# 6. Fundamentals of programming

*From a problem to the solution*

First considerations when for a problem a programme shall be designed:

WHAT – HOW – WITH WHAT

WHAT (which goal) shall
HOW (with what means) and
WITH WHAT (with which instruments) be achieved?

WHAT: problem specification

functional specification:

- input / output and their interrelation,
- formal-mathematical and informal description

specification of requirements:

- ways of usage
- usage rights
- duration of use
- security requirements
- financial context

etc.

HOW:

- algorithm
- structure of programme

WITH WHAT:

- hardware (computer, periphery, other technical equipment)
- software (operating system, programming language, development toolkit, programme libraries, ...)

14 steps from the problem to a solution:

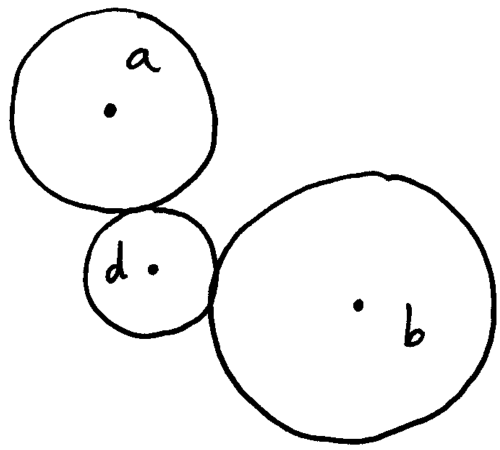
1. Problem
2. Mathematical formulation
3. Finding an algorithm *A* which solves the problem
4. Formulating *A*
5. Precise formulation of *A*
6. Proof of adequatness of *A*
7. Coarse programming
8. Choice of programming language and computer
9. Fine programming
10. Programme verification
11. Test of the programme
12. Calculation
13. (possibly) generalisations
14. (possibly) implementation as subprogramme or as part of a programme library

A simple example:

1. Problem:

In a gear mechanism, the driving cogwheel shall move two other cogwheels, one with *a* cogs (e.g., *a* = 105) and one with *b* cogs (e.g., *b* = 147). A full turn of each of the driven wheels shall correspond to one or several full turns of the driving wheel, and the gear ratios should be as small as possible. How many cogs must the driving wheel have?

2. Mathematical formulation



Let *d* be the number which we are looking for. Then, obviously, the following must be fulfilled:

*d* divides *a* and *d* divides *b*. Because the gear ratios shall be as small as possible, *d* must be maximum, thus *d* = <u>greatest common divisor</u> of *a* and *b* ( gcd(*a*, *b*) ).

3. Finding an algorithm which solves the problem:

Euclid of Alexandria (around 325 B.C., "Elements", book 7, problem 1, proposition 2); probably already found by Eudoxos of Knidos around 375 B.C.:

*"Euclid's Algorithm"*

147 : 105 → 1 rest 42

105 : 42 → 2 rest 21

42 : **21** → 2 rest **0**

gcd(105; 147) = **21**

## 4. Formulating the algorithm:

I. If $a \neq b$, let $x$ be the greater and $y$ the smaller one of the numbers $a$, $b$;
in the case $a = b$ let $x = a = y$.
II. Calculate $x/y$ with rest.
III. If there is no rest, we have $y = \gcd(a, b)$.
IV. If there is a rest $r$, replace $x$ by $y$ and $y$ by $r$ and go to II.

## 5. Precise formulation of this algorithm:

Let $a$, $b$ be positive integers and without loss of generality $b \leq a$,
furthermore $x_0 = a, x_1 = b$ and $x_{n-1} = q_n x_n + x_{n+1}$
with $q_n, x_{n+1} \in \{0; 1; 2; ...\}$
and $x_{n+1} < |x_n|$ for $n > 0$ and $x_n \neq 0$.
Let $n$ be the first index with $x_{n+1} = 0$. Then $x_n = \gcd(a, b)$.

## 6. Proof of adequatness of the algorithm

First to show: The algorithm specified above terminates, i.e. there is always an integer $n$ such that $x_{n+1} = 0$.
This holds because of $x_{n+1} < x_n$ and $x_{n+1} \geq 0$: There exist only finitely many integers between $x_n$ and 0, hence only finitely many steps.

Furthermore, we have
$x_n$ divides $q_n x_n + 0 = x_{n-1}$
$x_n$ divides $q_{n-1} x_{n-1} + x_n = x_{n-2}$
$x_n$ divides $q_{n-2} x_{n-2} + x_{n-1} = x_{n-3}$
...
using the "complete induction principle", we can conclude:
$x_n$ divides $q_2 x_2 + x_3 \quad = x_1 = b$
$x_n$ divides $q_1 x_1 + x_2 \quad = x_0 = a$

hence $x_n$ divides the integers $a$ and $b$ and thus also gcd($a$, $b$).

On the other hand, the gcd is a divisor of $x_0$ and $x_1$ and according to the above chain of equations also of $x_2, x_3, ..., x_{n-1}, x_n$.

Hence $x_n$ = gcd($a$, $b$), and this was to be proved ("q.e.d." = "quod erat demonstrandum").

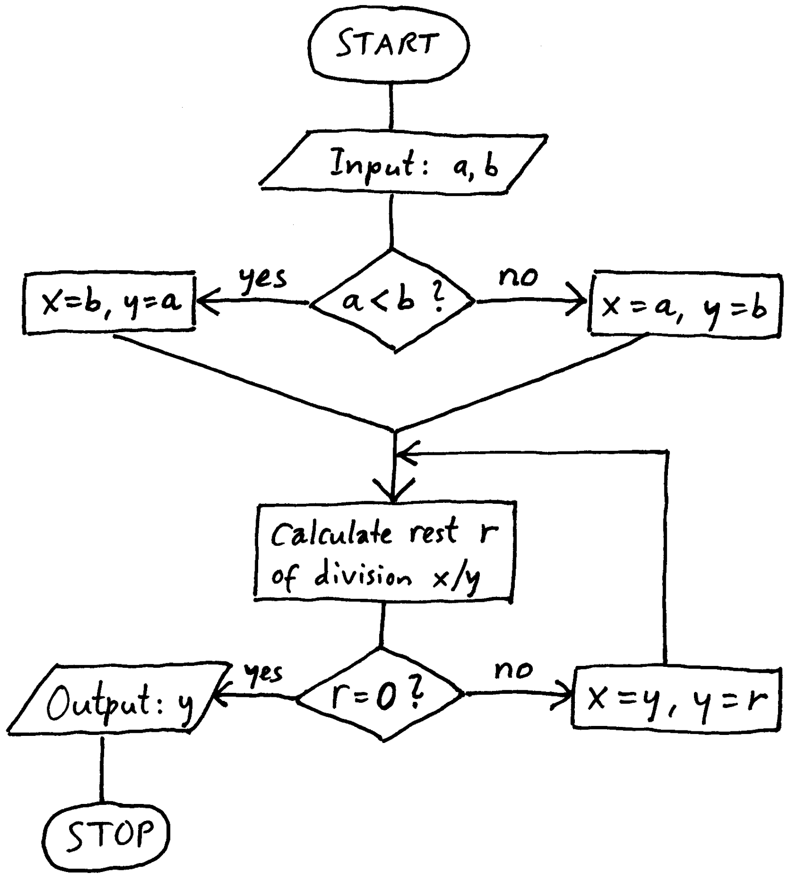
## 7. Coarse programming:

Several methods and principles possible. Examples:

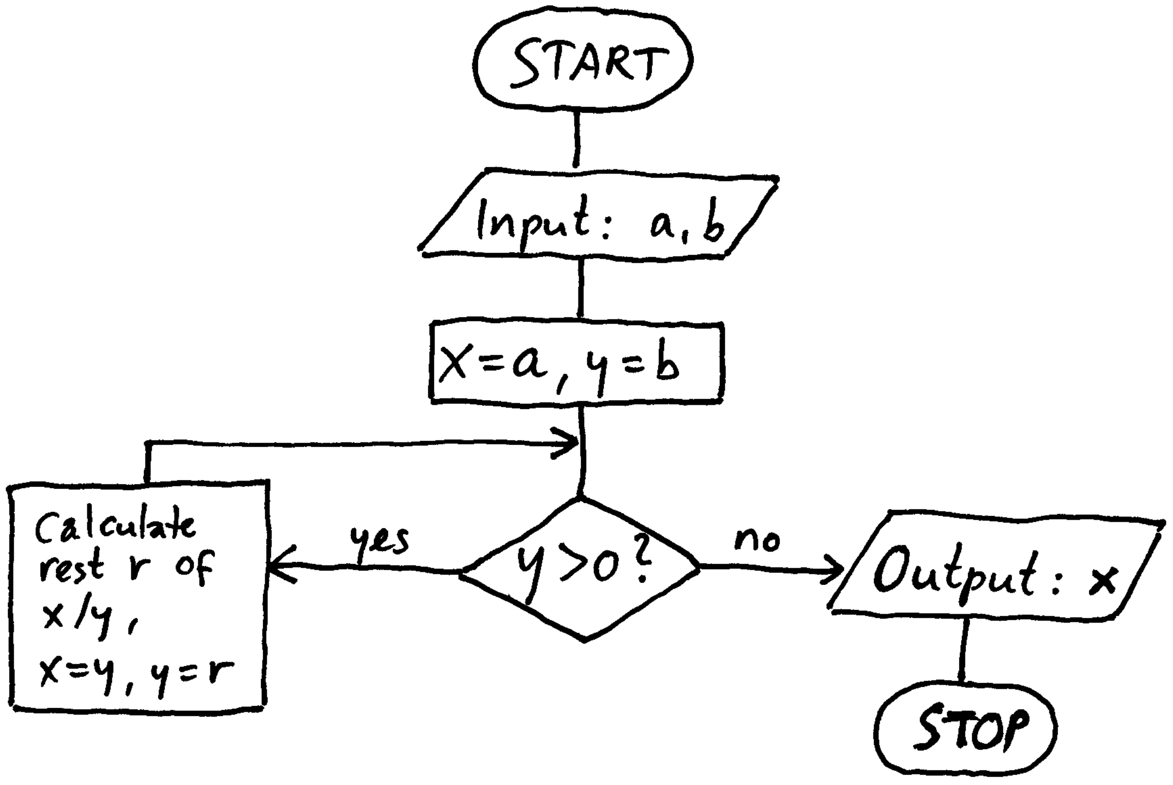
(a) Flowchart

Flow of programme written top-down and from left to right, if not specified otherwise by arrows.

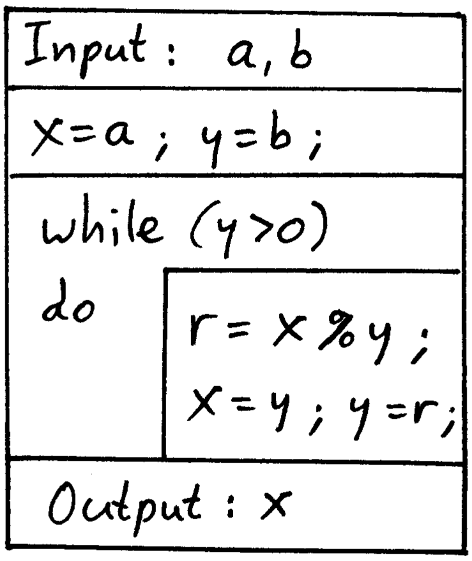
First try:



However, Euclid's Algorithm works also in the case $b \leq a$, thus a simplification is possible, yielding:



(b) Structogramme (Nassi-Shneidermann Diagramme), corresponding to the second flowchart:



## 8. Choice of programming language and computer:

C on Intel-PC under MS-Windows

## 9. Fine programming:

```
#include <stdio.h>
int main()
   {
   char inbuf[50];
   int x, y, r;
   printf("\nTwo input numbers, please: ");
   gets(inbuf);
   sscanf(inbuf, "%d %d", &x, &y);
   while (y>0)
      {
      r = x % y;
      x = y;
      y = r;
      }
   printf("\nResult: %d\n", x);
   return 0;
   }
```

## 10. Programme verification:

Checking the programme for correctness, i.e., if it does in every situation the right thing
- we must follow all manipulations of variables
Means: Assertions which must be fulfilled

*Verification assertions*
- after line 8: $x, y$ positive integers (if correctly read!)
- after line 11: $x = qy+r$, $q\geq 0$, $r \in \{0; 1; ...; y-1\}$
- after line 12: $x = y$

- after line 13: $y = r$
- after line 14: $y = 0$

one has to check that these assertions remain fulfilled in each possible programme execution

## 11. Test of the programme

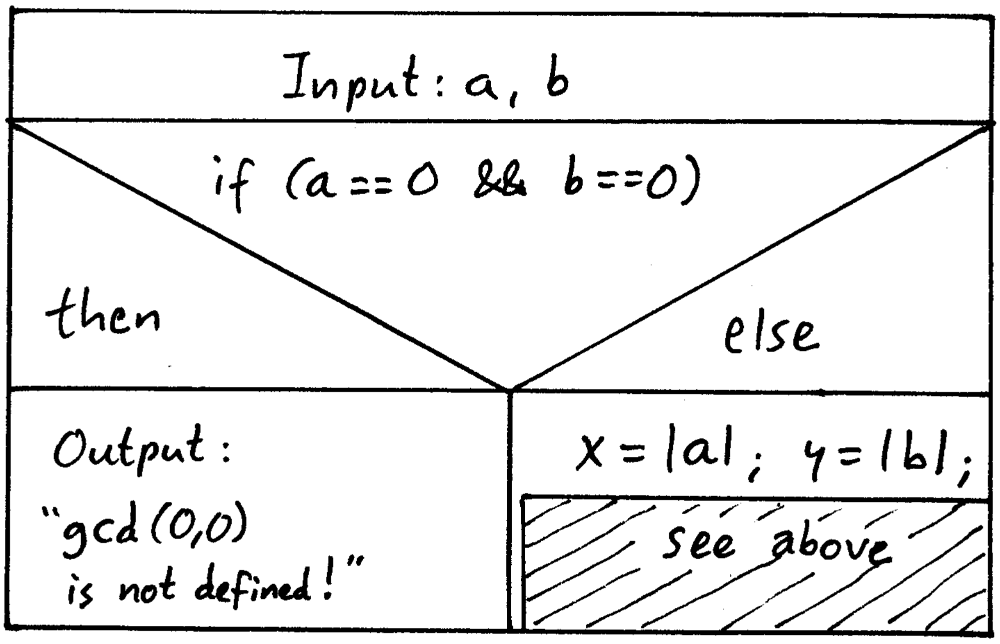
Tests using examples of number pairs with known result; try to do this in a way that all possible, qualitatively different situations are covered.
Particularly check all extreme cases (for example, something is = 0).

Examples:
30 40
100 100
0 100
100 0
100 1

## 12. Calculation

Input 105 147 $\rightarrow$ Output 21,
the driving wheel must have 21 cogs.

## 13. Possible generalisations, e.g., the gcd of arbitrary integers (up to now, we have only considered positive integers):



14. Possible implementation as a subprogramme (function in C):

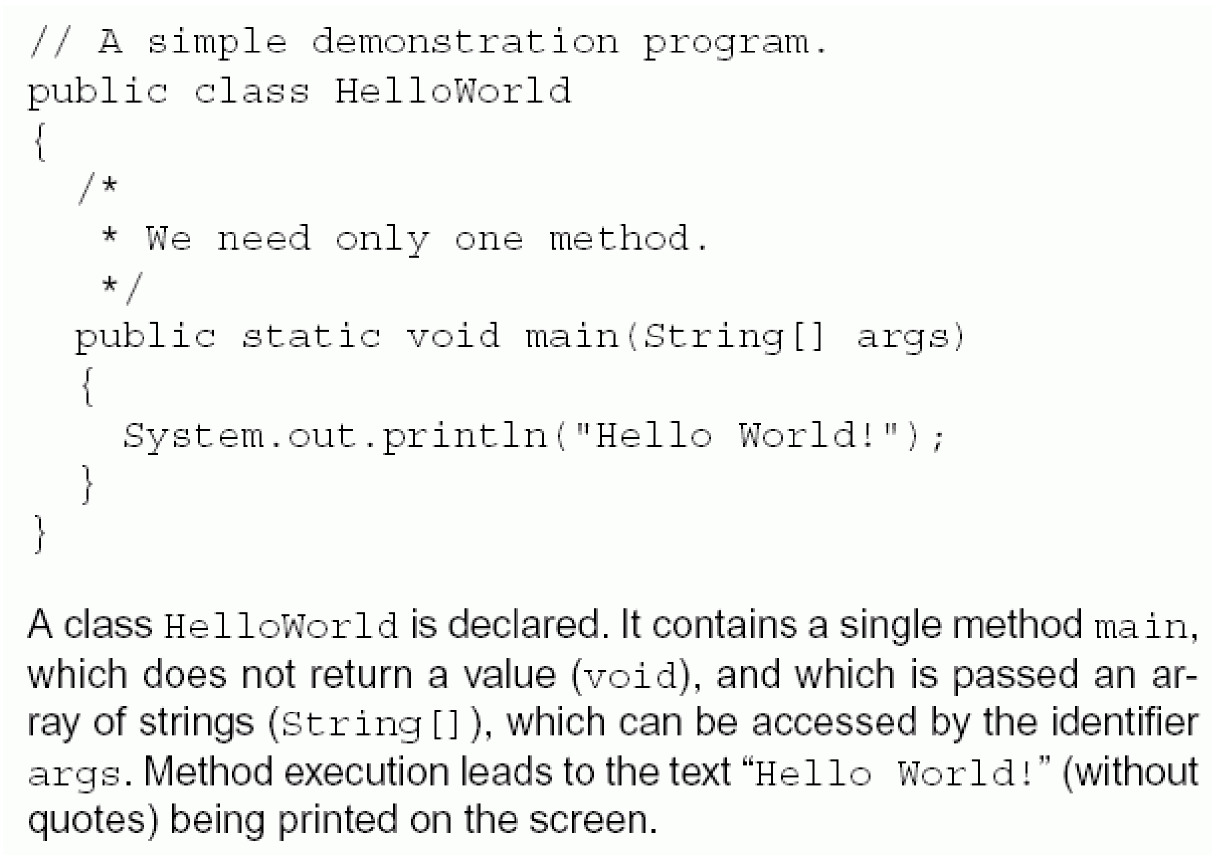
```
int gcd(int x, int y)
   {
   int r;
   while (y > 0)
      {
      r = x % y;
      x = y;
      y = r;
      }
   return x;
   }
```

this can be used in a larger programme – e.g., to calculate iteratively the gcd of more than 2 integers.

Possible call: **`z = gcd(p, q);`**

....

*Simple Java programmes*



```
// A simple demonstration program.
public class HelloWorld
{
  /*
   * We need only one method.
   */
  public static void main(String[] args)
  {
    System.out.println("Hello World!");
  }
}
```

A class `HelloWorld` is declared. It contains a single method `main`, which does not return a value (`void`), and which is passed an array of strings (`String[]`), which can be accessed by the identifier `args`. Method execution leads to the text “`Hello World!`” (without quotes) being printed on the screen.

**`/* ... */`** : comment (skipped by the compiler).
Alternative syntax for comments:
**`// ....`** (the whole line is skipped)

Terminology:

A *class* is a collection of *data*, together with appropriate *operations* on them.
In our example, **`HelloWorld`**, there is no data and only one operation, **`main`**.

A *method* is the definition of an operation, i.e., of something which can be done. Can be invoked by some (other) programme, or by the user.
In our example, the operation **`main`** prints a string on the screen.

*Readability of programmes by humans*

programmes: have to be executed by computers, but also *to be understood by humans*

Executability can be checked automatically, understandability not!

⇒ Recommendations:

- make frequent use of programme comments (`/* ... */` or `// ...` in Java)
- use plenty of newlines and blanks
- put braces { ... } in lines of their own, put matching braces in same horizontal position:
  ```
  {
   ....
  }
  ```
- *indentation* makes containment and nesting of programme components visible
- avoid long lines, insert line breaks for readability
- avoid very long methods
- use "speaking" variable and function names (**`int iteration_counter`** is better than **`int x127`** !)
- do not use variable names twice for different purposes, even if the language allows it
- Initialise constants, default values etc. at the beginning of a source code file, not somewhere "deep in the code" where you don't find them later on
- *adhere to conventions used by competent programmers!*

## *Essentials of Java*

**Basic components**

Comments, spaces, newline: For human readability, and for separating words (just like in normal written language).

**Special symbols**: To denote different kinds of groupings, to terminate commands, to construct paths etc.

Examples: Braces `{`, `}`; parentheses `(`, `)`; brackets `[`, `]`; dot; double-quotes `"`; semicolon

**Literal values**: character sequences representing a value directly, like a digit sequence for a number, or a character sequence in double quotes for a string.

Example: `"Hello World!"`

Sequences of letters or digits, starting with a letter: different categories: **1) Keywords**, **2) predefined identifiers**, **3) newly declared identifiers**.

**1) Keywords**: Are fixed in the language proper, can not be given a new meaning

Examples: `public`, `class`, `static`, `void`

**2) Predeclared identifiers**: Meaning fixed by a declaration in the context, often can be "overwritten", i.e. given a new meaning. Examples:

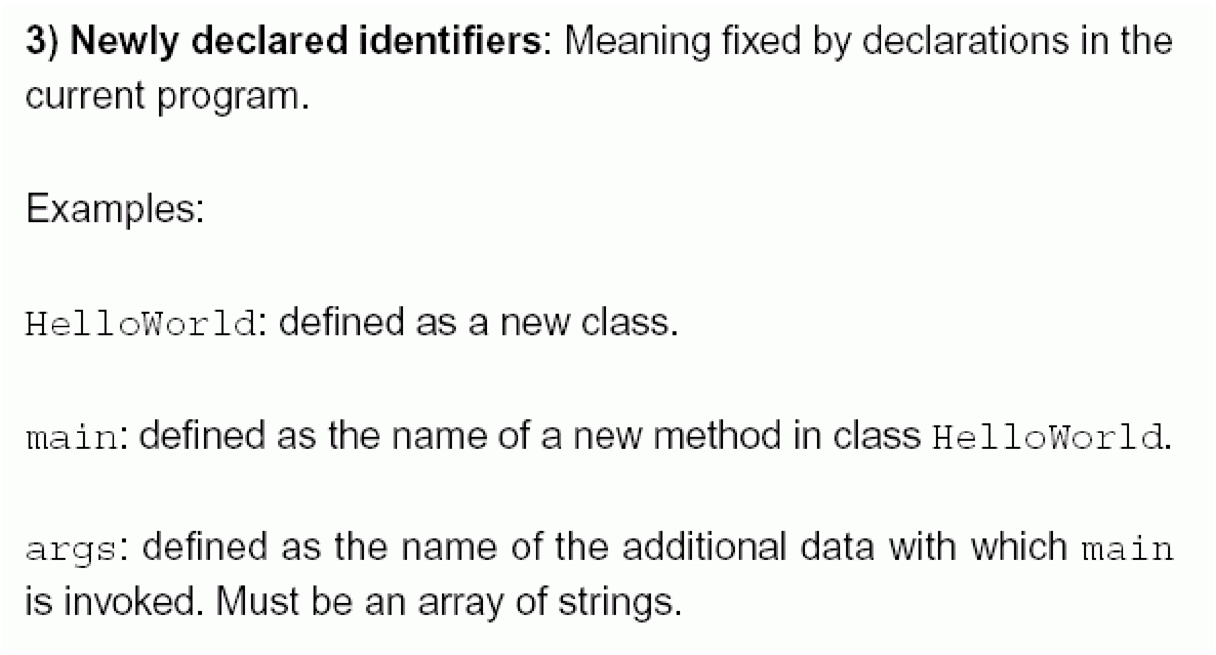
`String`: data type for character sequences

`System`: contains different objects for access to the environment

`out`: predefined in `System`; data stream to the computer screen

`println`: predefined method in `System.out`; invoked with a string it puts the characters of the string to `System.out`, then outputs an additional line break

Dot notation: used for access to components of an object.



**3) Newly declared identifiers**: Meaning fixed by declarations in the current program.

Examples:

`HelloWorld`: defined as a new class.

`main`: defined as the name of a new method in class `HelloWorld`.

`args`: defined as the name of the additional data with which `main` is invoked. Must be an array of strings.

How to compile and run our example programme:

we use the JDK (*Java Development Kit*)
– containing command-line oriented tools for compiling and interpreting Java programmes. (The user gives commands to the computer as lines of text, the computer answers with lines of text.)

Needed for invoking the JDK: A *shell*, i.e., a programme which accepts lines of text from the user and outputs lines of text from programmes on the screen. Example: the "MS-DOS prompt" shell in MS-Windows.

The source code file name of the programme must be the *same as the class name*, with the file name extension **`.java`** appended.

⇒ our example programme must be placed in a file called **`HelloWorld.java`** .

**Tools**

**Text editor**. A special program which allows to enter texts into the computer and store them somehow under some name as a sequence of characters, called **text file**.

The text file has to be compiled by a **compiler**. The JDK compiler is called `javac`.

The command line `javac HelloWorld.java` invokes the compiler with the name of the java file. Output: Error messages to the screen; a class-file to the store. Name of class file: `HelloWorld.class`.

Class file is executed by an **interpreter**. The JDK interpreter is called `java`.

The command line `java HelloWorld` invokes the `main` method of the class `HelloWorld` which is found in the file `HelloWorld.class`.

When run, the **output** of our example program is the text `Hello World!` and a line break.

## *Use of simple data types and the "while" loop*

```
// A simple demonstration program, printing out the
//   numbers from 0 to 10 and their squares, each pair
//   on a line by itself.
public class HelloWorld1
{
  /* We need only one method. */
  public static void main(String[] args)
  {
    int i;
    i = 0;
    while(i <= 10)
    {
      System.out.println(i + ":" + (i*i));
      i = i+1;
    }
  }
}
```

### While loop

`while` starts a **loop**: A sequence of commands which, under some condition, are executed repeatedly.

**First**, the condition given in parentheses is checked. Result must be boolean. **Our example**: Comparison of the current value of `i` (0) with `10`.

`0<10` is true: Thus, the body of the loop is executed: Pair of values 0 and 0*0 are printed, and `i` is incremented by one.

**Then**, execution continues with the check of the condition, and the loop is repeated until `i` has value `11`, such that `i <= 10` becomes false.

Then, the loop body is not repeated again, and the `main` method finishes.

*Assignments*

in our example:
**`i  =  0;`**
the variable named **`i`** gets the new value 0

- fundamental operation in the von Neumann programming paradigm

effect: content of a place in the memory is changed

**Attention:**
**`i  =  0`** in a Java programme does not have the same meaning as in a mathematical formula!
E.g., **`i = i+1`** would mathematically be a contradiction (it would imply 0 = 1)
– but makes sense in a programme (increment **`i`** by 1).
Mathematical meaning of this assignment:

$$i_{new} = i_{old} + 1.$$

In assignments, the *order is relevant*:
**`x1 = x2;`** has another effect as **`x2 = x1;`**

To underline the asymmetry, other languages (e.g., Pascal) use **`:=`** instead of **`=`** for assignments

*Comparison* (checking for equality) is expressed in Java by **`==`**

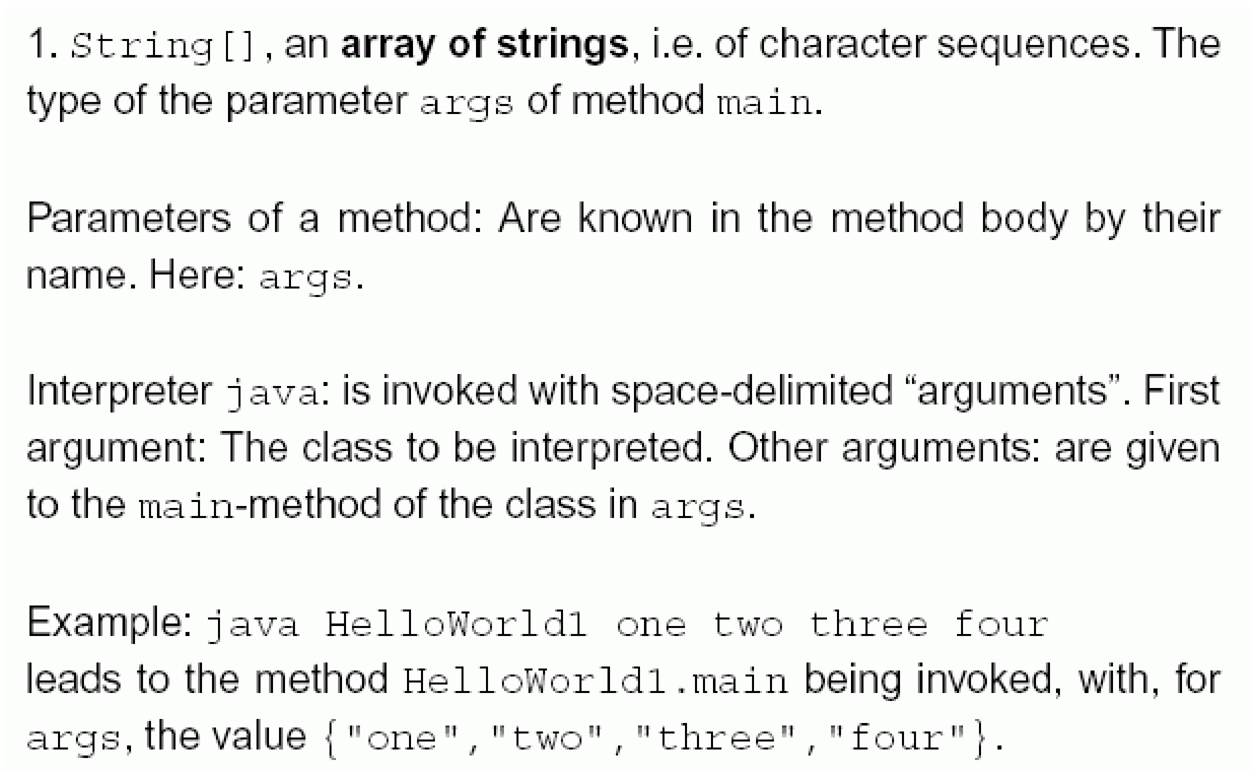
Java offers further assignment operators besides = :
**`a += b`** // add content of **b** to the content of **a**
**`-=, *=, /=`** etc. analogously.

*Data types*:

describe sets of values and the operations which can be performed on them.

Example: integers, with arithmetical operations (+, –, *, /, %) and comparisons (<, <=, >, >=, ...).

In the example programme:
**`String[]`**, **`int`**, **`String`**, **`PrintStream`**.



1. `String[]`, an **array of strings**, i.e. of character sequences. The type of the parameter `args` of method `main`.

Parameters of a method: Are known in the method body by their name. Here: `args`.

Interpreter `java`: is invoked with space-delimited “arguments”. First argument: The class to be interpreted. Other arguments: are given to the `main`-method of the class in `args`.

Example: `java HelloWorld1 one two three four`
leads to the method `HelloWorld1.main` being invoked, with, for `args`, the value `{"one","two","three","four"}`.

2. `int`, type of **32-bit two's-complement integers**. The variable `i` used for running through the argument list.

Variable `i` starts with value 0 and is incremented in the loop until it has value 11.

3. `String`, type of **character sequences**, the type of the expression `args[i]`, given as argument to the `println`-call, evaluates to an element of the `args` array.

4. `PrintStream`, type of **connections to some output**, `System.out` is a predefined value. On program start, the interpreter connects this character based output stream with the computer screen.

**Code fragment for computing the sum of an integer array**

```
// Compute, in (result), the sum of elements of (int[] p).
int result = 0;
{
  int i = 0;

  while(i<p.length)
  {
    result = result + p[i];
    i = i + 1;
  }
}
```

## *Ranges of declarations, visibility*

Example: `int i` introduced variable `i` which was known in all of `main`, starting from the point of declaration.

Visibility of a variable: is delimited by the closing brace of the range in which it is declared. In the same range, the same identifier can not be reused.

```
{
  int i;
  // -- 'i' is visible here --
  {
    int j;
    // -- 'i' and 'j' are visible here --
  }
  // -- 'i' is still visible, 'j' not any more --
}
```

## *Literals*

Literals denote values directly

**String literals**: Strings in quotes

Used character code for the string content: 16-bit Unicode

Special characters in strings: \: is used to introduce something "special". Examples:

**\uXXXX** (**XXXX**: up to four hexadecimal digits):
The number of a Unicode character

`\n`: a line break; `\t`: a tabulator; `\xxx`, `xxx` a three-digit n octal number: The character with the given octal code.

**Number literals**: Signed digit sequence for integer types; for float types: decimal point and "E"-Notation. Examples: +3453; 3.141592653; 1.17E-6

## *Primitive Java data types:*

| primitive data type | defaults | size (bits) | min/max |
|---|---|---|---|
| `boolean` | `false` | 1 | n.a./n.a. |
| Unicode characters: | | | |
| `char` | `\u0000` | 16 | `\u0000`/`\uFFFF` |
| Two's complement integers: | | | |
| `byte` | 0 | 8 | -128/127 |
| `short` | 0 | 16 | -32768/32767 |
| `int` | 0 | 32 | -2147483648/2147483647 |
| `long` | 0 | 64 | -9223372036854775808/ 9223372036854775807 |
| IEEE 754 floating-point numbers: (min/max are those of absolute values) | | | |
| `float` | 0.0 | 32 | 1.4023985E-45/3.40282347E+38 |
| `double` | 0.0 | 64 | 4.94065645841246544E-324/ 1.79769313486231570E+308 |

`void`: quasi-type for methods which return no value

Non-primitive Java data types: Arrays and objects

**Arrays**: collections of elements of the same type, accessed by **number** (from 0). Example declarations of integer arrays:

```
int[] p = {1,3,2,10};
int[] q = new int[5];
int[] r;
```

Values after these declarations:

`p` points to a memory block of four integers, with values 1, 3, 2 and 10.

`q` points to a memory block of five integers, all values 0.

`r` does not point anywhere (it has the special value `null`). This can be changed by the allocation of a block of memory via the Java operation `new`:

```
r = new int[1000];
```

Now, `r` points to a memory block of 1000 integers, all 0.

```
r = p;
```

Now, `r` points to the same memory block as p.

**Array declarations and operations**

Non-allocating declaration: `int[] a_empty;`

Allocated with room for 10 elements:
`int[] a_ten = new int[10];`

Initialized array: `int[] lookup = {1,2,4,8,16,32,64,128};`

Multiple dimensions: `boolean[][] bw_screen =`
`new boolean[1024][768];`

Non-rectangular: `int[][] pascal_triangle =`
`{{1},{1,1},{1,2,1},{1,3,3,1},{1,4,6,4,1},{1,5,10,10,5,1}};`

Array access: by integer-index in brackets. Start at 0. Array-access
is checked (index may not be negative or too large)

Number of elements of array `a`: `a.length`

**Objects**: collections of elements of arbitrary types, plus associated operations, accessed by **name**.

Object types must be **declared** before they can be used; example:

```
class color {
  String name;
  float red;
  float green;
  float blue;
}
```

Use of object types

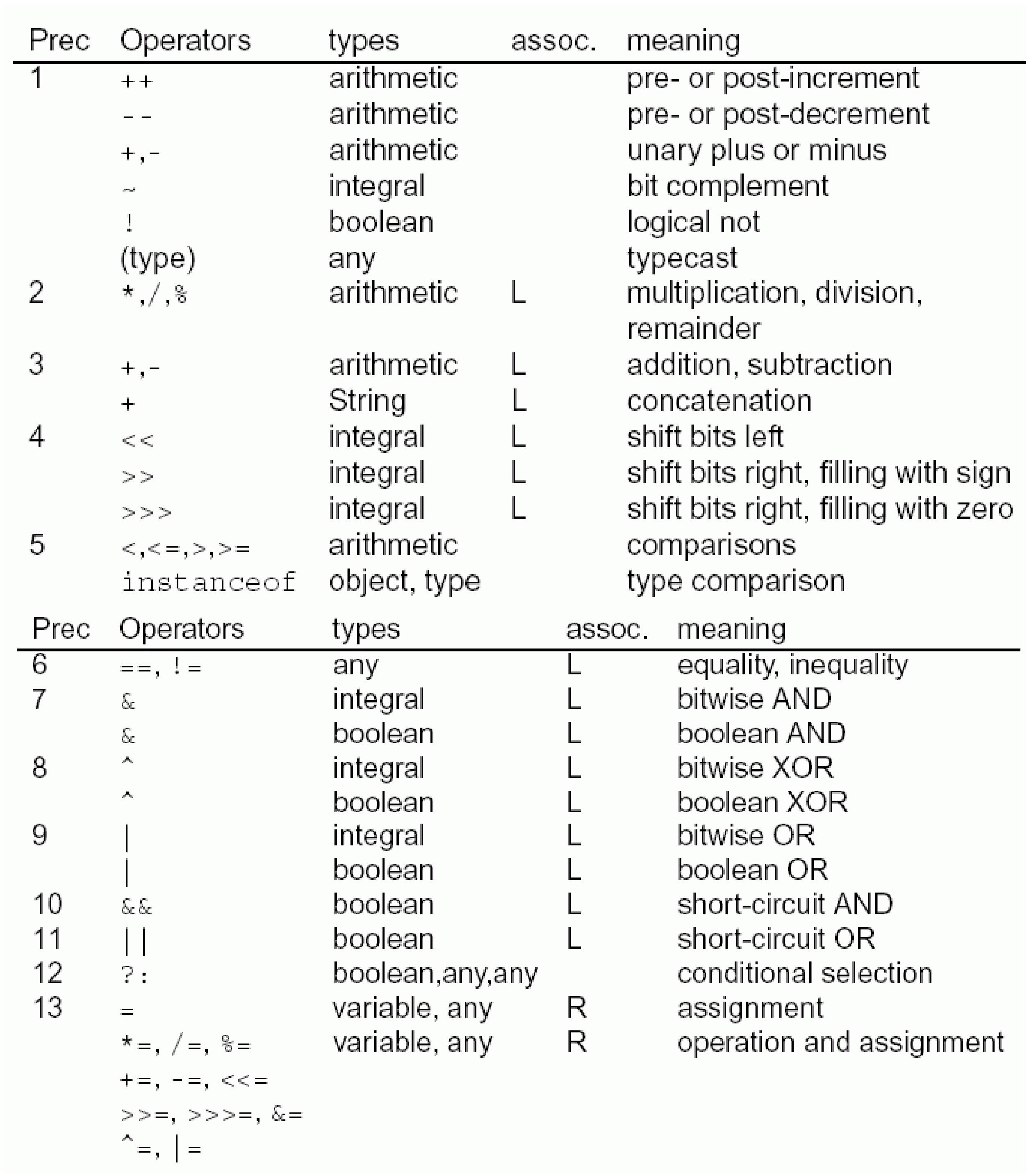
```
// Declare three color variables.
color r,w,b;

// Initialize the color variables to red, white and black.
r = new color;
r.name = "Red";   r.red = 1.0; r.green = 0.0; r.blue = 0.0;
w = new color;
w.name = "White"; w.red = 1.0; w.green = 0.0; w.blue = 0.0;
b = new color;
b.name = "Black"; b.red = 0.0; b.green = 0.0; b.blue = 0.0;
```

Both non-primitive data types are handled **by reference**: The variable content is just the address of a memory block.

An assignment to such a variable only changes this address, **not the data of the memory block**.

`null` is the default value for reference types

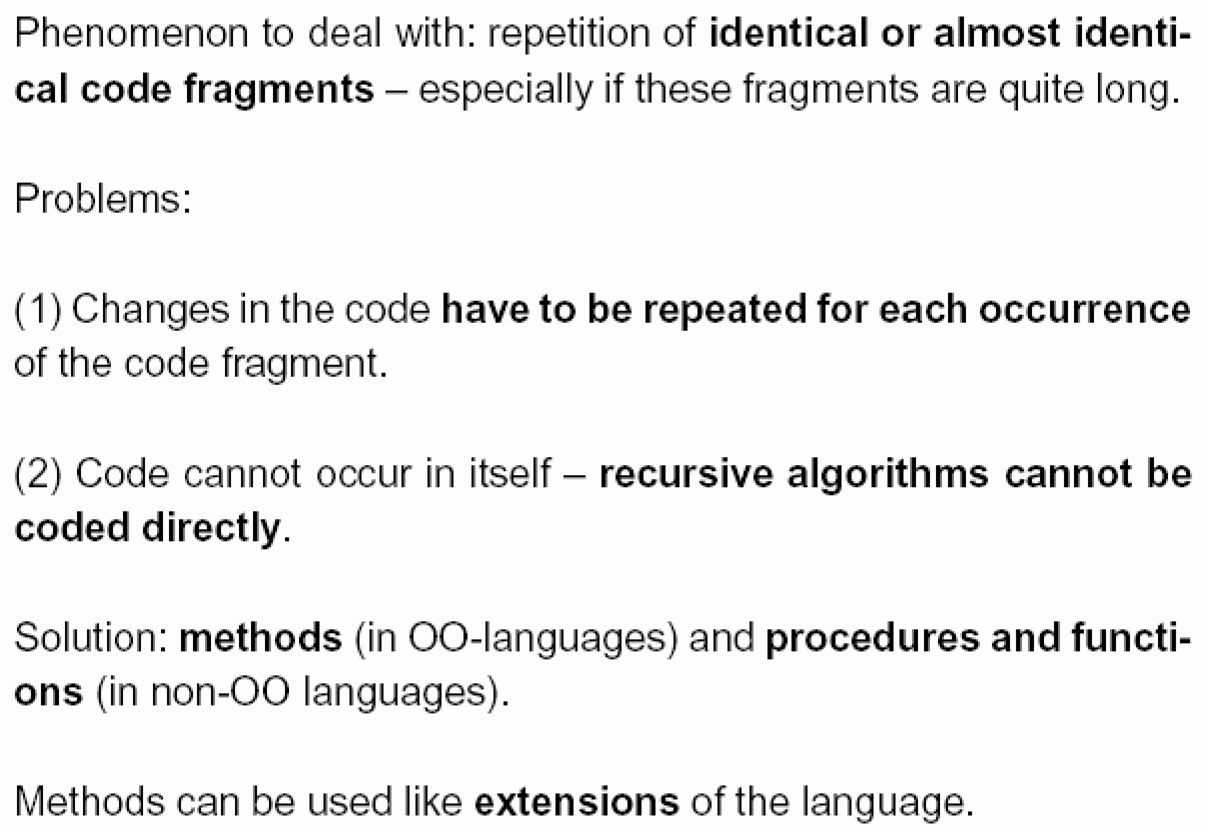
*Java operators*



| Prec | Operators | types | assoc. | meaning |
|---|---|---|---|---|
| 1 | `++` | arithmetic | | pre- or post-increment |
| | `--` | arithmetic | | pre- or post-decrement |
| | `+,-` | arithmetic | | unary plus or minus |
| | `~` | integral | | bit complement |
| | `!` | boolean | | logical not |
| | (type) | any | | typecast |
| 2 | `*,/,%` | arithmetic | L | multiplication, division, remainder |
| 3 | `+,-` | arithmetic | L | addition, subtraction |
| | `+` | String | L | concatenation |
| 4 | `<<` | integral | L | shift bits left |
| | `>>` | integral | L | shift bits right, filling with sign |
| | `>>>` | integral | L | shift bits right, filling with zero |
| 5 | `<,<=,>,>=` | arithmetic | | comparisons |
| | `instanceof` | object, type | | type comparison |
| 6 | `==, !=` | any | L | equality, inequality |
| 7 | `&` | integral | L | bitwise AND |
| | `&` | boolean | L | boolean AND |
| 8 | `^` | integral | L | bitwise XOR |
| | `^` | boolean | L | boolean XOR |
| 9 | `\|` | integral | L | bitwise OR |
| | `\|` | boolean | L | boolean OR |
| 10 | `&&` | boolean | L | short-circuit AND |
| 11 | `\|\|` | boolean | L | short-circuit OR |
| 12 | `?:` | boolean,any,any | | conditional selection |
| 13 | `=` | variable, any | R | assignment |
| | `*=, /=, %=` `+=, -=, <<=` `>>=, >>>=, &=` `^=, \|=` | variable, any | R | operation and assignment |

("assoc" = order of association, i.e., evalutation from left (L) or right (R) when several operators of the same level occur in the same expression)

## *Functional abstraction, self-defined methods*



Phenomenon to deal with: repetition of **identical or almost identical code fragments** – especially if these fragments are quite long.

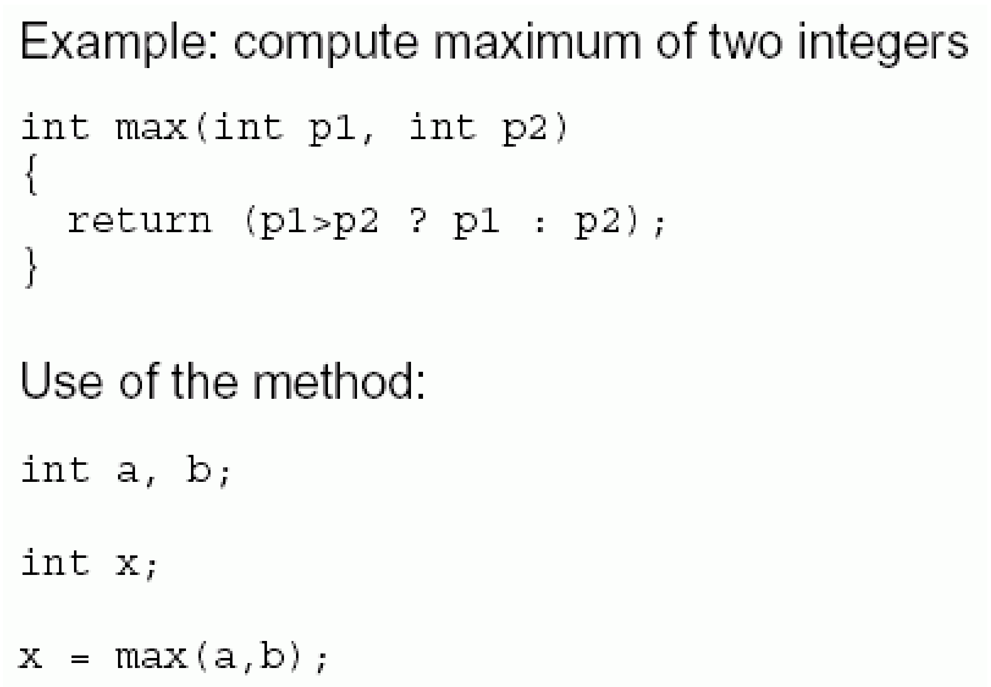
Problems:

(1) Changes in the code **have to be repeated for each occurrence** of the code fragment.

(2) Code cannot occur in itself – **recursive algorithms cannot be coded directly**.

Solution: **methods** (in OO-languages) and **procedures and functions** (in non-OO languages).

Methods can be used like **extensions** of the language.



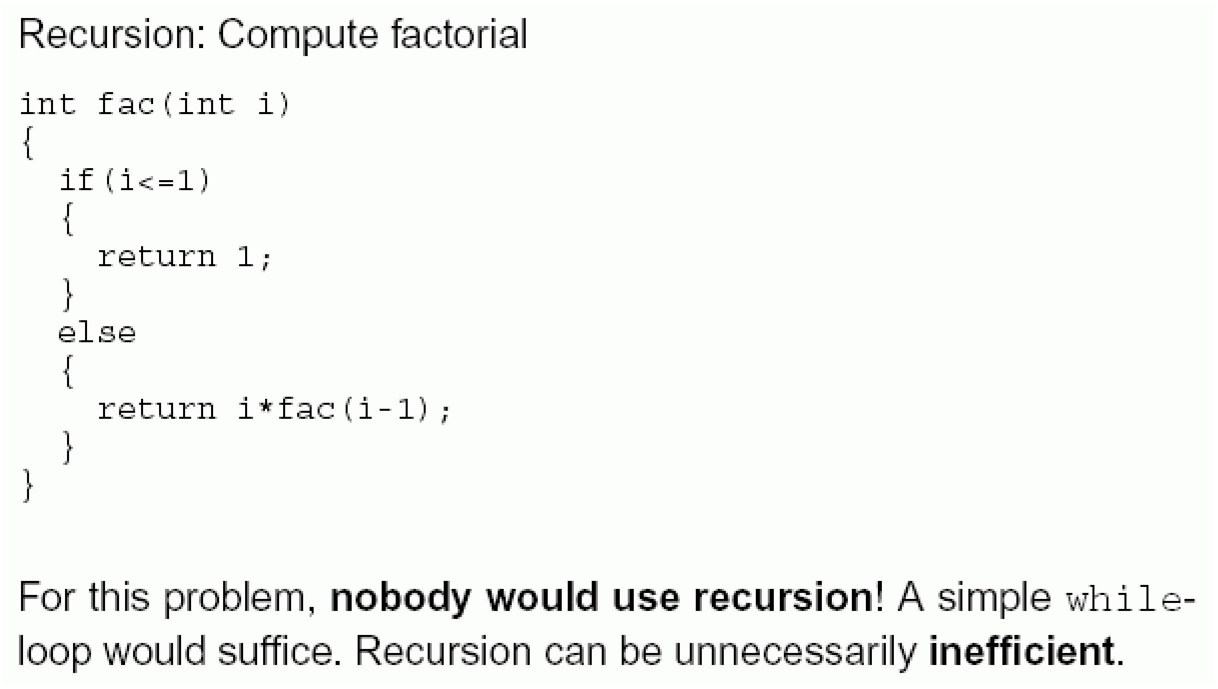
Example: compute maximum of two integers

```
int max(int p1, int p2)
{
  return (p1>p2 ? p1 : p2);
}
```

Use of the method:

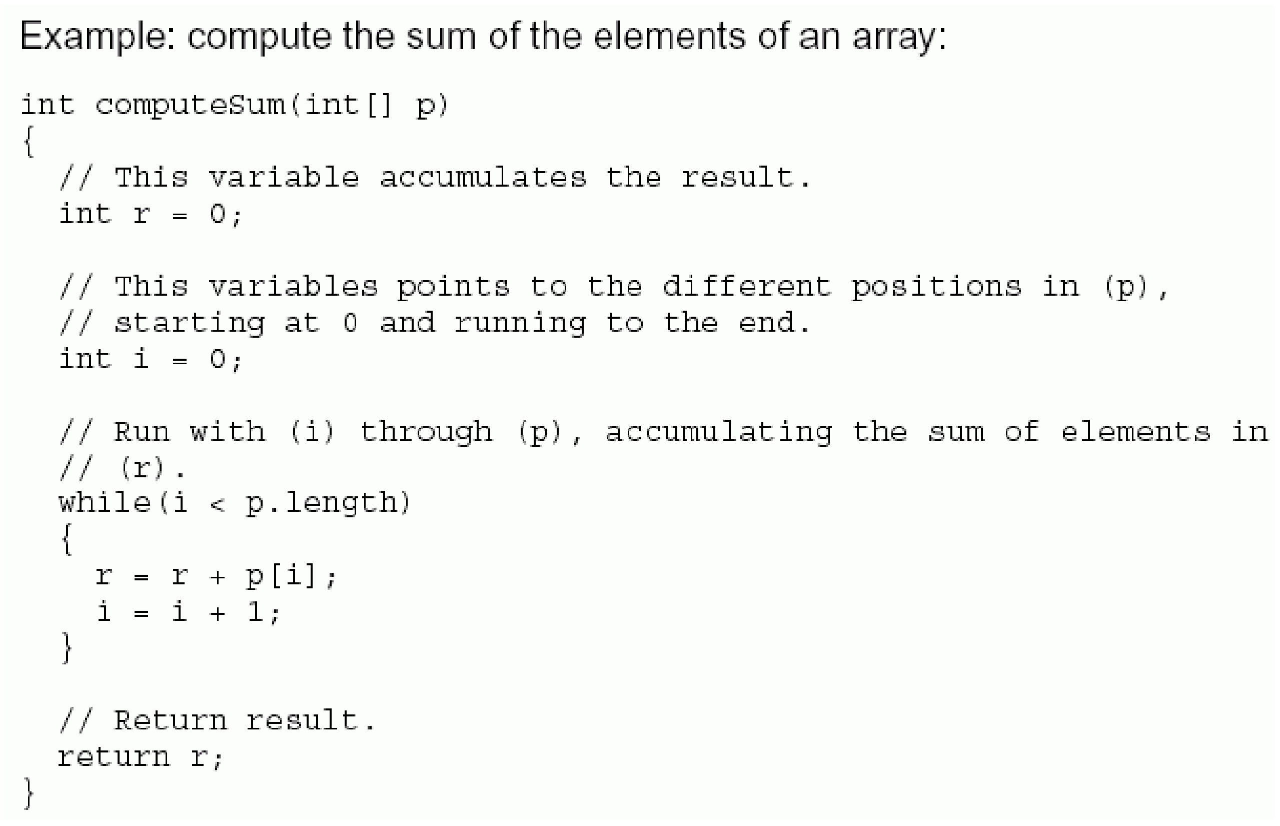
```
int a, b;

int x;

x = max(a,b);
```

Example: compute the factorial of an integer
"factorial":  n! = n * (n–1) * ... * 3 * 2 * 1.


Recursion: Compute factorial

```
int fac(int i)
{
  if(i<=1)
  {
    return 1;
  }
  else
  {
    return i*fac(i-1);
  }
}
```

For this problem, **nobody would use recursion**! A simple `while`-loop would suffice. Recursion can be unnecessarily **inefficient**.


Example: compute the sum of the elements of an array:

```
int computeSum(int[] p)
{
  // This variable accumulates the result.
  int r = 0;

  // This variables points to the different positions in (p),
  // starting at 0 and running to the end.
  int i = 0;

  // Run with (i) through (p), accumulating the sum of elements in
  // (r).
  while(i < p.length)
  {
    r = r + p[i];
    i = i + 1;
  }

  // Return result.
  return r;
}
```

Questions regarding `computeSum`: Details are important!

Does it work for empty `(p)`?

Is `<` the right comparison in the condition of the `while` clause, or would `<=` be right?

Should `i` start with another value than 0?

How could a solution look like in which `i` runs through `p` in the opposite direction?

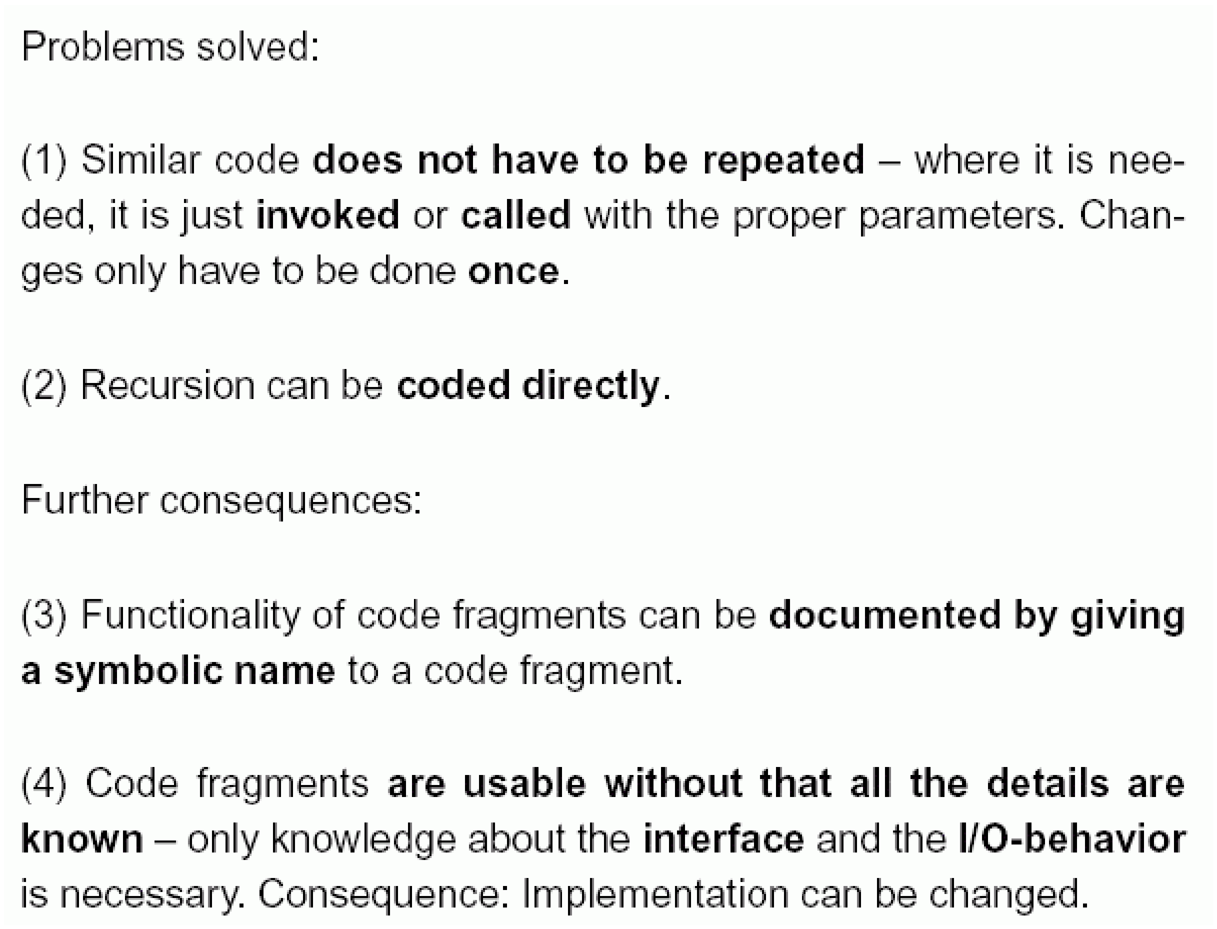
General structure of method declaration (incomplete version)

```
<type> <methodName> ( <parameterlist, empty for no parameters> )
{
  <method body, including ``return <expression>''>
}
```

**Method interface**: type of return value, name of method, and types and names of parameters.

**Method body**: code fragment performing the work.

`return` **statement**: Execution **leaves the method** and **returns the value of the expression** as result.


Problems solved:

(1) Similar code **does not have to be repeated** – where it is nee-ded, it is just **invoked** or **called** with the proper parameters. Chan-ges only have to be done **once**.

(2) Recursion can be **coded directly**.

Further consequences:

(3) Functionality of code fragments can be **documented by giving a symbolic name** to a code fragment.

(4) Code fragments **are usable without that all the details are known** – only knowledge about the **interface** and the **I/O-behavior** is necessary. Consequence: Implementation can be changed.

Method call:
e.g. **`x = max(a, b);`**
Effects:

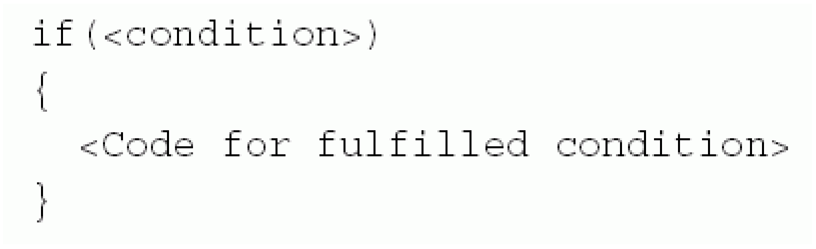
- control flow jumps from the place where the method is called to the place where the method is defined
- the method is executed
- the control flow jumps back to the place where the method was called and the return value is assigned to **`x`**.

*Control structures of Java*

control structures:
language concepts designed to control the flow of operations
– typical for the von Neumann paradigm

particularly: *branching* of the programme; *loops*.

Variants of branching:



```
if(<condition>)
{
  <Code for fulfilled condition>
}
```

(if the condition is false, nothing happens)

```
if (<condition>)
   {
      <Code for fulfilled condition>
   }
else
   {
      <Code for unfulfilled condition>
   }
```

Nesting of **`if...else`** possible:

```
if(<cond1>)
{
  <Code for fulfilled <cond1>>
}
else if(<cond2>)
{
  <Code for non-fulfilled <cond1>, but fulfilled <cond2>>
}
else
{
  <Code to be executed if NO condition is fulfilled>
}
```

## Example application: Finding the solutions of a quadratic equation ("pq-formula")

```
public static double[] solve_quadratic(double p, double q)
{
    double x = -p/2, y = x*x-q;
    double[] result;

    if(y<0)
    {
      // Term under square-root is negative.  No solution.
      result = new double[0];
    }
    else if(y < 1E-20)
    {
      // Term under square-root is zero.  One solution.
      result = new double[1];  result[0] = x;
    }
    else
    {
      // Term under square root is positive.  Two solutions.
      double z = Math.sqrt(y);
      result = new double[2];  result[0] = x+z;  result[1] = x-z;
    }
    return result;
}
```

Alternative: **`switch`** construction
Branching not binary, but with several alternatives at the same level

```
switch(<Expression>)
{
  case <Selector1>:
    <Code for the case <Expression>==<Selector1>>
    break;
  case <Selector2>:
    <Code for the case <Expression>==<Selector2>>
    break;
  ...
  default:
    <Code for the case that no
       selector equals <Expression>>
    break;
}
```

Example application:

```
public static String describe(Font font)
{
  String s;

  switch(font.getStyle())
  {
    case Font.ITALIC:
      s = "italic";
      break;
    case Font.BOLD:
      s = "bold";
      break;
    case Font.BOLD+Font.ITALIC:
      s = "bolditalic";
      break;
    default:
    case Font.PLAIN:
      s = "";
      break;
  }

  return s;
}
```

## Special form of branching for error handling: the **`try`** construction

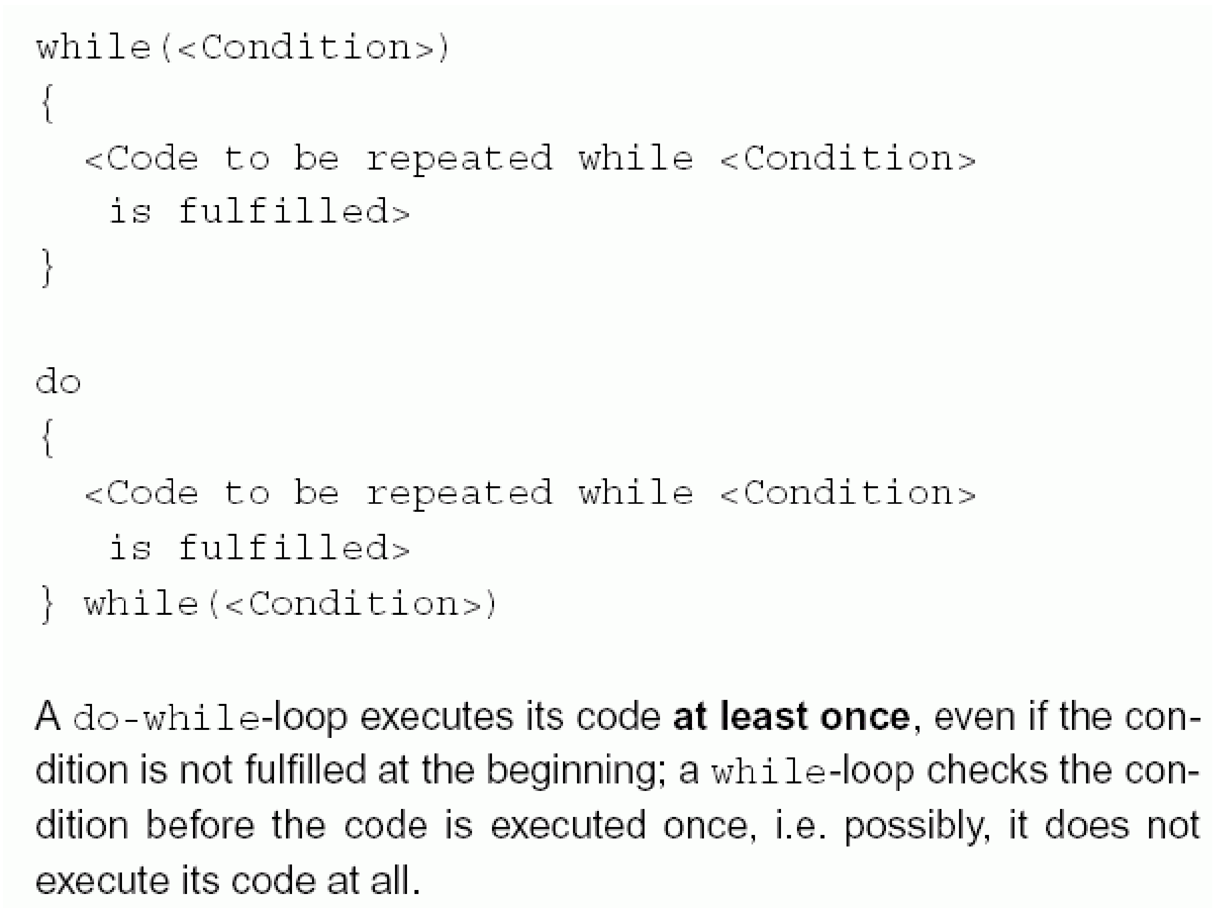
```
try
{
  <try-code>
}
catch(<exc1>)
{
  <Code to be executed if <exc1> is thrown in <try-code>>
}
finally
{
  <Code to be executed, if <try-code> finished
    in a normal or in some exceptional way.>
}

throw <exception>;
```

## Example application:

```java
static public int gcd(int p1, int p2)
  throws ArithmeticException
{
  // Compute absolute values.
  p1 = (p1<0 ? -p1 : p1);
  p2 = (p2<0 ? -p2 : p2);

  // Check that both parameters are strictly positive.
  if(p1==0||p2==0)
  {
    throw new ArithmeticException("Parameters not strictly positive");
  }

  while(p2!=0)
  {
    int tmp = p1 % p2;
    p1 = p2;
    p2 = tmp;
  }

  return p1;
}
```

```java
try
{
  int x = gcd(a,b);
  System.out.println("The greatest common divisor of "
                     + a + " and " + b
                     + " is " + x + ".");
}
catch(ArithmeticException ex)
{
  System.out.println("Cannot compute the greatest"
                     + " common divisor of " + a
                     + " and " + b + ".");
}
```

*Loops:*
we have already used the **`while`** loop.
Second variant: "**`do`** ... **`while`**"



```
while(<Condition>)
{
  <Code to be repeated while <Condition>
   is fulfilled>
}

do
{
  <Code to be repeated while <Condition>
   is fulfilled>
} while(<Condition>)
```

A `do-while`-loop executes its code **at least once**, even if the condition is not fulfilled at the beginning; a `while`-loop checks the condition before the code is executed once, i.e. possibly, it does not execute its code at all.

## The **`for`** loop

```
for(<Initialization>;<Condition>;<Increment>)
{
  <Code to be repeated>
}
```

Similar to:

```
<Initialization>;
while(<Condition>)
{
  <Code to be repeated>
  <Increment>
}
```

## Application example:

```
static public int computeSum(int[] p)
{
  int result = 0;

  for(int i=0; i<p.length; ++i)
  {
    result += p[i];
  }

  return result;
}
```