4. The World Wide Web

The World Wide Web (WWW) is
a *hypertext system* which is accessible via *internet*

(WWW is only one system using the internet –
others are e-mail, ftp, telnet, internet telephone ...)

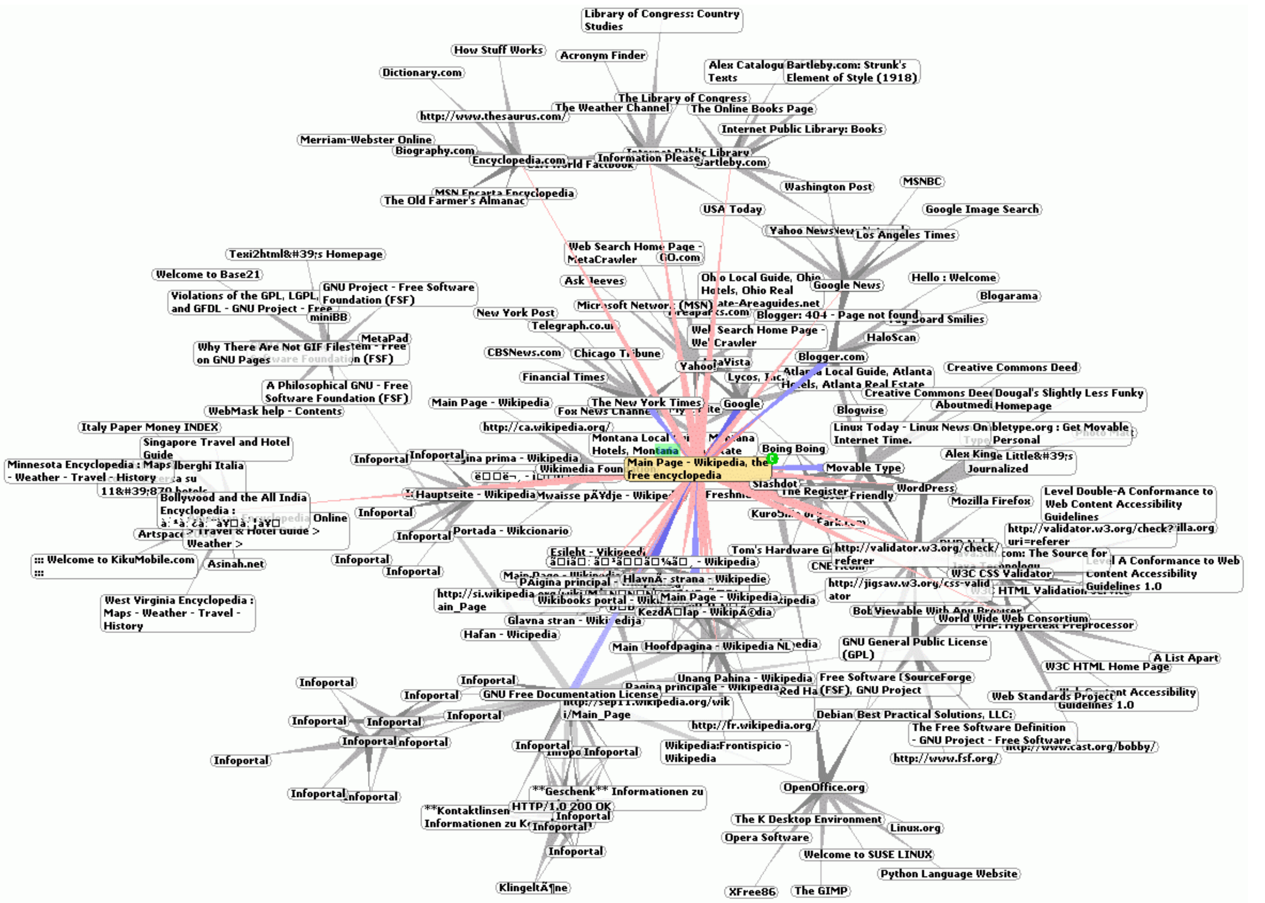
Hypertext: Pages of text containing *hyperlinks*
(short: *links*) referring to other pages

Hypertext is viewed using a program called a web browser which retrieves pieces of information, called "documents" or "web pages", from web servers and displays them, typically on a computer monitor. One can then follow hyperlinks on each page to other documents or even send information back to the server to interact with it. The act of following hyperlinks is often called "*surfing*" or "*browsing*" the web. Web pages are often arranged in collections of related material called "web sites."

(from **`www.wikipedia.org`**, the open www encyclopedia)

The link structure of the web forms a very large graph –

the following is a very small subgraph of it:



The Web can be seen as a sort of database – but very different from relational databases:

- highly distributed, decentralized;
- based on the hypertext model instead of the entity-relationship model;
- with only very weak standards to restrict form and content of the pages;
- very large
- without a universal query language.

(*Search engines* try to compensate the last item; see below.)

History of the WWW:

- Idea of hypertext: Vannevar Bush 1945
- Origin of WWW: a project at CERN (Geneva) in 1989
- *Tim Berners-Lee* and *Robert Cailliau*
- their system: ENQUIRE, realized core ideas of the Web in order to enable access to library information that was scattered on several different computers at CERN
- proposal for the WWW: published by Berners-Lee on November 12, 1990
- *first web page* on November 13 on a NeXT workstation
- Christmas 1990: Berners-Lee built the first web browser and the first web server
- August 6, 1991: summary of the WWW project posted in a newsgroup in the internet
- April 30, 1993: CERN annouced that the WWW would be free to anyone
- 1993: Browser Mosaic (forerunner of Internet Explorer or Firefox) starts to popularize the WWW

The three core standards of the Web:

• Uniform Resource Locator (URL): specifies how each page of information is given a unique address at which it can be found (e.g., **`http://en.wikipedia.org/wiki/World_Wide_Web`**)

• Hypertext Transfer Protocol (HTTP): specifies how the browser and server send the information to each other

• Hypertext Markup Language (HTML): a webpage description language used to encode the information so that it can be displayed on a variety of devices and under different operating systems.

Later extensions:

- Cascading Style Sheets (CSS): define the appearance of elements of a web page, separating appearance and content
- XML: more general language than HTML, designed to enable a better separation of appearance and content; also applicable to other sorts of information
- ECMAScript (also called JavaScript or JScript): a programming language with commands for the browser, enables embedding of programmes (scripts) into web pages. Thus web pages can be changed dynamically.
- Hypertext Transfer Protocol Secure (HTTPS): Extension of HTTP where the protocol SSL is evoked to encrypt the complete data transfer
- Java applets (small programmes) can be embedded in web pages and run on the computer of the Web user

The *World Wide Web Consortium* (*W3C*) develops and maintains some of these standards (HTML, CSS) in order to enable computers to effectively store and communicate different kinds of information.

Problems with the Web:

- *highly decentralized*, no control of the content

→ there is a lot of false and misleading information, hate campaigns, promotion of sexual exploitation, of terrorism and of other crimes...

- *highly dynamic*: *Web pages change all the time!*

Links point to nowhere when the target page was removed...

→ when you give a Web address in the References section of a scientific paper or in your thesis, you should add the *date* when you visited that page!

Archive of (a part of) the Web:

http://archive.org

→ lost Web references can (in some cases) be reconstructed if the date is known

- *highly chaotic*: no global index or table of content is available; search for a certain content is complicated and time consuming

→ development of specialized search engines,
the most well-known one: *Google* (**http://www.google.de**)

How does a search engine work?

- First component: a web crawler, visiting all accessible web pages worldwide, one after the other, following the hyperlinks

but: when you look for a certain keyword, this process would take much too long!
→

- second component: a large database, containing keywords and web addresses where these keywords were already found

the web crawler is working in the background and does only actualize the database

when you invoke Google, you search in Google's database, not in the Web!

→ not all Web pages can be found, because not all are in the database

Usually, you get many, many, many Web pages containing a given keyword (often millions...)
→
first remedy: make more intelligent queries
e.g., combining several keywords by "and", or looking for phrases instead of keywords (use quotation marks)
– Google provides such facilities under "extended search"

still there are often too many results

$\rightarrow$ priorisation of the found web pages necessary

- third component of the search engine (and best capital of the Google company): a *ranking algorithm* for search results

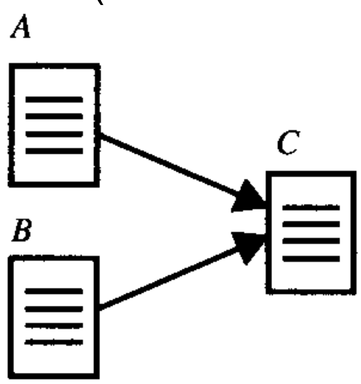
Basic principles of Google ranking of web pages

(Attention: the exact algorithm is changing continuously and is not published)

"Importance" of a web page:
recursively defined, using the hyperlink structure of the Web

The importance of a page is the larger, the more important pages refer to it!

More precisely:
Let *FLinks*(*A*) be the set of all outgoing links (forward links) of a page *A* and *BLinks*(*A*) the set of all incoming links (backward links) of *A*



$FLinks(A) = \{C\}$
$FLinks(B) = \{C\}$
$BLinks(C) = \{A, B\}$

- *A* has high page rank if the sum of the page ranks of its incoming links is high,
- a page *B* distributes its importance in equal parts to all pages which are referred by it:

$$PageRank(A) = \frac{1}{c} \sum_{B \in BLinks(A)} \frac{PageRank(B)}{|FLinks(B)|}.$$

(c = normalisation factor)

Iterative determination of the page rank:

- initially, an arbitrary mapping of values to all web pages is done (typically, the *constant value* 1 is used),
- *iterate the calculation* using the above formula for all pages, until the values remain stable,
- they *converge* against the Eigenvectors of the adjacency matrix of the graph consisting of the web pages (nodes) and their links (edges). (*Adjacency matrix*: $a_{ij} = 1$ iff nodes i and j are connected by an edge.)

Additionally, the Google page rank utilizes:

- *proximity* of the given key words to each other (in the text),
- the *anchor texts* of the links: these are the texts which can be clicked upon. A page *A* gets higher importance when the anchor texts of links referring to *A* contain the keywords, too.

The underlying technology of the WWW:
the ***Internet*** (short for "Interconnected Networks")

predecessor (end of the 1960s): ARPANET (U.S. military project)
was later used to connect universities and research labs

Internet today: A worldwide network of computer networks

- Computers in this network communicate using the standardized *TCP/IP protocol* (Transmission Control Protocol / Internet Protocol: Rules governing the communication)
- Transmission of the information in small portions
- For identification, each computer in the net has a unique number, the *IP address*
- IP address: 32 bit integer; for better comprehensibility usually split in 4 bytes (these 4 bytes are often written as decimal integers, separated by dots: e.g., 194.77.124.35)
  $\rightarrow$ more than 4 billion addresses
- to get identifiers which can better be memorized: *Domain Name System* (DNS)
  – system of (textual) names,
  association between names and IP addresses

- hierarchy: Domains, subdomains, sub-subdomains..., e.g.,
  `www.uni-forst.gwdg.de`
  (from right to left!)
- *Top-level domains*: Country abbreviations and some others ("generics"): .de, .fr, .eu, .com, .edu, .gov ...
- Lowest level: host name of a single computer (here: www, Web server of the forestry faculty)
- domain name corresponds to IP address
- transformation of domain names into IP addresses and vice versa: Task of special computers, so-called *nameservers*
- this transformation takes place any time when you click on a hyperlink on a web page!
- each nameserver is responsible for a certain part of the hierarchical name space

5. Foundations of programming

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, ...)

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*)

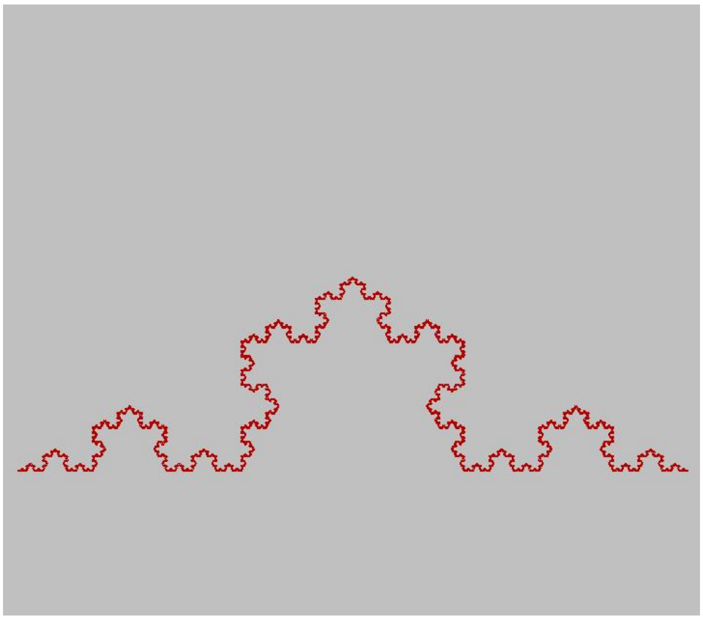
Each step of programme application consists of two substeps: Finding an applicable rule (*matching step*) and transformation of the current structure according to that rule (*rewriting step*).

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`**)

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.)

Ranges of declarations, visibility

Example: In our last example, **`i`** can be checked and used inside the method **`init`**, from its declaration on.

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 = 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><td>Prec</td><td>Operators</td><td>types</td><td>assoc.</td><td>meaning</td></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)

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
Reminder: "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 (**prog_ex03.rgg**): Usage of compound data structures (*arrays*)

```
/* Computation of the sum of elements of
an integer array. */

protected void init()
   {
   int result = 0;
   int[] p = { 4, 3, 3, 5, 15 };
         /* initialization of an array */

   int i = 0;
   while (i < p.length)
      {
      result += p[i];
      i = i+1;
      }
   println("The sum is: " + result);
   }
```

The same as an extra method:

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 needed, it is just **invoked** or **called** with the proper parameters. Changes 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 imperative programming 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")

prog_ex04.rgg

```
/* Computation of the solutions of a quadratic
   equation, using a self-defined method */

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

   if (y < 0)
      {
      // term under the square root is
      // negative. No solution.
      result = new double[0];
      }
```

```
   else
      if (y < 1e-20)
         {
         // term under the square root is zero.
         // One solution.
         result = new double[1];
         result[0] = x;
         }
      else
         {
         // term under the 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;
   }

module A(double p, double q) extends Sphere(3);

protected void init()
{
   [
   Axiom ==> A(0, 0);
   ]
   println("Click on object for input (p,q)!");
}

public void calculate()
{
   double[] res;
   double p, q;

   [
   a:A ==> { p = a[p]; q = a[q]; };
   ]
```

```
    res = solve_quadratic(p, q);

    if (res.length == 0)
       println("There is no solution.");
    if (res.length == 1)
       println("Single solution: " + res[0]);
    if (res.length == 2)
       {
       println("First solution: " + res[1]);
       println("Second solution: " + res[0]);
       }
}
```

Alternative to if-else:
`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;
}
```

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>;
```

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;
}
```