Foundations of programming (continued)

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

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

("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]);
      }
}
```

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;
}
```