intY` after step two is the original `intX`, after the third step, the changes in `intY` to `intX` in the first step are undone, and `intX` gets the original value of `intY`.

To conclude: The two variables change their values!

17. Find the sum of the integers underlying the `char`s in the English alphabet.
Solution: We need to cast each of the characters `'a', 'b', ..., 'z'` to `int` and sum the results. This code performs these operations, keeping a running total.

```
int sumOfChars = 0;
for (char c = 'a'; c <= 'z'; c++){
    sumOfChars += (int) c;
}
```

The answer is 2847. Note that if we drop the explicit cast `(int)`, Java casts implicitly for us.

18. Find the smallest n such that

$$\sum_{i=1}^{n} i^2$$

is too large to be represented as an `int`.
Solution: An easy solution is to compute using `long`s, and compare with the largest `int`, also represented as a `long`-. Here is code using this approach.

```
long maxInt = Integer.MAX_VALUE;
long runningTotal = 0;
int i = 0; //summation index
while (runningTotal < maxInt){
    i ++;
    runningTotal += i*i;
}
System.out.println(i);
```

The answer turns out to be 1861.

19. In this exercise, we define a couple of sets of integers, and let Java compute the set-theoretical operations. We first fix our universe, which will consist of the 11 elements

$$universe = \{0,1,2,3,4,5,6,7,8,9,10\}$$

A and B willl be some more or less random subsets. The initialization of what we've discussed so far can be done as follows:

```
int[] universe = {0,1,2,3,4,5,6,7,8,9,10};
int[] setA = {0,1,2,3,4,5};
int[] setB = {2,3,5,7};
```

To check whether an element, say 3, is in the union of A and B, we can use the code block

```
boolean isInUnion = false;
for (int numberA : setA) {
    if (3 == numberA) {
        isInUnion = true;
    }
}
for (int  numberB : setB) {
    if (3 == numberB) {
        isInUnion = true;
    }
}
if (isInUnion) {
    System.out.printf("%d is in the union\n", 3);
}
```

To check for all elements, we loop through the entire universe:

```
for (int number : universe) {
    boolean isInUion = false;
    for (int numberA : setA) {
        if (number == numberA) {
            isInUnion = true;
        }
    }
    for (int  numberB : setB) {
        if (number == numberB) {
            isInUnion = true;
        }
    }
    if (isInUnion) {
        System.out.printf("%d is in the union\n", number);
    }
}
```

Modify this programme to find the intersection. Also find the complement of A.

For the more ambitious, modify the code so that fewer equalities are checked. Use for instance **break**.

Solution: We answer this exercise by giving the entire programme. Note that the structure for finding the boolean test variables is the same as in the text, it is only the way they are used that is different.

```
//Initialization
int[] universe = {0,1,2,3,4,5,6,7,8,9,10};
int[] setA = {0,1,2,3,4,5};
int[] setB = {2,3,5,7};

//Intersection: we need both in A and in B
for (int number : universe){
boolean testA = false;
    boolean testB = false;
    for ( int numberA : setA){
        if (number == numberA){
            testA = true;
        }
    }
    for ( int  numberB : setB){
        if (number == numberB){
            testB = true;
        }
    }
    if (testA && testB){
        System.out.printf("%d is in the union\n", number);
    }
}

//Complement of A:
for (int number : universe){
    boolean testA = false;
    for ( int numberA : setA){
        if (number == numberA){
            testA = true;
        }
    }
    if (!testA){
        System.out.printf("%d is in the complement of A\n", number);
    }
}
```

20. In this exercise, we will use Java to check whether a relation is a partial order. First, we define a set and a relation on it. This time, we use **char**.

```
char[] setA = {'a','b','c','d','e','f'};
char[][] relation ={{'a','a'},{'b','b'},{'c','c'},{'d','d'},
    {'e','e'},{'f','f'},{'a','d'},{'a','f'},{'c','d'},
    {'c','e'},{'c','f'},{'d','f'}};
```

We need to check reflexivity, anti-symmetry and transitivity.

To check for reflexivity, we create a pair (c,c) for each element c of A, and checks if it is in the relation. We then count how many times this happens, and if it happens as many times as there are elements in A, the relation is reflexive.

```
int reflexivePairs = 0;
for (char c : setA) {
    char[] cPair = new char[2];
    cPair[0] = c;
    cPair[1] = c;
    for (char[] pair : relation) {
        if (pair[0] == cPair[0] && pair[1] == cPair[1]) {
            reflexivePairs++;
        }
    }
}
if(reflexivePairs == setA.length) {
    System.out.println("The relation is reflexive");
}
```

Note that we checked the equality of pairs by checking both entries.

To check for anti-symmetry, we search for pairs (x,y) and (y,x). If both are in the relation, we check whether $x = y$. Unless we find a counter-example, this condition holds. Therefore we intialize

```
boolean isAntiSymmetric = true;
```

and change the value if we find a counter-example.

```
boolean isAntiSymmetric = true;
for (char[] pair : relation) {
    for (char[] newPair : relation) {
        if (newPair[0] == pair[1] && newPair[1] == pair[0]
                && pair[0] != pair[1]) {
```

```
            isAntiSymmetric = false;
        }
    }
}
if (isAntiSymmetric) {
    System.out.println("The relation is antisymmetric");
}
```

The main task left to you is to check for transitivity. As for antisymmetry, the condition holds unless we find a counter-example. We need to check, for all two pairs (x, y) and (y, z), whether (x, z) is also in the relation.

Finally, write a statement if all three conditions are fulfilled.

Solution: We answer this exercise by giving the entire programme, with a note to what is added to the text of the exercise.

```
//Initialization from the text
 char[] setA = {'a','b','c','d','e','f'};
char[][] relation ={{'a','a'},{'b','b'},{'c','c'},{'d','d'},
    {'e','e'},{'f','f'},{'a','d'},{'a','f'},{'c','d'},
    {'c','e'},{'c','f'},{'d','f'}};
//reflexivity from the text
int reflexivePairs = 0;
for (char c : setA){
    char[] cPair = new char[2];
    cPair[0] = c;
    cPair[1] = c;
    for (char[] pair : relation){
        if (pair[0] == cPair[0] && pair[1]==cPair[1]){
            reflexivePairs++;
        }
    }
}
if(reflexivePairs == setA.length){
    System.out.println("The relation is reflexive");
}
//reflexivity ends

//antiSymmetry from the text
boolean isAntiSymmetric = true;
for (char[] pair : relation){
    for (char[] newPair : relation){
```

```java
        if (newPair[0] == pair[1] && newPair[1] == pair[0] &&
                pair[0]!=pair[1]){
            isAntiSymmetric = false;
        }
    }
}
if (isAntiSymmetric){
System.out.println("The relation is antisymmetric");
}
//antiSymmetry ends

//transitivity - new stuff
boolean isTransitive = true;
for (char[] pair1 : relation ){
    for (char[] pair2 : relation){
        if (pair1[1]==pair2[0]){
            //check if pair1[0],pair2[1] is in the relation
            char[] testPair = new char[2];
            testPair[0] = pair1[0];
            testPair[1] = pair2[1];
            boolean test = false;
            for (char[] pair : relation){
                if(testPair[0] == pair[0] && testPair[1]==pair[1]){
                    test = true;
                }
            }
            if (!test){
                isTransitive = false;
            }
        }
    }
}
if (isTransitive){
    System.out.println("The relation is transitive");
}
//transitivity ends

//if partial order:
if(isAntiSymmetric && isTransitive && reflexivePairs == setA.length){
    System.out.println("All three conditions are fulfilled.");
    System.out.println("The relation is a partial order.");
}
```

21. We will count the number of true and false statements in a Java generated truth table, as in Exercise 2. The expressions we will be considering have three variables x, y, z. In the beginning of this exercise, consider the expression $x \wedge (y \rightarrow z)$.
Solution: By combining the code snippets in each part, you will get a complete programme for this problem.

a. Declare an array of booleans, that is of type `boolean[]`, to hold the values of the truth table. How long must the array be? How long would it have to be if there were 12 variables instead of just 3?
Solution: The length must be $2^3 = 8$. If there were 12 variables, we would need $2^{12} = 4096$. This is because we have two choices for each variable.

```
boolean[] truthTable =  new boolean[8];
int index = 0; //keep track of index in truthTable
```

b. Find a loop structure to generate the table as in Exercise 2. Inside the loop, fill the table with values instead of printing the values.
Solution:

```
boolean[] truthValues = {false, true};
for (boolean x : truthValues){
    for (boolean y : truthValues){
        for (boolean z : truthValues){
            truthTable[index] = x && (!y || z);
            index++;
        }
    }
}
```

c. Iterate through the table to find the number of times `true` appears. Write a statement about the number of true and false values.
Solution:

```
int numberOfTrue = 0;
for (boolean tableValue : truthTable){
    if (tableValue){
        numberOfTrue ++;
    }
}
System.out.println("Number of true");
System.out.println(numberOfTrue);
System.out.println("Number of false");
System.out.println(8-numberOfTrue);
```

d. Write an additional statement if the expression is a contradiction or a tautology.
Solution:

```
if (numberOfTrue == 0){
    System.out.println("The expression is a contradiction");
}
if (numberOfTrue == 8){
    System.out.println("The expression is a tautology");
}
```

e. Modify the programme to test the expressions $(x \wedge z) \wedge (y \wedge \neg z)$ and $(x \vee z) \vee (y \vee \neg z)$. Does it work as planned?
Solution: To check for $(x \wedge z) \wedge (y \wedge \neg z)$, we only need to change the innermost line in the loop to

```
truthTable[index] =(x && z) && (y && !z);
```

Similarly, for $(x \vee z) \vee (y \vee \neg z)$, we need

```
truthTable[index] =(x || z) || (y || !z);
```

In the first case, we get a contradiction, in the second a tautology.