Goal:

- Reusability of frequently used sequences of instructions

Typical approaches:

- methods, procedures, functions, subprograms, macros, ...

Remarks:

- Similar concepts exist in practically all programming languages.
- We use *subprograms* and *procedure* also as more general terms for the different approaches.
- Every executable Java program contains a method **`main`** which is executed when the program starts.

Terminology with methods:

```
1  public class K3B25E_Rectangle {
2
3    static int wd, ht;
4    static int perimeter (int width, int height){ definition of method 1
5       return 2 * width + 2 * height;
6    };
7
8    static int area (int width, int height){ definition of method 2
9       return width * height;
10   };
11
12   public static void main(String[] args) {
13     int per, ar;
14
15     wd = Integer.parseInt(System.console().readLine("width : "));
16     ht = Integer.parseInt(System.console().readLine("height:  "));
17
18     per = perimeter(wd, ht); method call 1
19     ar = area  (wd, ht); method call 2
20
21     System.out.println ("perimeter: " + per + "\narea: " + ar );
22     System.exit (0);
23   }
24 }
```

```
1   static(1) int(2) perimeter(3) (int width, int height(4) )  {
2       int per;
3       per = 2 * width + 2 * height;
4       return per(5);
5   }
6 ...
7 myPerimeter = perimeter(6) (wd, ht(7));
```

(1) modifier	`static`
(2) return type	`int`
(3) method name	`perimeter`
(4) formal parameters	`(int width, int height)`
method head	`static int perimeter (int width, int height)`
method body	`{int per; ...; return per;}`
(5) return value	`return per;`
(6) method call	`perimeter (wd, ht)`
(7) actual parameters	`wd, ht`

- Usually a procedure executes a closed algorithm for *changing* input data.
- In programming languages input and output data are exchanged between caller and procedure in different ways:
  - through parameters
    (input and output values)
  - through function values
    (only output values, only functions)
  - through global variables
    (input and output values)

formal parameters / actual parameters:

In the declaration of the method, the formal parameters are placeholders for the actual parameters given when calling the method:

```
1   static int perimeter (int width, int height){ formal parameters
2     return 2 * width + 2 * height; } return value
3     ...
4    perimeter (wd, ht); actual parameters
```

- Input parameters can be given as expressions (see below).
- The value of an input parameter defines the initial value of the corresponding formal parameter.
- Actual parameters must be compatible to the formal parameters.

Parameter passing:

Techniques to pass actual parameters to subprograms:

- **Call by reference** (for input and output parameters)
  Passes the address of the variable such that the name of the formal parameter identifies the storage space of the actual parameter during the execution of the procedure.
- **Call by value** (for input parameters)
  Evaluation of the actual parameter and assignment of this value to the corresponding formal parameter.
- **Call by result** (for output parameters)
  Evaluation of the return value and assignment of this value to the corresponding actual parameter.
- **Call by name** (for input and output parameters)
  The formal parameter is exchanged by the actual parameter in the program text. During the execution of the procedure every instance of the actual parameter must be newly evaluated.

Execution of **Call by value**:

- With **Call by value** the actual parameters are first copied to newly allocated memory locations for the formal parameters.
- Only the formal parameters are changed in the procedure (i.e., only the copies!).
- Modifications of the formal parameters thus do not change the actual parameters (but: see later!).
- If large objects are passed this way, this is very costly in terms of computational time (the actual copying)
  and memory usage (data is stored redundantly).

Consider the following procedure:

```
1 int doSomething (int hugo, int oskar) {
2 //assumption: call using Call by value...
3    hugo = hugo - 11;
4    oskar ++;
5    return hugo + oskar;
6 }
```

```
1 value = doSomething ( a, b );
```

changes **value**, but not **a** or **b**.

Since we copy the data, actual parameters may be expressions, e.g.,

```
1 value = doSomething (3*a, b+a);
```

Execution of **Call by reference**:

- Instead of copying the data, references to the data are passed.
- Thus the formal parameter only states where in memory the actual parameter is stored.
- Every change of the formal parameter thus changes the actual parameter.

```
1 int doSomething (int hugo, int oskar) {
2 //assumption: call using Call by reference...
3    hugo = hugo - 11;
4    oskar ++;
5    return hugo + oskar;
6 }
```

```
1 value = doSomething ( a, b );
```

changes **value**, and also **a** and **b**!

The actual parameters must not be expressions!

Parameter passing in Java

- Input parameters of primitive type (**`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`boolean`**, **`char`**) are passed using **Call by value**.
- Output parameters of primitive type (**`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`boolean`**, **`char`**) are returned as a function value (mostly **Call by result**, the return value can be directly used in an expression)
- Objects (including arrays and String) are identified using reference variables (see later). A method is passed only the value of the corresponding reference variable (using **Call by value**). This '**Call by reference-value**' is suitable for input and output parameters.
- Remark: **Call by reference** would pass a reference to the reference variable. One could then change it such that it referenced a new object.

Function values:

- Functions return their result in form of a function value.
- Functions can be called directly in procedures.
- The type of the function determines the type of the return parameter.

Example declaration:

```
1  int perimeter (int width, int height)  {
2    return 2 * width + 2 * height; }
3
4  int area (int width, int height)  {
5    return width * height;   }
```

Example calls:

```
1  int w = 7, h = 5;
2  y = perimeter (w, h);
3  x = 3 * area (5, y / 4);
4  if (area (5, 12) == perimeter (12,18)) {...}
```

void functions:

Functions that do not return a value ('type' **`void`**) are called 'stand-alone'.

Example:

- declaration:

```
1  void perimeter (int width, int height)  {
2    int per;
3    per = 2 * width + 2 * height;
4    System.out.println(per);
5  }
```

- call:

```
1  perimeter (7,5);
```

Local variables and global variables:

A variable

- is local in the block B where it is declared
- is global in all inner blocks of B
- is unknown in all blocks outside of B
- is known from the point of its declaration to the end of the corresponding block
- ceases to exist when its local block is left

A procedure does not have its own internal memory, all locally defined variables cease to exist once the procedure ends.

If a procedure is executed repeatedly, all required information must be passed as input parameters!

- If a procedure changes a global variable, we call this a **side effect**. Example: In Java a method can access all variables known in the surrounding class (**`class`**).
- In some programming languages, global variables can be hidden by local variables, e.g.,

```
1  int x;
2  int z (int y) {
3    int x;
4    x = 5*y;     //  outer x remains unchanged
5    return x+y;
6  }
```

Examples for side effects, i.e., changing global variables in a procedure:

```
1  public class K3B26E_Sideeffect {
2
3     static int x,y;
4
5     static void swap ()  {
6        int help;
7        help = x;    // x and y are global !
8        x = y;
9        y = help;
10    }
11
12    public static void main(String[] args) {
13
14       x = Integer.parseInt(System.console().readLine("x: "));
15       y = Integer.parseInt(System.console().readLine("y: "));
16
17       System.out.println("before:  x= " + x + "  y= " + y );
18
19       swap ();
20
21       System.out.println("after: x= " + x + "  y= " + y );
22    }
23 }
```

Visibility in Java: Variables from outer blocks may not be 'overwritten' in Java!

```
1  public class K3B27E_Block {
2    public static void main(String[] args) {
3      int a = 3, b = 4;
4      { int c = 5, d = 6; }
5      { int c = 6, e = 7;
6          { int a = 2, f = 1;
7            int c = 9;
8          }
9      }
10   }
11 }
```

error messages from the compiler:

```
1  K3B27_Block.java:9: error: variable a is already defined ...
2          { int a = 2, f = 1;
3                ^
4  K3B27_Block.java:10: error: variable c is already defined ...
5            int c = 9;
6                ^
7  2 errors
```

(This would be allowed, for example, in **C++**!)

Methods in Java:

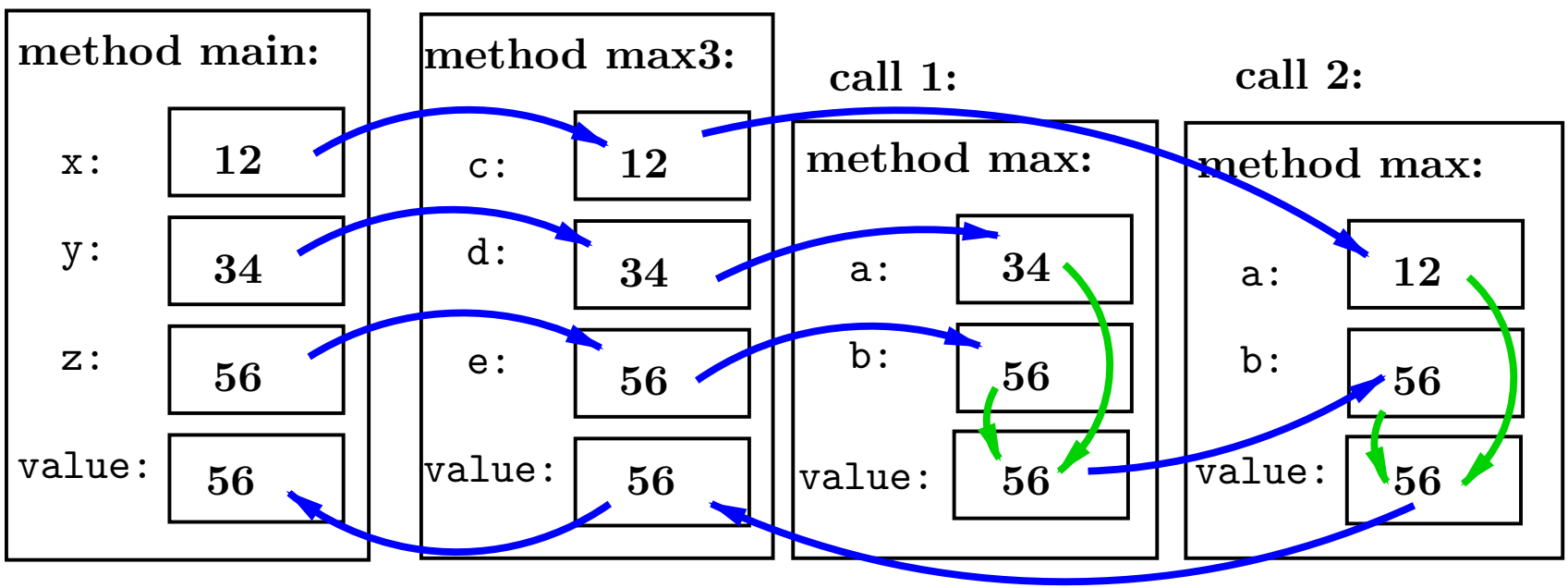
- Java allows only functional procedures, called ‘methods’.
- Parameter passing in and out of methods:
  - Input:
    - ★ parameters
      Call by value (variables, primitive types)
      ‘Call by reference-value’ (referenced types)
    - ★ global variables (can be bad programming style...)
  - Output:
    - ★ function value
      Call by result
    - ★ parameters
      ‘Call by reference-value’ (referenced types)
    - ★ global variables (can be bad programming style...)

- The **`return`** instruction causes the end of the procedure. Any method can include an arbitrary number of **`return`** instructions (including none).
- Local variables have no value when they are declared.
- Methods can be overloaded. (see later...)
- Methods can call themselves (recursive methods). (see later...)
- Methods are defined within classes. (see later...)
- Methods cannot be defined within other methods, ‘local’ methods exist only with restrictions.

Method calls can be nested arbitrarily deep:

```
1  public class K3B28E_Nesting {
2
3     static int max (int a, int b)  {
4        return a > b ? a : b;
5     }
6     static int max3 (int c, int d, int e)  {
7        return max (c, max (d, e) );
8     }
9
10    public static void main(String[] args) {
11
12       int x, y, z;
13       x = Integer.parseInt (args[0]);
14       y = Integer.parseInt (args[1]);
15       z = Integer.parseInt (args[2]);
16
17       System.out.println("Maximum: " + max3 (x, y, z) );
18
19    }
20 }
```

```
1  ...
2     static int max (int a, int b)  {
3        return a > b ? a : b;
4     }
5     static int max3 (int c, int d, int e)  {
6        return max (c, max (d, e) );
7     }
8     public static void main(String[] args) {
9        int x, y, z;
10       ...  max3 (x, y, z) )  ...
```



Recursive methods

- A method/procedure is called **recursive** when it calls itself.
- At runtime a separate memory area is reserved for every call (for parameters, local variables, etc).
- With recursive methods it is often very simple to implement problems defined in a recursive way.
- Example 1, factorial of n:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

- Example 2, Fibonacci numbers:

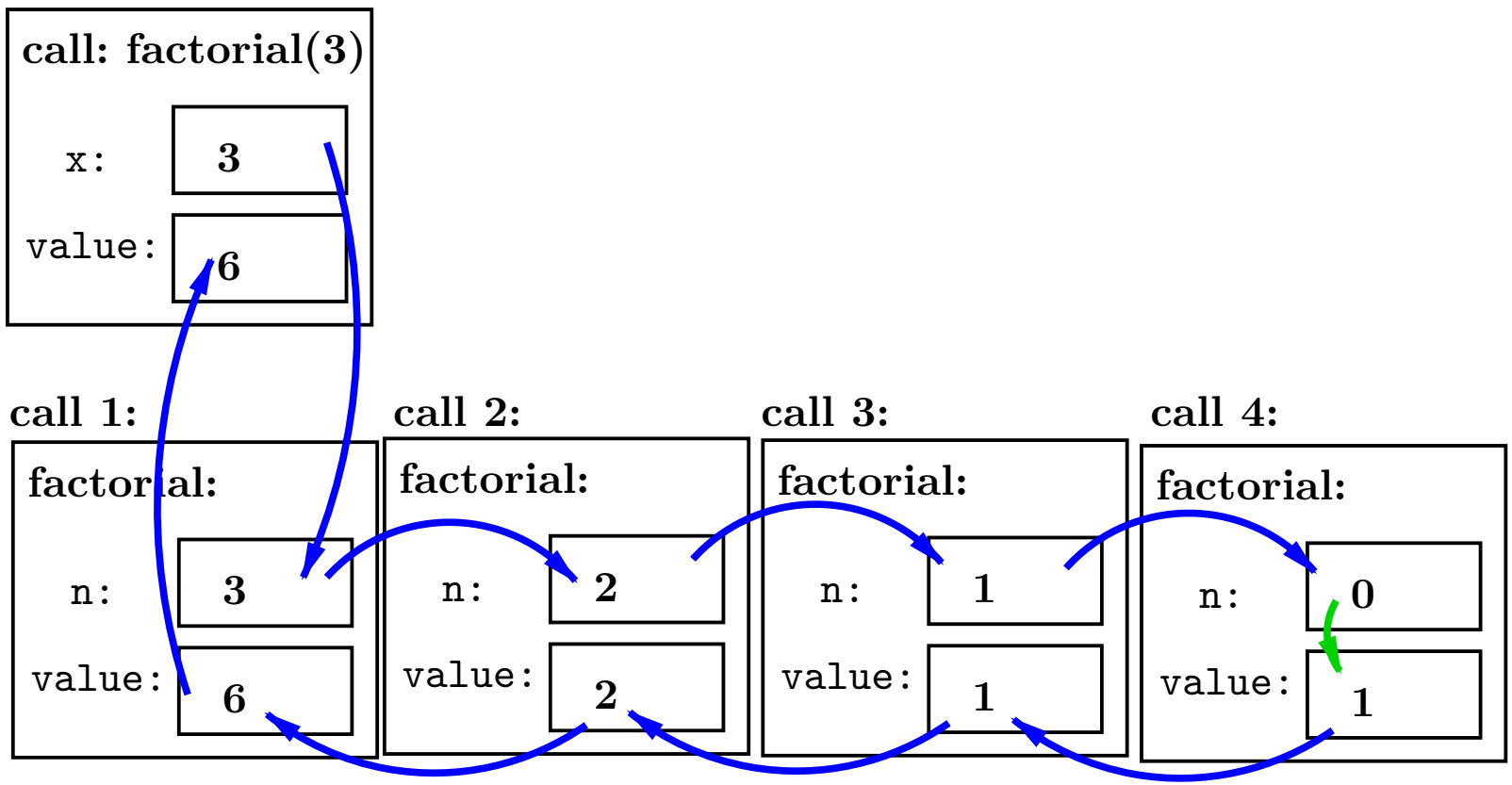
$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \text{ for } n > 1 \end{aligned}$$

```
1  public class K3B29E_Factorial {
2
3     static long factorial (int n) {
4        if (n == 0) return 1;
5        else return (n * factorial (n-1));
6     }
7
8     public static void main (String[] args) {
9       for (int  k=0; k<22; k++)
10         System.out.println(
11          "Factorial of " + k + "␣is: " + factorial (k) );
12    }
13 }
```

(The values up to **20! = 2.432.902.008.176.640.000** are correct, **21!** is too large for **`long`**.)

```
1    static long factorial (int n) {
2       if (n == 0) return 1;
3       else return (n * factorial (n-1));
4    }
```

Evaluation: “recursive descend”



```
1  public class K3B30E_Fibonacci {
2
3     static long fib (int n)  {
4        if (n==0) return 0;
5        else
6           if (n==1) return 1;
7           else return ( fib (n-1) + fib (n-2) );
8     }
9
10    public static void main(String[] args) {
11       int k= Integer.parseInt(args[0]);
12       System.out.println(
13              "Fibonacci␣(␣" + k + "␣)␣=␣" + fib (k) );
14    }
15 }
```

Effort of the recursive solution:

- ***fib*(40) = 102 334 155**
  (requires 267 914 295 calls, about 0.8s on PC)
- ***fib*(45) = 1 134 903 170**
  (requires 2 971 215 072 calls, about 8.4s)
- ***fib*(90) = 2 880 067 194 370 816 120**
  (requires 7 540 113 804 746 346 428 calls...)
  with 350 000 000 calls/s this takes about $\mathbf{2{,}1 \cdot 10^9}$s (about 66 years!)
- Conclusion: not every solution is a good (= efficient) solution...
- Problem:
  - non-linear recursion
  - repeated repetition of calls
    (especially ***fib*(0)**, ***fib*(1)**)

Fibonacci numbers can also be expressed via the 'golden ratio':

$$\mathit{fib}(n) = \frac{\Phi^n - \Psi^n}{\sqrt{5}}$$

with $\Phi = \frac{1+\sqrt{5}}{2} = 1.61803...$ and $\Psi = \frac{1-\sqrt{5}}{2} = -0.61803....$

Since $|\Psi| < 1$ we have $\frac{|\Psi|^n}{\sqrt{5}} < \frac{1}{\sqrt{5}} < \frac{1}{2}$

thus

$$|\mathit{fib}(n) - \frac{\Phi^n}{\sqrt{5}}| \leq \frac{1}{2}$$

and therefore:

$$\mathit{fib}(n) = \lfloor \frac{\Phi^n}{\sqrt{5}} + \frac{1}{2} \rfloor$$

(where $\lfloor x \rfloor$: greatest integer less or equal to x)

Fibonacchi numbers are thus growing exponentially:

$$\mathbf{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...}$$

Let $\boldsymbol{ra(n)}$ be the number of (recursive) calls of $\boldsymbol{fib(n)}$, then:

$$\boldsymbol{ra(n) = ra(n-1) + ra(n-2) + 1}$$

The number of calls grows similarly to the Fibonacci numbers themselves (even a little faster), i.e., exponentially.

Computation of Fibonacci numbers with “dynamic programming”

- partial results are stored ($\leadsto$ array)

```
1  public class K3B31E_Fibonacci_dynamic {
2
3     static long fib (int n)  {
4
5        long[] f = new long[n+1];
6
7        if (n <= 1) return n;
8        f[0] = 0;
9        f[1] = 1;
10       for (int i = 2; i <= n; i++ ) {
11          f[i] = f[i-1] + f[i-2];
12       }
13       return f[n];
14    }
15
16    public static void main(String[] args) {
17       int k= Integer.parseInt(args[0]);
18       System.out.println(
19              "Fibonacci ( " + k + " ) = " + fib (k) );
20    }
21 }
```

Reduction to partial results that are still needed:

- Only two old values are still used...

```
1  public class K3B32E_Fibonacci_iterative {
2
3     static long fib (int n)  {
4        if (n <= 1) return n;
5        long fib_0 = 1, fib_1 = 0, fib_2;
6        for (int i = 2; i <= n; i++ ) {
7           fib_2 = fib_1;
8           fib_1 = fib_0;
9           fib_0 = fib_1 + fib_2;
10       }
11       return fib_0;
12    }
13
14    public static void main(String[] args) {
15       int k= Integer.parseInt(args[0]);
16       System.out.println(
17              "Fibonacci ( " + k + " ) = " + fib (k) );
18    }
19 }
```

Recursion vs. iteration

- The example of the Fibonacci numbers shows that an ‘elegant’ algorithm is not necessarily efficient.
- The efficiency of recursive algorithms often depends on how well the compiler can optimize them.
- It is always possible to reformulate recursive algorithms as iterative algorithms,
  in ‘more complex’ cases this requires the data structure ‘stack’ (see later).

Predefined methods

- The Java Software Development Kit (SDK) includes a large number of predefined methods.
- J2SE 5.0 includes about 3000 methods in more than 3000 classes.
- Example: class **`Math`** (**`java.lang.Math`**), see https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

constants $e = 2.71828182846...$ and $\pi = 3.14159265359...$

```
1    public static final double E
2    public static final double PI
```

methods:

```
1    public static double ceil (double a)
2    public static double floor (double a)
3    public static double rint (double a)
4
5    public static long round (double a)
6    public static int round (float a)
```

- Predefined methods can be directly used without declaration.
- **`Math`** belongs to the package **`java.lang`** (see later) and thus does not have to be imported.
- All methods in **`Math`** are declared as **`static`** (see later), they are called using **`classname.methodname`**, for example **`Math.ceil (3.14);`**

```
1 public class MathCall {
2
3   public static void main (String[] args) {
4
5   System.out.println( "square root of (e * pi) : "
6           +   Math.sqrt ( Math.E * Math.PI ) );
7    }
8 }
```

More methods from **`java.lang.Math`**:

```
1  public static double sin (double angle) // sine
2  public static double cos (double angle) // cosine
3  public static double tan (double angle) // tangent
4
5  public static double exp (double x) // e^x
6  public static double log (double x) // logarithm ln x
7  public static double pow (double x, double y) // x^y
8
9  public static double random () // random number between 0.0 and 1.0
10
11 public static double sqrt (double x) // square root of x
```

`log`, **`pow`** and **`sqrt`** yield **`NaN`** in the case of an error.

```
1     public static double abs (double x)
2     public static float abs (float x)
3     public static long abs (long x)
4     public static int abs (int x)
5
6     public static double max (double x, double y)
7     public static float max (float x, float y)
8     public static long max (long x, long y)
9     public static int max (int x, int y)
10
11    public static double min (double x, double y)
12    public static float min (float x, float y)
13    public static long min (long x, long y)
14    public static int min (int x, int y)
15 ...
16 ...
```

abs, **max**, **min** and **round** are examples for *overloaded methods*.

Method Overloading

```
1  public class K3B33E_Overloading {
2     public static void main (String[] args) {
3
4     double db = -6.12345678908642;
5     float fl  = -6.1234567F;
6     long lg   = -12345678908642L;
7     int in    = -12345678;
8
9     System.out.println(
10          "abs (db) = " + Math.abs ( db )
11      + "\nabs (fl) = " + Math.abs ( fl )
12      + "\nabs (lg) = " + Math.abs ( lg )
13      + "\nabs (in) = " + Math.abs ( in ) );
14    }
15 }
```

- An operator is *overloaded* when it has multiple meanings for the same syntax, i.e., when it can be used for different tasks.
- "-" as an example for an overloaded operator in elementary mathematics:

	unary sign operator	binary subtraction operator
natural numbers	-	-
integers	-	-
rational numbers	-	-
real numbers	-	-

- Some languages (e.g., **C++**) allow to define operators for new data types ("operator overloading")
- In **Java** "operator overloading" is not possible.

- A method is overloaded when variants of this method with the same name exist, that perform a different task and thus have a different semantics. (Examples: **`abs`**, **`max`**, **`min`** in **`java.lang.Math`**).
- Variants of overloaded methods have the same name, but must have different parameter lists,
  i.e., a different number of parameters and/or different parameter types.
- Thus the *signature* of a method is the combination of return type, method name and parameter list.
- The declaration of two methods whose signatures differ only in the return type yields a compiler error message.
- Overloaded methods are typical for object-oriented programming languages, since here the operators for the different classes of objects are implemented as methods (see later).

Local **`static`** methods can have the same signature as global **`static`** methods, without overwriting them.

```
1  public class K3B34E_local_Max {
2
3    static int max (int x, int y)  {
4      return x > y ? x : y;
5    }
6
7    public static void main(String[] args) {
8      System.out.println (
9              "Maximum: " + max (3, 7)       + " (local method!)"
10        + "\nMaximum: " + Math.max (3, 7) + " (Math.max)"
11     );
12   }
13 }
```

The global **`static`** method can always be uniquely identified using the class name, e.g. **`max (3,7)`** vs. **`Math.max (3,7)`**.