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*)

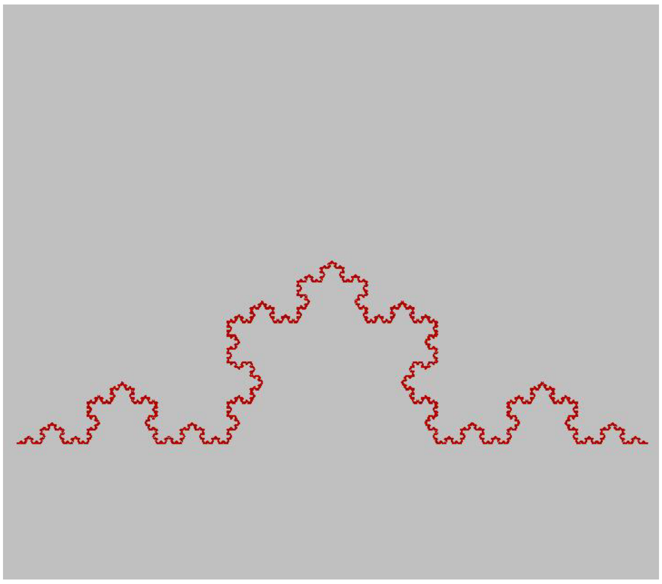
Each step of programme application consists of two substeps: Finding an applicable rule (*matching step*) and transformation of the current structure according to that rule (*rewriting step*).

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://sourceforge.net/projects/groimp/

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.

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: **`int`**, **`String`**.

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

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

`String`: type of character sequences. **`println`** expects a variable of this type as its argument.

Numbers are implicitly converted to strings here. Concatenation of strings by **`+`**.

("*Operator overloading*": different meanings of + for numbers and for strings.)

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 = 1.0;  w.blue = 1.0;
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

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

("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)