Foundations of programming (continued)

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]);
      }
}
```

Loops:

We have already introduced the **`while`** loop.

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;
}
```