

5. Foundations of programming

Paradigms of programming:

Different viewpoints and ways of thinking about how to conceive a computer and a programme

Imperative paradigm:

Computer = machine for the manipulation of variables

Programme = sequence of commands which change values of variables, together with specifications of the *control flow* (telling which command is executed next)

Languages: Fortran, Pascal, Basic, C ...

Example (works in C or Java or XL):

```
x = 0;  
while (x < 100)  
    x = x + 2;
```

The variable **x** is used to produce the even numbers from 0 to 100.

Attention: The *assignment command* **x = x + 2** is not a mathematical equality!

Object-oriented paradigm:

Computer = environment for virtual objects which are created and destroyed during runtime (and can interact)

Programme = collection of general descriptions of objects (so-called *classes*), together with their hierarchical dependencies (*class hierarchy*)

Objects can contain data and functionality (*methods*)

Languages: Smalltalk, C++, Java, ...

Example (in Java):

```
public class Car extends Vehicle
{
    public String name;
    public int places;
    public void print_data()
    {
        System.out.println("The car is a " + name);
        System.out.println("It has " + places + "places");
    }
}
```

Typical: class (**Car**) with data (**name**, **places**) and methods (**print_data**). The class **Car** *inherits* further data and methods from a superclass, **Vehicle**.

Rule-based paradigm:

Computer = machine which transforms a given structure according to given rules

Programme = set of transformation rules
(sometimes also called a *grammar*)

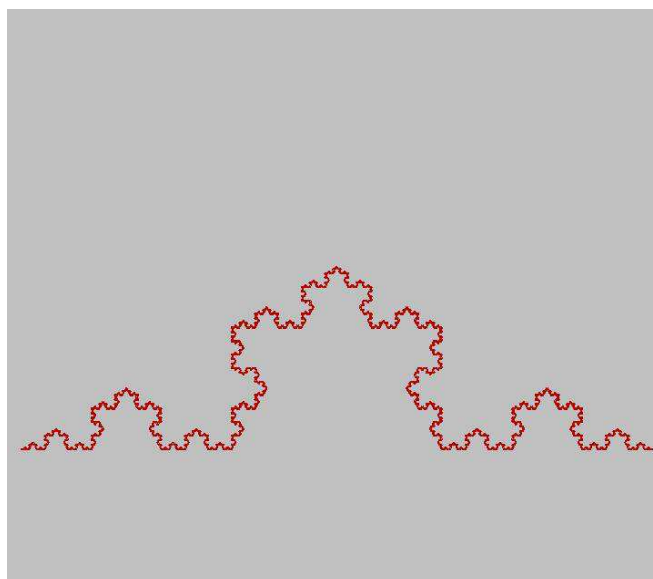
Each step of programme application consists of two substeps: Finding an applicable rule (*matching step*) and transformation of the current structure according to that rule (*rewriting step*).

Languages: Prolog, AI-languages, L-system languages, particularly XL

Example (in XL):

```
public void apply()  
[  
  F(x)  ==> F(x/3) RU(-60) F(x/3) RU(120)  
          F(x/3) RU(-60) F(x/3);  
]
```

produces the so-called Koch curve:



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, C++ or XL)
- 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!*

Basic parts of Java and XL

Remark: The language XL is an extension of Java. The following examples can be compiled and run with GroIMP (see www.grogra.de), a modelling platform which contains a development toolkit for XL and possibilities for visualization.

A first demonstration programme:

```
/* A simple Java programme for execution  
with the GroIMP software. */  
protected void init()  
{  
    println("Hello World!");  
}
```

(= example file **prog_ex01.rgg**)

Download of GroIMP:

<https://gitlab.com/grogra/groimp/-/releases>

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 , protected`

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

```
println: predefined method – invoked with a string as its  
argument, it writes the string to the GroIMP console (a special  
output window) and adds a line feed.
```

3) newly declared identifiers: Their meaning is fixed by (explicit or implicit) declarations in the programme itself. Example: **init** is the name of the method which writes the text to the console. It expects no arguments (**init()**).

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 an extra line. */
```

```
protected void init()  
{  
    int i;  
    i = 0;  
    while (i <= 10)  
    {  
        println(i + ": " + (i*i));  
        i = i+1;  
    }  
    println("Finished!");  
}
```

(example file **prog_ex02.rgg**)

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 imperative 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.

XL allows both notations

(but with a slightly different meaning: **:=** denotes a deferred assignment, i.e., it enables a quasi-parallel execution with other assignments.)

Comparison (checking for equality) is expressed in Java, C and XL by **==**

Java offers further assignment operators besides **=** :

a += b // add content of **b** to the content of **a**

-=, ***=**, **/=** etc. analogously.