Java - lecture notes

with example programs from archive ***java_ examples***

Tomasz R. Werner

Warsaw, October 23, 2018 18:52

RULES: classes — 20 points, 'big' tests (two in the semester, during the lecture) — 50 points, 'small' tests (two or three in the semester, during the tutorials) — 30 points.
In order to pass the classes and be admitted to the exam, the sum must be at least 50.
Students with the score at least 70 do not need to take the exam and their final note will be calculated as stated below.
For those who *will* take the exam, its result will be scaled to the range $[0, 100]$. A score below 35 means a failure... If the score for the exam is at least 35, the total score will be calculated as half of the sum of the results obtained for the classes, tests and for the exam. The final note will be then:
GRADES:

$[50 - 59) \Rightarrow 3.0 \quad [60 - 69) \Rightarrow 3.5 \quad [70 - 79) \Rightarrow 4.0 \quad [80 - 89) \Rightarrow 4.5 \quad [90 - 100) \Rightarrow 5.0$

Contents

Section 1

General introduction

1.1 Computers and programming languages

1. Hardware: processors, memory, caches, disks and the like...
2. Operating system: "system software that manages computer hardware and software resources and provides common services for computer programs.[...] For hardware functions such as input/output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware" (Wikipedia). Operating system interacts with file system, launches other programs assigning resources to them, interacts with the user, detects events (movement of the mouse, clicks, pressing a key, etc.).
   On desktop computers, the dominant operating system is Microsoft Windows, followed by Apple's macOS and various distributions of Linux. On smartphones and tablets, Google's Android is the leader, followed by Apple's iOS. Linux distributions are dominant in the server and super-computing sectors.
   According to Stack Overflow, among professional developers 50% uses Windows, 27% uses macOS and 23% a Linux distribution.
3. Bits and bytes, hexadecimal and octal system.
   - Bit: — the smallest unit of information; only two values possible (0 or 1, 'up' or 'down', 'black' or 'white',...). By convention, we use 0 and 1 picture, because it allows us to interpret sequences of bits as numbers in the binary system.
   - Byte: — a sequence of 8 bits. There are $2^8 = 256$ such sequences possible; interpreted as numbers in the binary system, they assume values in the closed interval $[0, 255]$.
   - hexadecimal notation: — very practical, because four bits can be in exactly $2^4 = 16$ combinations, so there is a one-to-one correspondence between any sequence of four bits and a single digit in hexadecimal notation (0-9, A, B, C, D, E, F — hex-digits A-F can also be written in lowercase). It follows that there is a one-to-one correspondence between all possible bytes and all sequences of exactly two hex-digits. For example $255_{10} = 11111111_2 = FF_{16}$ and $46_{10} = 00101110_2 = 2E_{16}$.
4. Processor: — is what 'does the job' in the computer. Important components of a processor are **registers** — information, in the form of sequences of bits, can be stored there and manipulated by **instructions**, which themselves are also expressed as sequences of bits. There is a limited number of instructions that any given processor 'understands' (its **instruction set**). These are elementary instructions, like 'set a register X to a value', 'copy data from a memory location M to the register Y' (or *vice versa*), 'add (subtract, multiply, divide) the values of two registers, X and Y, and place the result in register Z', and so on. It is important to realize that no matter what programming language we use, our program must be ultimately somehow transformed into the form of a long sequence of these simple, elementary instructions (i.e., into the form of the **machine code** or **executable**).
5. Program: — a sequence of instructions (perhaps from many source files) in any language which, after transforming into machine code, performs an indicated task.

6. Compiler: — a program which reads one or more source files (just text files) and transforms it into machine code which can be passed to the processor. Sometimes the result is not an executable, but some intermediate form, which is then compiled 'to the end' and passed to the processor by another, additional, program. This, for example, is the case for Java, as we will see. Some languages are not compiled — there is a program, called **interpreter**, which reads the source file line by line and transforms it into machine code 'on the fly' in memory, without creating separate executable files (this is the case, for example, for the Python programming language).
7. Programming languages:
   - Low-level: machine code, assembly language.
   - High-level: interpreted or compiled. There are many attempts to categorize languages, but it seems not to be possible to do it. Broadly speaking, we can divide languages into categories like
     - imperative: procedural, object-oriented;
     - declarative: relational, functional, logic.

   Currently, the most popular programming languages are Java, C, C++, Python and Java Script. On the other hand, popularity of languages depends on the subject domain; for scientific and engineering calculations, the Fortran and Mathematica programming languages would be closer to the top of this list, while for statistical applications the language called R is extremely useful and popular.
8. Algorithm: — "an unambiguous specification of how to solve a class of problems" (Wikipedia). Therefore, an algorithm is a kind of a recipe which tells us how to obtain the result we want in finite number of steps. For example: given a collection of, say, 3 million, numbers, how to sort them in ascending order? Or, given two whole positive numbers, how to find their greatest common divisor (there is a famous solution of this problem given by Euclid in his *Elements*). Generally, when writing a program or a part of it, what we have to consider first is just an algorithm which should be used to achieve our goal correctly and efficiently.
   The word *algorithm* has been derived from the name of a IX century Persian scholar Muḥammad ibn Mūsā al-Khwārizmī and Greek word αριϑμός (which means number). [By the way, the term *algebra* comes form the Arabic word *al-jabr*, appearing in the title of al-Khwārizmī's main work.]

1.2 What is Java?

Java — high level imperative programming language, object-oriented with some elements of functional programming. In the design of Java, emphasis has been put on

1. platform independence;
2. simplicity;
3. safety (no direct access to memory, as in C/C++, garbage collector, managing security issues, etc);
4. efficiency;
5. very rich standard library.

Some features of Java:

1. Designed with commercial use in mind by Sun (James Gosling, mid-nineties).
2. Compiled to platform independent byte code.

3. Executed by (platform *dependent*) JVM — Java Virtual Machine, which interprets byte code, transforms it into machine code (dynamically, without storing it on disk) which is passed to the processor(s).
4. Simple syntax very close to that of C/C++.
5. Built-in (in the form of a platform independent standardized library):
    - graphics (building GUIs — graphical user interfaces);
    - multithreading (concurrency); ;
    - network programming;
    - working with data bases;
    - multimedia (processing images, sound);
    - support for various security issues;
    - support for microprogramming — for 'small' devices, mobile phones, etc.
    - handling various data formats, like XML, JSON, etc.;
    - ...and much, much more...

Installation: Oracle web page[1] — install something called JDK (Java Development Kit). It installs the JVM (allowing to run Java programs) but also various tools which allow the developer to create Java programs (in particular, the compiler).
Documentation — as one big ***zip*** file — can also be downloaded there, or it can be viewed online[2].

[1]https://www.oracle.com/technetwork/java/javase/downloads/index.html
[2]https://docs.oracle.com/javase/10/docs/api

Section 2

Compiling and running a Java program

- Program in Java is a collection of classes. (what *class* really means, we will learn shortly). Normally, each class is written in a separate (text) file with extension *.java*.
- Whatever we write must be in a class; there must be at least one class declared as **public** and containing public function **main**;
- Names are important: (public) class **Person** *must* be defined in file *Person.java*. Names of classes (and therefore of files containing their definition) should start with a capital letter; strictly speaking, it is not required, but we should always stick to this convention.

Let us look at a very simple example: a program which consists of only one class (and, consequently, one file) which contains nothing but the function **main**. The program just prints (i.e., writes to the screen) “Hello, World”.

Listing 1 AAC-HelloWorld/Hello.java

```java
/*
 *  Program Hello
 *  It prints "Hello, World"
 */

public class Hello { /* Entry class must be public
                        File name = class name!   */
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
    // signature of 'main' always like this
}
```

Some elements to note:

- We have to define classes — everything is in a class!
- Name of the (public) class = name of the file.
- All names are case sensitive (xy, XY and Xy are completely distinct names having nothing in common) — this may be not so obvious for Windows' users.
- There must be a (static) function **main**, with ‘signature’ as shown in the example, in the class which is the entry point to the whole application.
- Each class is compiled to a separate class-file with extension *class* (which contains something like machine code, but for JVM, not for a real processor).
- Comments (from **//** to the end of line, and from **/*** through ***/**).
- Each statements of the program must end with a semicolon and may be written in many lines: sequences of white spaces (including end of line characters) are treated as one space.
- Printing (**System.out.println**).

Having a source file(s), we have to compile it. The Java compiler is a program named *javac* (*javac.exe* on Windows). We invoke it passing, as arguments, one or more *.java* source files (or '*.java' to compile all *.java* files in the current directory). The compiler creates the so called **class files**: one for each class defined in our source files. The names of these files are the same as the names of the classes, but with extension *.class* instead of *.java*. They contain the so called **byte code** corresponding to classes. This is *not* the machine code for any real processor, but rather for a *virtual* processor which doesn't physically exist but is standarized and platform independent. Hence, it doesn't matter on which platform the process of compilation takes place — the resulting byte code can be run on any platform where Java is installed.

As the byte code is still not in the form of the machine code, we still need another program to run (execute) the compiled Java application. This program is called **JVM** — Java Virtual Machine — program which is named just *java* (*java.exe* on Windows). Invoking it, we pass, as the argument, the name of the class which is the entry point to our application (without any extension) — this class must contain the function **main**. The program reads the byte code, compiles it 'to the end' into the form of machine code appropriate for a given platform and executes it (without creating any additional files). Strictly speaking, JVM (or its sub-process called **JIT** — *just-in-time compilation*) compiles the byte code constantly 'on the fly'; it can dynamically detect 'bottle necks' of the program and optimize these parts of the code because it has access to dynamic run-time information (which is not the case for traditional compilers).

Continuing our example, the process of compiling and running our application might look like this

```
$ ls                             what's in the current directory?
Hello.java
$ javac Hello.java               we compile...
$ ls                             what do we have now?
Hello.class  Hello.java
$ java Hello                     we run the program
Hello, World                     and get its output
```

Section 3

Types, variables, literals

3.1 Primitive types

Any piece of data must have a **type**. In Java, the types that may correspond to named variables are called **primitive** (or fundamental) types. We may think of a **variable** of a primitive type as of a named piece of memory containing a single value of a well defined type. The type of a variable determines its length (number of bytes it occupies) and the way its contents is interpreted. In Java, only variables of primitive types can be created locally, on the stack — objects of all other types can only be created on the heap and never have names (identifiers). We will explain what stack and heap are shortly.

The primitive types are (number of bytes is given in parentheses):

- Numerical types:
  - integral types correspond to integer (whole) numbers. Their possible values belong to interval $[-2^{N-1}, 2^{N-1}-1]$, where N is the number of bits (one byte = 8 bits). The exception is **char** whose values are always interpreted as non-negative and belong to interval $[0, 2^{16}-1]$.
    - **byte** (1) — values in range $[-128, 127]$;
    - **short** (2) — values in range $[-32\,768, 32\,767]$;
    - **char** (2) — values in range $[0, 65\,535]$ interpreted as Unicode code points of characters (always non-negative);
    - **int** (4) — values in range $[-2\,147\,483\,648, 2\,147\,483\,647]$;
    - **long** (8) — values in an astronomical range $[-9\,223\,372\,036\,854\,775\,808, 9\,223\,372\,036\,854\,775\,807]$.
  - floating point types correspond to real numbers (with fractional parts). There are two such types with different ranges and precision.
    - **float** (4) — values in range $[\approx 1.4\cdot 10^{-45}, \approx 3.4\cdot 10^{+38}]$ positive or negative, with roughly 7 significant decimal digits – rarely used;
    - **double** (8) — values in range $[\approx 4.9\cdot 10^{-324}, \approx 1.8\cdot 10^{+308}]$ positive or negative, with roughly 16 significant decimal digits.
- Logical type **boolean** — has only two possible values: **true** and **false**. They are *not* convertible to numerical values, neither are numerical values convertible to **boolean** (as they are in many other languages);
- References to objects: — variables of these types hold as their values *addresses* of objects of the so called object types (not their values). In C/C++ such variables are called **pointers**.

Let us explain how integral values are represented in computer's memory. Consider a **byte**: it contains 8 bits, so we can represent it by a sequence of eight digits, each of which is 0 or 1:

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

The value it corresponds to is

$$-b_7\cdot 2^7 + b_6\cdot 2^6 + b_5\cdot 2^5 + b_4\cdot 2^4 + b_3\cdot 2^3 + b_2\cdot 2^2 + b_1\cdot 2^1 + b_0\cdot 2^0$$

As we can see, the term with the highest power of 2 is taken with minus sign. Therefore, to get the highest possible value of a byte, we should set this negative part to zero, and all remaining terms with coefficient 1

01111111

which is 127. Expressing groups of four bits as hexadecimal digits (see below) the same number is 7F. The smallest **byte** will have 1 at the negative part and all zeroes at the positive ones

10000000

what in hexadecimal notation would be 80 (it corresponds to -128). This reasoning applies to the remaining integral types, except **char** — here we count all terms with plus sign.

3.2 Object types

Besides primitive types, there are also the so called **object types**. They are defined by the user, although many such types are already defined by implementers of the standard library and we can use them in our programs. Names of the object types should always start with an upper case letter. Their objects (variables) cannot be created locally on the stack and never have names. They are created on the heap and are automatically removed from memory when not needed anymore. We have access to such objects only through references (pointers) to them which physically contain only their addresses, not values. There is a special process, called **garbage collector**, which detects object on the heap which are not referenced anymore by any reference variable in the program and removes these unnecessary objects from memory. Object types are defined by **classes** which we will cover in the following chapters. Generally, they describe objects more complicated than just a single number: the object may contain several numbers, strings, etc.; moreover, the class also defines operation that may act upon all this data.

3.3 Variables and literals

Variable may be understood as a named region of memory storing a value of a specified type. Each variable, before it can be used, has to be created (declared and defined) — in declaration we specify its name and type. It is also recommended to assign a value to any newly created variable (initialize it). Java compiler will not allow us to refer to the value of a variable until it can see an assignment of a value to this variable. When assigning a value to a variable, we can use the value of any expression yielding a value of an appropriate type; in the simplest case this may be just a value specified literally. A number written without a decimal dot is understood as being of type **int**.

```
int a = 7;
int b = a + 5;
```

We can add a letter ’L’ (or lower-case ’l’) at the and if we want the compiler to treat it as a **long**

```
long m = 101L, n = 2147483648L;
```

(note that 2147483648 without the ’L’, would be treated as an **int**, but that would be illegal, because it is too big for an **int**!). Literal integers may also be written in octal (0 at the beginning), hexadecimal (0x at the beginning) or binary (0b at the beginning) form:

```
int a = 077, b = 0xFF, c = 0b1101;
```

(the a is 63, b is 15 and c is 13). Hexadecimal notation is especially convenient, because there are 16 hexadecimal digits (0-9, A-F) and exactly 16 possible values of any four-bit group of bits. Therefore, one byte can always be described by two hexadecimal digits and *vice versa* — any two hexadecimal digits describe uniquely one byte. For example, the greatest **short** has representation

0111 1111 1111 1111

(see above) which is 0b0111111111111111 or 0x7FFF, while the smallest

1000 0000 0000 0000

which is 0b1000000000000000 or 0x8000.

A number written literally, but with a decimal point is understood as a **double**

```
double x = 1.5;
double y = x + 0.75;
```

We can add a letter 'F' (or 'f') at the end if we want the compiler to treat a literal as a **float**

```
float x = 1.5F;
```

Floating point numbers can also be written in the so called **scientific notation**. In this notation we have a number (possibly with a decimal dot), then the letter 'E' (lower- or uppercase) and then a integral number indicating the power of 10. For example 1.25E2 means $1.25 \cdot 10^2$ (i.e., 125), while 1E-7 means $1 \cdot 10^{-7}$ (0.0000001).

Literal of type **char** may be written as a single character in apostrophes

```
char c = 'A';
```

As **char** is a numerical type, the value will be a number, namely the Unicode code of a given character (which for 'A' is 65). As **char** occupies two bytes and is treated as a non-negative number, its values are in the range $[0, 2^{16} - 1] = [0, 65\,535]$ which is enough for all letters in almost all languages to be represented. Characters that are not present on our keyboard may be entered like this

```
char c = '\u03B1';
```

by writing, after a backslash and letter the 'u', the Unicode code of a character in hexadecimal notation. In this case, the code 0x03B1 corresponds to the Greek letter α. There are also some special characters that cannot be entered from the keyboard, like CR (carriage return), LF (line feed), etc. They can be specified using the Unicode notation, as above, but they also correspond to special symbols: a backslash and a letter or another symbol

- \a – (BEL) alert;
- \b – (BS) backspace;
- \f – (FF) formfeed (new page);
- \n – (LF) new line (linefeed);
- \r – (CR) carriage return;
- \t – (HT) horizontal tab;
- \v – (VT) vertical tab;
- \' – apostrophe;
- \" – quotation mark;
- \\ – backslash;

Some examples of literals can be found in the program below:

Listing 2 AAG-Literals/Literals.java

```java
public class Literals {
    public static void main(String[] args) {
        System.out.println(22);          // decimal
        System.out.println(022);         // octal
        System.out.println(0x22);        // hexadecimal
        System.out.println(0b1001);      // binary
        System.out.println(22.22);       // double
        System.out.println(2.22e-1);     // "scientific"
        System.out.println(1/3 );        // this is 0 !
        System.out.println(1/3.);        // one third
        System.out.println(1/3D);        // 3D -> double
        System.out.println(2147483648L);// long
        System.out.println(2147483647  + 1 ); // ooops!
        System.out.println(2147483647L + 1 );
        System.out.println('A');         // char
        System.out.println('A'+2);       // char
        System.out.println((char)('A'+2));
        System.out.println('\u03a3');    // also char
        System.out.println("Hello, World");
        System.out.println("\u017b\u00F3\u0142w");
        System.out.println("number = " +  2+2);
        System.out.println("number = " + (2+2));
        System.out.println(false);
        System.out.println(2*3 == 6);
        System.out.println("\"TAB\"s and 'NL'\n"+
                           "a\tb\tc\te\tf\n\tg\th\ti\tj");
        System.out.println("C:\\Program Files\\java");
    }
}
```

and examples of creating and using variables in the program below:

Listing 3 AAL-Variables/Variables.java

```java
public class Variables {
    public static void main(String[] args) {
        int    ifour = 4;
        double xhalf = 0.5;
        double four  = ifour;     // automatic conversion
        // int badFour = 4.0;     // WRONG
        int k = 1, m, n = k + 3;
        m = 2;
        final double two = xhalf*ifour;
        // two = two + 2;         // WRONG
        boolean b = true;
        if (b) System.out.println(
                "k=" + k + ", m = " + m + ", n = " + n +
                "\nSum by 4 is equal to " + (k+m+n)/ifour);
        String john,            // does string john exist?
               mary = "Mary";
        john = "John";
        System.out.println(john + " and " + mary);
        john = mary;
          // reference copied, string "John" lost!
        System.out.println(john + " and " + mary);
    }
}
```

Note that variables john and mary are not objects of type **String** — they are references (pointers) whose values are *addresses* of such objects! Therefore, `john=mary` means that we copy the address of the object corresponding to `"Mary"` to the variable john; from now both john and mary refer to exactly the same object somewhere in memory. Object which was before referred to by the variable john is now lost (because we have lost its address) and can be garbage collected.

3.4 Conversions

Sometimes a value of one type should be used as a value of another type. Changing the type of a value is called **conversion** or **casting**. Of course, it is impossible to change the type of a *variable*: conversions always involve *values*. For example, in

```
int a = 7;
double x = a + 1;
```

the value of the right-hand side in the second line is of type **int** and a **double** is needed to initialize the variable x; however, the compiler will silently convert **int** value to the corresponding **double** value. Such conversions, performed automatically by the compiler, are called **implicit** conversions. Generally, they will be performed if they don't lead to a loss of information. Conversion in the opposite direction

```
double x = 7.7;
int    a = x;    // WRONG
```

will *not* be performed; the snippet above wouldn't be even compiled. This is because an **int** occupies four bytes and has no fractional part, while **double**s have fractional part and occupy eight bytes. Hence, conversion from **double** to **int** would lead to inevitable loss of information. We can, however, enforce the compiler to perform such conversions (taking the responsibility for possible consequences). We do it by specifying, in parentheses, name of the type we want to convert to:

```
double x = 7.7;
int    a = (int)x;   // now OK
```

Of course, after conversion, a will be exactly 7, as there is no way for an **int** to have a fractional part.

Note also that this conversion does *not* affect the variable x as such: its type is still **double** and its value is still 7.7.

The exact rules of conversions are more complicated, but general principle is that conversions from "narrow" types to "wider" are performed implicitly (**byte**,**char**,**short** →**int**, **int**,**float**→**double**, etc.), while conversions in the opposite direction must be explicit.

Section 4

Quick introduction to I/O

We will show here, how data can be read from the console (standard input by default is connected to the keyboard) and from a graphical window. First, let us consider a console. The simplest way to read from the standard input is by creating an object of class **Scanner**, as shown in the example below. The **import** statements at the beginning are not necessary, but without them, one would have to use full, qualified names of classes, e.g., **java.util.Scanner** instead of just **Scanner**.

There is a little problem with reading values of floating-point types: whether to use a dot or a comma as the decimal separator. This depends on the locale; for example, for Polish locale a comma should be used, for an American one – a dot. The current locale may be changed, as explained below in comments.

Listing 4 AAI-ReadKbd/ReadKbd.java

```java
import java.util.Scanner;
import java.util.Locale;  // see below

public class ReadKbd {
    public static void main(String[] args) {

        // If locale is Polish, floating point numbers
        // have to be entered with  c o m m a  as the
        // decimal separator. Locale can be changed to,
        // e.g., American, by uncommenting the line below:
        //    Locale.setDefault(Locale.US);
        // (then the dot is is used as decimal separator).
        // When reading data, any nonempty sequence of
        // white characters is treated as a separator.

        Locale.setDefault(Locale.US);
        Scanner scan = new Scanner(System.in);

        System.out.println("Enter an int, a string " +
                           "and two double's");
        int    k = scan.nextInt();
        String s = scan.next();
        double x = scan.nextDouble();
        double y = scan.nextDouble();

        System.out.println("\nEntered data:\n\nint     = " +
                k + "\nString  = " + s  + "\ndouble1 = " +
                x + "\ndouble2 = " + y + "\n");
        scan.close();
    }
}
```

We used here **nextInt** (to read an **int**), **nextDouble** (to read a **double**), and **next** (to read a **String**): there are analogous functions **nextByte**, **nextShort**, **nextLong**, **nextBigInteger**, **nextFloat**, **nextBigDecimal**, and **nextBoolean**. Note: when all data has been read, the scanner should be closed (by invoking `scan.close()`, as shown in the example).

In order to read data from a graphical widget, or to display a message (string), one can use static functions from class **JOptionPane**, as shown below (the first argument of these functions is **null** for reasons we will learn about later). Note that **showInputDialog** returns a string (strictly speaking the reference to a string); if we know that this string represents a number and we want this number as an **int** or a **double**, we have to parse this string to get numbers using **Integer.parseInt**, or **Double.parseDouble**):

Listing 5 AAJ-ReadGraph/ReadGraph.java

```java
import javax.swing.JOptionPane;

public class ReadGraph {
    public static void main(String[] args) {

          // simple form of showInputDialog...
        String s = JOptionPane.showInputDialog(
                                null,"Enter an integer");
          // parsing string to get an int
        int k = Integer.parseInt(s);

          // parsing string to get a double
        s = JOptionPane.showInputDialog(
                         null,"Enter a double");
        double x = Double.parseDouble(s);

        s = JOptionPane.showInputDialog(
                         null,"Enter a string (spaces OK)");
        JOptionPane.showMessageDialog(null,
                "Data entered: int=" + k + ", double=" +
                x + ", string=\"" + s + "\"");
        System.exit(0); // kills JVM
    }
}
```

Section 5

Instructions and operators

5.1 Operators

Operators are usually expressed by symbols (like +, *, etc.) and represent operations to be performed on their **operands** (arguments). Most operators are *binary*, i.e., they operate on two operands; some operators act on one operand only (they are called *unary* operators). Finally, there is one *ternary* operator.

5.1.1 Assignment operator

Assignment operator

```
b = expr;
```

evaluates the value of the right-hand side and stores the result in a variable appearing on the left-hand side. The type of the value of expr must be the same as the type of b (or be implicitly convertible to this type). It is important to remember that the whole expression a=b has a type and a value: the type of this expression is the type of the left-hand operand, and its value is equal to the value of the left-hand operand *after* the assignment. Therefore, after

```
int a, b = 1, c;
c = (a = b+1) + 1;
```

values of a, b and c will be 2, 1, 3.

Note that addition (subtraction, etc.) does *not* modify values of operands — it just yields a new *temporary value* and we have to do something with this value: print it, assign it to a variable, use it in an expression, etc. For example, after

```
int a = 5, b = 1;
b = 2*(a + 1) + b;
```

the value of a is still 5; when calculating a+1 we got a value (in this case 6) which is subsequently multiplied by 2 and added the current value of b, i.e., 1. We haven't assigned any new value to a, so it remains as it was. However, the value of the whole expression on the right-hand side of the assignment (13) has been assigned to b (erasing its previous value), so b *is* modified here.
There is a special form of assignment, the so called **compound assignment operator**. It has the form a @= b, where @ stands for any binary operator, like +, *, %, >>, etc.) and is (almost) equivalent to a = a @ b. For example a += 5 would mean *increment* a *by 5*, and a /= 2 — *divide* a *by 2.*

Incrementing and decrementing integral values by 1 is so often used that it has a special syntax. If a is a variable of an integral type, then ++a and a++ cause a to be incremented by 1 (with - instead of + — decremented). However, there is a difference between these two forms. *Pre*incrementation or decrementation (++a or --a) are 'immediate', while *post*incrementation or decrementation takes place *after* the whole expression has been evaluated. For example, after

```
int a = 5, b = 1, c;
c = ++a + b--;
```

c will be 7, as on the right-hand side a had been incremented before it was used to evaluate the whole expression (so its new value, 6, was used) while b was still 1 during evaluation and was decremented only *after* the assignment. However, after

```
int a = 5, b = 1, c;
c = ++a + --b;
```

c will be 6, as a was incremented and b was decremented before evaluation. In both cases, after the assignment to c, values of a and b are 6 and 0, respectively.

5.1.2 Arithmetic operators

Well known examples of arithmetic operators include addition (a+b), multiplication (a*b), division (a/b), and subtraction (a-b). As we can see, binary operators are placed *between* operands they act on. Less known is the remainder (also called modulus operator) operator: a%b, yields the remainder after the division of a by b, so, for example, 20%7 is 6, as 20 is 2*7+6.

Important: arithmetic operations on values of 'small' integral types are always performed with their values converted to **int**, and the result is therefore of type **int**. For example, two values of type **byte** added together give **int**, *not* **byte**:

```
byte a = 1, b = 2;
byte c = a + b;      // WRONG
```

will not compile, because the result of a + b is of type **int**, so explicit casting would be required

```
byte a = 1, b = 2;
byte c = (byte)(a + b);   // now OK
```

Also remember that if two operands are of integral type, so is the result. Therefore, 1/2 is exactly 0, and 7/2 is exactly 3!. When you divide two integers values, the result is alway truncated (its fractional part removed) towards zero — for example 7/3 and −7/3 are 2 and −2, respectively.

This convention is related to the remainder operator. By definition, the following equivalence

$$(a/b) * b + (a\%b) \equiv a$$

should always hold. As division always truncates towards 0, it follows that taking the remainder yields remainder of absolute values of operands with the sign of the *first* operand; for example

```
System.out.println(" 7 /  2 = " + ( 7 /  2 ));
System.out.println(" 7 / -2 = " + ( 7 / -2 ));
System.out.println("-7 /  2 = " + (-7 /  2 ));
System.out.println("-7 / -2 = " + (-7 / -2 ));
System.out.println(" 7 %  2 = " + ( 7 %  2 ));
System.out.println(" 7 % -2 = " + ( 7 % -2 ));
System.out.println("-7 %  2 = " + (-7 %  2 ));
System.out.println("-7 % -2 = " + (-7 % -2 ));
```

prints

```
 7 /  2 = 3
 7 / -2 = -3
-7 /  2 = -3
-7 / -2 = 3
 7 %  2 = 1
 7 % -2 = 1
-7 %  2 = -1
-7 % -2 = -1
```

(instead of remembering these rule, it is better not to use negative numbers in remainder operations.)

5.1.3 Conditional and relational operators

Relational operators yield Boolean values. Comparing values (of various types) we do get true or false, i.e., a Boolean value. For example

```
a  < b   // is a smaller than b ?
a  > b   // is a bigger  than b ?
a <= b   // is a smaller or equal to b ?
a >= b   // is a bigger or equal to b ?
a == b   // are a and b equal ?
a != b   // are a and b different ?
```

Logical values may be combined: the logical conjunction (AND) is denoted by && and alternative (OR) by ||. An exclamation mark denotes negation (NOT); for example

```
  a <= b && b <= c
  b  < a || b >  c
!(a <= b || b <= c)
```

The first condition corresponds to checking if b belongs to the $[a, c]$ interval, while the second if b is outside this interval. The third condition is just a negation of the first, so is equivalent to the second.

Very important: && and || operators are **short-circuited** (this feature is also known as *McCarthy evaluation*) — if, after evaluation of the first term, the result is already known, the second term will *not* be evaluated (and this is guaranteed). Therefore, if an expression to be evaluated is of the form

```
expr1 && expr2
```

then if expr1 is false, then the result is already known (must be false) and expr2 will *not* be evaluated at all. Similarly, for

```
expr1 || expr2
```

if expr1 is true, then the result must be true and expr2 will not be evaluated. For example, if a and b are integers, an expression like

```
if ( b != 0 && a/b > 5) ...
```

will never lead to division by zero: if b *is* zero, the condition b != 0 is false, the whole condition must be therefore false and division by b will not be even tried. Note that changing the order

```
if ( a/b > 5 && b != 0 ) ...
```

could result in divide-by-zero error!

In rare situations, we *do* want both operands of an OR or AND operator to be evaluated regardless of the result of the evaluation of the first operand. In these cases, we can use operators | and & (single, not doubled). Then both operands are always evaluated. Note, that these symbols stand for logical AND and OR only if their operands are themselves of type **boolean**. If operands are integer numbers, then the same symbols mean something different: they denote bit-wise AND and bit-wise OR operators which result in integer values (see the next section).

There is also the so called 'exclusive OR' (XOR) operator, denoted by ^. If expr1 and expr2 have values of type **boolean**, then the expression

```
expr1 ^ expr2
```

is true if, and only if, the values of expr1 and expr2 are *different*, i.e., **true** and **false** or **false** and **true**. For example, if x is a **double**, after

```
boolean b = (x >=2 && x <= 5) ^ (x >=3 && x <= 7);
```

b will be true if $x \in [2,5] \cup [3,7]$ but $x \notin [2,5] \cap [3,7]$, i.e., when x belongs to the symmetric difference of the two intervals (their sum but without their intersection).

Listing 6 ABY-RelOps/RelOps.java

```java
public class RelOps {
    public static void main (String[] args) {
        int a = 1, b = 8, d = 8;
        System.out.println(
                "Is d in [a,b]: " +
                ( a <= d && d <= b ));
        System.out.println(
                "Is d in (a,b): " +
                ( a <  d && d <  b ));
        System.out.println(
                "Is d outside (a,b): " +
                ( d <  a || d >  b ));
        System.out.println(
                "Is d outside (a,b): " +
                ( !(d >=  a || d <=  b) ));
        System.out.println(
                "Is d == b:  " + (d == b));
        System.out.println(
                "Is d != b:  " + (d != b));
    }
}
```

The **conditional expression** is the only *ternary* operator, i.e., an operator with three operands. It looks like this

```
cond ? expr1 : expr2
```

Here cond is an expression of type **boolean**. If it is **true**, then the value of expr1 will become the value of the whole expression and expr2 will not be evaluated; if cond is **false**, then the value of expr2 becomes the value of the whole expression and expr1 is not evaluated. The simplest example would be (a and b are **int**s):

```java
int mx = a > b ? a : b;
```

which initializes mx with the bigger of a and b values. The ternary operator resembles the `if...else if` construct, but is *not* equivalent! In particular, expression `b ? e1 : e2` has a value, while `if...else if` has not. For example,

```java
a > b ? System.out.print("a") : System.out.print("b"); // NO!!!
```

doesn't make sense, because **print** has no value.

Another example:

Listing 7 ACB-CondOp/Largest.java

```java
import java.util.Scanner;
import java.util.Locale;

public class Largest {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        scan.useLocale(Locale.US);

        System.out.print(
            "Enter three numbers (with " +
            "DOT as dec. separator!) -> ");

        double a = scan.nextDouble();
        double b = scan.nextDouble();
        double c = scan.nextDouble();

        double largest = b >       a ? b :       a;
        largest        = c > largest ? c : largest;

        double smallest = b <        a  ? b :        a;
        smallest        = c < smallest ? c : smallest;

        System.out.println("Number " + largest + " is " +
                           "the largest and " + smallest +
                           " is the smallest");
    }
}
```

5.1.4 Bit-wise operators

We can operate on variables of integral types (mainly **int**) treating them as "buckets" of single bits. In what follows, remember that operations of shifting, ANDing, ORing etc., that we will discuss, do not modify their arguments: they return *new* values that we have to handle in some way (display it, assign to a variables, and so on).

As we know, data in a variable is stored as a sequence of bits, conventionally represented by 0 and 1. In particular an **int** consists of 32 bits. We can interpret individual bits as coefficients at powers of 2: the rightmost (least significant) bit is the coefficient at 2^0, the next, from the right, at 2^1, the next at 2^2 and so on, to the last (i.e., the leftmost, most significant) bit which stands at 2^{31}.

Shifting

Shift operators act on values of integral types: they yield another value, which corresponds to the original one but with all bits shifted by a specified number of positions to the left or to the right.

Left shift (<<) moves the bit pattern to the left: bits on the left which go out of the variable are lost, bits which enter from the right are all 0. The value to be shifted is given as the left-hand operand while number of positions to shift – by the right-hand operand. For example (we use only eight bits to simplify notation, in reality there are 32 bits in an **int**):

```
    a             1 0 1 0 0 1 1 0
    a << 3        0 0 1 1 0 0 0 0
```

The *unsigned* right shift operator (>>>) does the same but in the opposite direction

```
    a             1 0 1 0 0 1 1 0
    a >>> 3       0 0 0 1 0 1 0 0
```

The *signed* right shift operator (>>) behaves in a similar way, but what comes in from the left is the sign bit: if the leftmost bit is 0, zeros will come in, if it is 1, these will be ones

```
    a             1 0 1 0 0 1 1 0
    a >> 3        1 1 1 1 0 1 0 0
    b             0 0 1 0 0 1 1 0
    b >> 3        0 0 0 0 0 1 0 0
```

ANDing, ORing, etc.

Bit-wise operations work similarly to logical ones (ANDing, ORing, XORing, negating) but operate on individual bits of their operands which must be of an integral type. For example, when ANDing two values with bit-wise AND operator (&) we will get a new value, where on each position there is 1 if and only if in both operands there were 1s on this position, and 0 otherwise

```
    a             1 0 1 0 0 1 0 0
    b             1 0 0 0 0 1 1 0
    a & b         1 0 0 0 0 1 0 0
```

For bit-wise OR operator (|), each bit of the result is 1 if there is at least one 1 at the corresponding position in operands, and 0 if both bits in the operands are 0

```
    a             1 0 1 0 0 1 0 0
    b             1 0 0 0 0 1 1 0
    a | b         1 0 1 0 0 1 1 0
```

The bit-wise XOR operator (`^`), sets a bit of the result to 1 if bits at the corresponding position in operands are different, and to 0 if they are the same

```
a              1 0 1 0 0 1 0 0
b              1 0 0 0 0 1 1 0
a ^ b          0 0 1 0 0 0 1 0
```

As can be expected, negating operator (`~`) just reverses the bits

```
 a             1 0 1 0 0 1 0 0
~a             0 1 0 1 1 0 1 1
```

Listing 8 BAA-Bits/Bits.java

```java
public class Bits {
    public static void main (String[] args) {
        int a = 0b11111111;    // 255 or 0xFF
        System.out.println("a = " + a);
        int b = 0x7F;          // 127
        System.out.println("b = " + b);

        a = 3; // 00...011
        System.out.println(a + " " + (a << 1) + " " +
                          (a << 2) + " " + (a << 3));
        a = -1;
        int firstByte  = a & 255;
        int secondByte = (a >> 8) & 0xFF;
        System.out.println("-1: " + secondByte + " " +
                                firstByte);
        a = 0b1001;
        b = 0b0101;
        System.out.println("AND: " + (a & b) + "; " +
                            "OR: " + (a | b) + "; " +
                           "XOR: " + (a ^ b) + "; " +
                           " ~a: " + (~a)    + "; " +
                           " ~b: " + (~b));
    }
}
```

5.1.5 Precedence and associativity

As we remember from school, various operators may have different precedence (priority). For example, in

```
d = a + b * c;
```

`b` could be the right operand of addition or the left operand of multiplication, but we know that multiplication has higher priority, so it is equivalent to

```
d = a + (b * c);
```

rather than

```
d = (a + b) * c;
```

Sometimes, however, it is not so obvious; is the statement

```
if ( a << 2 < 3) System.out.println("yes");
```

correct or not? If << has higher priority than <, than it is equivalent to

```
if ( (a << 2) < 3) System.out.println("yes");
```

what makes sense, but if < is 'stronger', than

```
if (a << (2 < 3)) System.out.println("yes"); // NO!!!
```

would be a mistake. If in doubt, just add parentheses to make your intentions clear. Still, it may be helpful just to know the precedences of operators. The following table shows them in decreasing order (from the 'strongest' to the 'weakest')

Table 1: Precedence of operators

Operator type	**Operators**
postfix	expr++ expr--
unary	++expr --expr +expr −expr ∼ !
multiplicative	* / %
additive	+ −
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	ˆ
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	cond ? expr1 : expr2
assignment	= += -= *= /= %= &= ˆ= |= <<= >>= >>>=

Another property of operators is **associativity**: it tells, whether in a series of operations with equal priority (without parentheses), they will be performed from left to right or from right to left. All binary operators are left-associative (from left to right) with the exception of assignment, which is right-associative; something like this

```
int a, b = 1, c;
c = a = b;
```

is correct, because it is equivalent to

```
int a, b = 1, c;
c = (a = b);
```

while

```
int a, b = 1, c;
(c = a) = b;
```

would not compile, because we assign uninitialized value of `a` to `c`.
The only ternary operator, the conditional operator, is also right-associative; this instruction

```
String s = a == b ? "equal" : a > b ? "a" : "b";
```

is correct, because it's equivalent to

```
String s = a == b ? "equal" : (a > b ? "a" : "b");
```

while

```
String s = (a == b ? "equal" : a > b) ? "a" : "b";
```

would be incorrect, as the expression to the left of the second '?' doesn't evaluate to a Boolean value.

Let us look at another example:

Listing 9 AAP-BasicOps/BasicOps.java

```java
public class BasicOps {
    public static void main(String[] args) {
        int x = 2, y = 2*x, z = x + y;
        // precedence, associativity
        System.out.println(x +  x + z  *  y - z  / 3); // 26
        System.out.println(x + (x + z) * (y - z) / 3); // -3

        // div and mod
        int u = 13;
        System.out.println(
                "u = " + u + ", u%7 = " + u%7 + ", u/7 = " +
                u/7 + ", 7*(u/7)+u%7 = " + (7*(u/7)+u%7));
        System.out.println();

        u %= 7; // compound assignment

        System.out.println("1. u="+u+", x="+x+", y="+y);
        x = ++u;
        System.out.println("2. u="+u+", x="+x+", y="+y);
        y = x--;
        System.out.println("3. u="+u+", x="+x+", y="+y);
        u = (x=--y);
        System.out.println("4. u="+u+", x="+x+", y="+y);
        u = x = y = (int)(9.99*y);
        System.out.println("5. u="+u+",x="+x+",y="+y);
        u = 6 << 2;
        x = u >> 1;
        y = 7 >> 1;
        System.out.println("6. u="+u+",x="+x+",y="+y);
        u = '\u0041';
        x = 'a';
        char c = (char)98;
        System.out.println("7. u="+u+",x="+x+",c="+c);
        System.out.println("Unicode of "+c+" is "+(int)c);
    }
}
```

5.2 Instructions

An instruction can be viewed as a single, in a given programming language, 'order' to be performed by the computer. It does *not* correspond to one instruction on the machine level; rather, to a sequence of such basic instructions. Instructions which do not have any value are sometimes called **statements**, as opposed to instructions (or parts of instructions) which do have values — these are called **expressions**.

Of course, the form and syntax of instructions may be different in different languages, but usually, at least for languages of the same 'family', there are more similarities than differences (this, in particular, applies to languages like C, C++, Java, C#, in which main syntactic constructs are almost identical).

Very often, when only *one* instruction is required by the syntax, we would like to use more; in such situations, we can use the so called **compound instruction**, i.e., a set of instructions enclosed in curly braces and therefore constituting a **block**. **Important:** local variables of primitive types (numbers, characters, reference variables, etc.) defined in a block are not visible — actually, they simply don't exist — outside of the block.

```
{           // block
   // ...
   int k = 7;
   // ...
}
// k does not exist here
```

5.2.1 Conditional statements

Conditional statements may look like this

```
if (cond) {
    // ...
}

// or

if (cond1) {
    // ... (code 1)
} else if (cond2) {
    // ... (code 2)
} else if (cond3) {
    // ... (code 3)
} else {
    // ... (code 4)
}
```

where cond are expressions whose values are of type boolean (i.e., true or false). The else if clauses are optional, as is the else clause; however, if they are used, the else clause must be the last. Conditions are checked in order, and for the first which evaluates to true, the corresponding block of code will be executed (subsequent blocks will be ignored). If the else clause is present, the corresponding block will be executed if none of the previous conditions is true. If there is only one instruction in the block corresponding to a condition, the curly braces are not obligatory (although recommended).

In the following example, we check if a given year is a leap year or not:

Listing 10 ABX-Ifs/LeapYear.java

```java
import java.util.Scanner;

public class LeapYear {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter a year (integer) -> ");
        int year = scan.nextInt();
        scan.close();

        boolean is_leap;

        if (year%400 == 0)
            is_leap = true;
        else if (year%100 == 0)
            is_leap = false;
        else if (year%4 == 0)
            is_leap = true;
        else
            is_leap = false;
          // ?: operator used below!
        System.out.println("Year " + year + " is "        +
                    (is_leap? "" : "not ") + "a leap year");
    }
}
```

5.2.2 Switch statement

The **switch statement** resembles a series of else if statements (but is not equivalent). It looks like this

```java
switch (expr) {
    case val1 :
        statement1;
        // ...
        statementN;
        break;
    case val2 :
        statement1;
```

```
            // ...
            statementN;
            // should we break?
        // ...
        default:
            statement1;
            // ...
            statementN;
            break;
    }
```

The value of expr must be of an integral type (but not long), it can be the reference to a String or a value of an enumeration. Symbols val stand for values with which the value of expr is compared (in the specified order). They have to be constants (not variables). If the value of expr is equal to any of vals, the code in the corresponding branch is executed and then execution continues (*falls through*) with all branches below; to avoid it, use break, which transfers control flow out of the switch block (sometimes this "falling through" may be just what we want, as in the example below). At the end of this example, we used a branch default, where we don't make any comparisons: this will be the branch selected if all other comparisons yield false. The default clause is optional and doesn't have to appear as the last. For example:

Listing 11 ACC-SimpleSwitch/SimpleSwitch.java

```java
import java.util.Scanner;

public class SimpleSwitch {
    public static void main (String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter an initial: ");
        char initial = scan.next().charAt(0);
        scan.close();

        switch (initial) {
            case 'A':
            case 'a':
                System.out.println("Amelia");
                break;
            case 'B':
            case 'b':
                System.out.println("Barbra");
                break;
            case 'C':
            case 'c':
                System.out.println("Cindy");
                break;
            case 'D':
            case 'd':
                System.out.println("Doris");
                break;
            default:
                System.out.println("Invalid input");
            }
        }
    }
```

The next example shows how to find a numerical value of a hexadecimal digit:

Listing 12 ACD-Switch/Switch.java

```java
import java.util.Scanner;

public class Switch {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        System.out.print(
            "Enter a single hex digit -> ");
        char ch = scan.next().charAt(0);
        scan.close();

          // toLower by hand...
        if (ch >= 'A' && ch <= 'Z')
            ch = (char)(ch + 'a' - 'A');

        int num;

        switch (ch) {
            case '0': case '1': case '2':
            case '3': case '4': case '5':
            case '6': case '7': case '8':
            case '9':
                num = ch - '0';
                break;
            case 'a': case 'b': case 'c':
            case 'd': case 'e': case 'f':
                num = 10 + ch - 'a';
                break;
            default:
                num = -1;
        }

        System.out.println("Character '" + ch + "' is " +
            (num >= 0 ? "" : "not ") + "a hex digit. "    +
            "Its numerical value: " + num);
    }
}
```

5.2.3 Loops

***while* loop** The simplest form of a loop is the so called **while-loop**. It looks like this

```
while (condition) {
      // ...
    if (cond1) break;
      // ...
    if (cond2) continue;
      // ...
}
```

where condition is an expression yielding Boolean value (true or false). The loop is executed as follows:

1. condition is evaluated, and if it is false, the flow of control jumps out of the loop;
2. if condition evaluates to true, the body of the loop (everything inside the block) is executed and then the flow of control goes back to item 1.

Inside the loop, you can (but not have to) use break and continue instructions. When break is executed, the flow of control goes out of the loop immediately. When continue is encountered, the current iteration of the loop is considered completed, and we go back to next iteration (so the condition is checked again).

Listing 13 AFA-While/Prime.java

```java
import java.util.Scanner;

public class Prime {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        while (true) {
            System.out.print(
                "Enter natural number (0 to exit) -> ");
            int n = scan.nextInt();

            if (n == 0) break;

            boolean prime = true;

            if (n == 1) {
                System.out.println("Number 1 is neither " +
                    "prime nor composite");
                continue;
            } else if (n > 2 && n%2 == 0) {
                prime = false;
            } else {
                int p = 3;
                while (p*p <= n) {
                    if (n%p == 0) {
                        prime = false;
                        break;
                    }
                    p += 2;
                }
            }
            System.out.println("Number " + n + " is " +
                      (prime ? "prime" : "composite"));
        }
        scan.close();
    }
}
```

One can assign names (labels) to loops, like this

```java
LAB1: for (int i = 0; i < size; ++i) {
    // ...
    LAB2: while (cond1) {
        // ...
        if (cond2) break LAB1;
        // ...
        while (cond3) {
            // ...
            if (cond4) continue LAB2;
            // ...
        }
    }
}
```

where LAB is any identifier. Named loops may be used with all types of loops (**while-loop**, **do-while-loop** and **for-loop** — see below). The advantage of naming loops, is that we can use the labels in **break** and **continue** instructions inside nested loops. Without them, both **break** and **continue** instructions always apply to the innermost loop only.

Listing 14 AFB-WhileBis/Prime.java

```java
import java.util.Scanner;

public class Prime {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        MAIN_LOOP:
        while (true) {
            System.out.print(
                "Enter natural number (0 to exit) -> ");
            int n = scan.nextInt();

            if (n == 0) break;

            if (n == 1) {
                System.out.println("Number 1 is neither " +
                    "prime nor composite");
                continue;

            } else if (n > 2 && n%2 == 0) {
                System.out.println("Number " + n +
                    ", being even, is composite");
                continue;

            } else {
                int p = 3;
                while (p*p <= n) {
                    if (n%p == 0) {
                        System.out.println("Number " + n +
                                         " is composite");
                        continue MAIN_LOOP;
                    }
                    p += 2;
                }
                System.out.println("Number " + n +
                                     " is prime");
            }
        }
        scan.close();
    }
}
```

***do-while* loop** Loops of this type are similar to **while-loops**, but checking a condition is performed *after* execution of the body of the loop.

```
do {
      // ...
    if (cond1) break;
      // ...
    if (cond2) continue;
      // ...
} while(condition);
```

Therefore,

1. body of the loop is executed;
2. the value of condition is evaluated; if it is **true**, flow of control goes to item 1, if it is **false**, the flow of control jumps out of the loop.

For example, in the program below, we roll two dice until two sixes are thrown:

Listing 15 AFE-DoWhileDice/Dice.java

```
public class Dice {
    public static void main (String[] args) {
        int a, b;
        do {
```

```java
            a = 1 + (int)(Math.random()*6);
            b = 1 + (int)(Math.random()*6);
            System.out.println("a=" + a + " b=" + b);
        } while (a != 6 || b != 6);
    }
}
```

A possible outcome of the program might be

```
a=5 b=6
a=4 b=3
a=6 b=5
a=1 b=2
a=1 b=3
a=1 b=6
a=3 b=6
a=1 b=3
a=2 b=1
a=4 b=5
a=6 b=6
```

for **loop**

For loops looks like this:

```java
for ( init ; condition ; incr ) {
      // ...
    if (cond1) break;
      // ...
    if (cond2) continue;
      // ...
}
```

where (below, by **expression** we mean anything that has a value)

- *init*: one declaration (possibly of several variables of the same type) or zero, one or several comma-separated expressions;
- *condition*: expression with a value of type **boolean**; if left empty – interpreted as **true**;
- *incr*: zero, one or several comma-separated expressions.

Any (or even all) of the three parts may be empty, but exactly two semicolons are always required.

In the following example there are nested for-loops: in each iteration of the main (outer) loop, the program executes two inner loops which print first some spaces and then some asterisks in such a way that a "pyramid" is formed with number of asterisks in the bottom line equal to a number read from input:

Listing 16 AFH-ForPyram/Stars.java

```java
import java.util.Scanner;

public class Stars {
    public static void main (String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter a positive odd number: ");
        int n = scan.nextInt();
        scan.close();

        for (int len=1, sp=n/2; len <= n; len+=2, --sp) {
            for (int i = 0; i < sp; ++i)
                System.out.print(" ");
            for (int i = 0; i < len; ++i)
                System.out.print("*");
            System.out.println();
        }
    }
}
```

For example, after inputing 9, the program prints

```
    *
   ***
  *****
 *******
*********
```

In the next example, we use while- and for-loops to calculate Euler's totient function $\varphi(n)$ (number of positive integers up to a given integer n that are relatively prime to n):

Listing 17 AFJ-ForWhileEuler/ForWhileEuler.java

```java
import java.util.Scanner;
/*
 * Finding and printing values of Euler's totient
 * function, i.e., number of positive integers up
 * to a given integer n that are relatively prime to n.
 */

public class ForWhileEuler {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        while (true) {
            System.out.print(
                "\nEnter a natural number (0 to exit) -> ");
            int n = scan.nextInt();

            if (n == 0) break;

            int count = 0;
            for (int p = 1; p <= n; ++p) {
                int a = n, b = p;
                  // Euclid's algorithm for GCD
                while (a != b) {
                    if (a > b) a -= b;
                    else       b -= a;
                }
                if (a == 1) ++count;
            }
            System.out.print("\u03c6(" + n + ") = " + count);
        }
    }
}
```

Section 6

Static functions

Classes usually contain, besides fields, constructors and methods (that we will cover later, when talking about classes) also **static functions** (or static methods). We can think about a function as a piece of code that can be executed (invoked) many times from other functions (for example, from **main**) in such a way that in each invocation some data that the function operates on may be different. Let us consider an example of two static functions: **isPrime** and **primesBetween**. Of course, as always in Java, they must be placed in a class. The first of these two functions checks if a given number is prime, the second prints prime numbers in a given interval.

Listing 18 BHK-StatFun/StatFun.java

```java
public class StatFun {

    static boolean isPrime(int n) {
        n = n >= 0 ? n : -n;
        if (n <= 1) throw new IllegalArgumentException();
        if (n <= 3) return true;
        if (n%2 == 0) return false;
        boolean res = true;
        for (int p = 3; p*p <= n && res; p += 2)
            if (n%p == 0) res = false;
        return res;
    }

    static void primesBetween(int a, int b) {
        for (int num = a; num <= b; ++num) {
            boolean prime = isPrime(num);
            System.out.println(num + " is " +
                (prime ? "" : "NOT ") + "prime");
        }
    }

    public static void main (String[] args) {
        int c = 2;
        primesBetween(c, 20);
    }
}
```

As we can see, the definition of a static function is of the form

```java
static Type funName(Type1 parName1, Type2 parName2, ...) {
    // body of the function
}
```

and consists of

- The keyword **static** (there are also functions which are not static). Before or after this keyword we could place an access specifier of the function (**public** or

private) — we will discuss it later.
- The name of the type of a result which this function yields (the so called *return type*); void if the function doesn't return any result (as, in our example, primesBetween).
- A name of the function (it should always start with a lower-case letter but is otherwise arbitrary).
- In round parentheses, a list of parameters: these are comma separated pairs Type parName where Type is the name of a type and parName is an (arbitrary) name of this parameter (it should start with a lower-case letter). The list of parameters can be empty, but parentheses are always required.
- The body of the function enclosed in braces.

Names of parameters are arbitrary and have nothing to do with names declared in other functions (as their parameters or variables defined inside them). We invoke (call) the function just by its name with **arguments** corresponding to the function's parameters enclosed in round parentheses:

```
... funName(arg1, arg2, ...) ...
```

What happens is:

- *Copies* of the values of arguments are pushed (placed) on the program's stack (a special region of memory).
- These copies of the arguments, laying on the stack, can be accessed inside the body of the functions by names of the corresponding declared parameters. Their names in the calling function are completely irrelevant, because a function can see only *values* laying on the stack. We can say that it doesn't even know 'who calls' it and where from. It only knows that there must be values lying on the stack that can be associated with its parameters. In particular, we can use literals as arguments.
- All variables defined *inside* a function are *local* to this function — they are also located on the stack (in our example, variable res in isPrime is such a local variable). Names of local variables are arbitrary and unrelated to local variables of the same name in other functions.
- Instructions in the body of the function are executed. If the function is void, execution stops when the end of the definition is reached or a `return;` statement is encountered. If it is non-void, execution ends when statement `return expr;` is encountered, where expr is an expression whose value is of type declared as the return type of the function (or is convertible to this type).
- When the function returns, the stack is "rewound": it is reverted to the state it had before invocation. In particular, all local variables, including those corresponding to parameters, cease to exist.
- If the function returns a value, the invocation expression (something like `fun(a)`) may be considered to be a *temporary*, unmodifiable variable whose type is the return type of the function and value is that of expr appearing in the `return expr;` expression. It is a temporary variable, so normally we have to do something with it: print it, assign its value to a variable, or use it in another expression.

The program above prints:

```
2 is prime
3 is prime
```

```
4 is NOT prime
5 is prime
6 is NOT prime
7 is prime
8 is NOT prime
9 is NOT prime
10 is NOT prime
11 is prime
12 is NOT prime
13 is prime
14 is NOT prime
15 is NOT prime
16 is NOT prime
17 is prime
18 is NOT prime
19 is prime
20 is NOT prime
```

Functions can be **recursive**, which means that they can call itself: each such invocation creates independent frame on the stack. Of course, we have to ensure that such recursive invocations will not be executed forever — somewhere in the body of the function, usually at the beginning, some condition must be checked telling if the function should return without calling itself. A classic example is the factorial function: factorials 0! and 1! are returned without recursive invocation, for other values recursion $n! = n \cdot (n-1)!$ is used:

Listing 19 BHL-RecFun/RecFun.java

```java
public class RecFun {

    static int fact(int n) {
        if (n < 0) throw new IllegalArgumentException();
        if (n <= 1) return 1;
        return n*fact(n-1);
    }

    public static void main (String[] args) {
        for (int num = 0; num <= 12; ++num)
            System.out.println("Factorial of " + num +
                               " is " + fact(num));
    }
}
```

which prints

```
Factorial of 0 is 1
Factorial of 1 is 1
Factorial of 2 is 2
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

```
    Factorial of 6 is 720
    Factorial of 7 is 5040
    Factorial of 8 is 40320
    Factorial of 9 is 362880
    Factorial of 10 is 3628800
    Factorial of 11 is 39916800
    Factorial of 12 is 479001600
```

(note that 13! is already too big to fit in an **int**!)

Section 7

Arrays

Arrays are the simplest data structure. An array can be viewed as a fixed-sized collection of elements of the same type in which elements are ordered and can be accessed by specifying their **index**. Indices start with 0 (first element), so the last element has index size-1, where size is the size (length, dimension) of the array, i.e., number of its elements. In Java, arrays are *objects* — this means that they carry not only information on their elements but also some other information, in particular on their size. It also means that they are always created on the heap and never have names: we can refer to them only using *references* to them (i.e., in C/C++ language, pointers — variables which hold, as their values, *addresses* of objects).

7.1 Creating arrays

Arrays can be created in several ways, illustrated in the example below. Points to note:

- When an array is created, its size must be specified and then cannot be modified.
- The type *reference to array of elements of type* **Type** is denoted by **Type[]**. Statement `int[] arr;` means that arr is a reference to array of **int**s — only a reference (with value **null**) is created, not an array! One can also write `int arr[];` but this notation is not recommended.
- If arr is a reference to an array, the expression arr.length is of type **int** and its value is the length (size, dimension) of the array referenced to by arr.
- There is a special kind of loop which can be used to iterate over elements of an array (and, as we will see later, over elements of other types of collections as well). It has the form:
  `for (Type elem : arr) { ... }`
  where **Type** is the type of elements of the array arr, elem is any identifier, and arr is the reference to an array. In consecutive iterations of the loop, elem will be a *copy* of consecutive elements of the array.

Example: program

```
public class Arr {
    public static void main (String[] args) {
        int[] a0 = new int[]{1, 2, 3};
        int[] a1 = {4, 5, 6};
        int[] arrsum = new int[a0.length];
        for (int i = 0; i < arrsum.length; ++i) {
            arrsum[i] = a0[i] + a1[i];
        }
        for (int n : arrsum) {
            System.out.print(n + " ");
        }
    }
}
```

prints

```
5 7 9
```

Variables declared as arrays are really pointers (in Java called *references*) to anonymous objects representing arrays. This means that when we pass an array to a function (or return an array from a function), what we are really passing is a copy of the *address* of the array, not the array itself; consequently, having this address, functions which receive it *can* modify the original array. Examples can be found in the following program:

Listing 20 CYD-BasicArray/BasicArr.java

```java
import static java.lang.System.out; // for convenience

public class BasicArr {

    public static void main(String[] args) {
        int[] a1 = {1,2,3};
        printArr(a1, "Array a1");

        int[] a2;  //  <-- no array here, only reference!
        a2 = new int[]{4,5,6};
        printArr(a2, "Array a2");

        a1 = a2;   // <-- whatever was in a1 is lost!
        printArr(a1, "After a1=a2, a1 is");

          // a1 and a2 now refer to the same array!
        a1[0] = 44;
        a2[2] = 66;
        printArr(a1, "After modifications a1 is");
        printArr(a2, "After modifications a2 is");

          // ad hoc array
        printArr(new int[]{7,8,9}, "Ad hoc array");

          // array returned from a function
        a2 = getArr(5);
        printArr(a2, "Array returned from function");

          // passing  r e f e r e n c e  to function
        reverseArr(a2);
        printArr(a2, "Array a2 reversed");
    }

    /**
     *  returns first n triangular numbers
     */
    private static int[] getArr(int n) {
        int[] arr = new int[n];
        for (int i = 0; i < n; ++i)
            arr[i] = (i+1)*(i+2)/2;
        return arr;
    }

    /**
     *  prints an array of integer numbers
     */
    private static void printArr(int[] a, String message) {
        out.print('\n' + message + ": [");
        for (int i : a)
            out.print(" " + i); // <-- 'i' cannot be changed
        out.println(" ]; size = " + a.length);
    }
    /**
     *  modifies input array (reversing order of elements)
     */
    private static void reverseArr(int[] a) {
        for (int i = 0, j = a.length-1; i < j; ++i,--j) {
            int p = a[i];
            a[i]  = a[j];
            a[j]  = p;
        }
    }
}
```

7.2 Arrays of references to objects

Arrays of objects do not exist in Java (as they *do* exist in C++). Instead, we can create arrays of references (pointers) to objects. If not initialized otherwise, all elements of such an array will initially be **null**:

```java
public class ArrStrings {
    public static void main (String[] args) {
        String[] arr = null;
          // prints: null
        System.out.println(arr);

        arr = new String[4];
          // prints: null null null null
        for (String s : arr) System.out.print(s + " ");
        System.out.println();

        arr[0] = arr[2] = "Ala";
          // prints: Ala null Ala null
        for (String s : arr) System.out.print(s + " ");
        System.out.println();
    }
}
```

7.3 Multi-dimensional arrays

Strictly speaking, there are no multi-dimensional arrays in Java. However, it is possible that elements of an array are references to arrays of some type. Therefore, after

```
int[][] b = { {1,2,3}, {4,5,6,7}, {11,12} };
```

the variable b is the reference to an array of references to arrays of **int**s. In particular, the type of b[1] is *reference to array of* **int***s*, in this case the array {4,5,6,7}. Expression b.length is 3, as there are three references to arrays in b, while b[1].length is 4, as b[1] is the reference to array of **int**s with four elements. The type **int[][]** is *reference to array of references to arrays of* **int***s*.
Let us consider another example:

Listing 21 CYB-SimpleArrays/SimpleArrays.java

```java
public class SimpleArrays {
    public static void main(String[] args) {

        // ==================================================
        int[] a = {1,2,3};
        System.out.println("a.length = " + a.length);
        for (int i = 0; i < a.length; ++i)
            a[i] = (i+1)*(i+1);
        for (int i = 0; i < a.length; ++i)
            System.out.print(a[i]+" ");
        System.out.println('\n');

        // ==================================================
        int[][] b = { {1,2,3}, {4,5,6,7,8}, {11,12} };
        System.out.println("b.length = " + b.length);
        for (int row = 0; row < b.length; ++row)
            System.out.println("b["+row+"].length = " +
                                             b[row].length);
        System.out.println();

        for (int row = 0; row < b.length; ++row) {
            for (int col = 0; col < b[row].length; ++col)
                System.out.print(b[row][col]+" ");
            System.out.println();
        }
        System.out.println('\n');

        // ==================================================
        int[] c = new int[]{1,2,3}; // <- size inferred
        System.out.println("c.length = " + c.length);
        for (int i = 0; i < c.length; ++i)
            System.out.print(c[i]+" ");
        System.out.println('\n');

        // ==================================================
        int[] d = new int[5];       // <- elements are 0
        System.out.println("d.length = " + d.length);
        for (int i = 0; i < d.length; ++i)
```

```java
                System.out.print(d[i]+" ");
            System.out.println('\n');

            // ================================================
            int[][] e = new int[3][2];   // <- a 3x2 matrix
            System.out.println("e.length = " + e.length);
            for (int row = 0; row < e.length; ++row)
                for (int col = 0; col < e[row].length; ++col)
                    e[row][col] = row+col;

            // ================================================
            int[][] f = new int[3][];
            for (int row = 0; row < f.length; ++row)
                System.out.print(f[row]+" ");
            System.out.println();

            for (int row = 0; row < f.length; ++row)
                f[row] = new int[row*row+2];

            for (int row = 0; row < f.length; ++row) {
                System.out.println("f["+row+"].length = " +
                                            f[row].length);
                for (int col = 0; col < f[row].length; ++col)
                    f[row][col] = row+col;
            }
            System.out.println();

            for (int row = 0; row < f.length; ++row) {
                for (int col = 0; col < f[row].length; ++col)
                    System.out.print(f[row][col]+" ");
                System.out.println();
            }
        }
    }
```

In the next example, we use a three-dimensional array of **ints**: the first index corresponds to a student, the second to a course he/she is taking and the third to the grades for this student and this course:

Listing 22 CYJ-Arr3D/Arr3D.java

```java
public class Arr3D {

    public static void main(String[] args) {
        String[] subjects = {
            "Math", "Programming", "English"
        };

        String[] students = {
            "John",  "Mark",  "Jim", "Henry",
            "Peter", "Kevin", "Jack"
        };
          // three-dimensional array of grades:
          //     first index  - student
          //     second index - subject for a given student
          //     third index  - grades for a given student
          //                    and a given subject
        int[][][] grades = {
          //   Math     Programming    English
            { {3,4,3}, {4,3,3,4,4,3}, {4,3,3} }, // stud. 0
            { {3,5},   {5,2,3,3,4},   {2,4}   }, // stud. 1
            { {5,4,4}, {5,5,5,4},     {3}     }, // stud. 2
            { {3,4,3}, {4,3,3,3,3},   {3,3,4} }, // stud. 3
            { {4,3},   {4,3,3},       {5,3}   }, // stud. 4
            { {5,3},   {4,2,3},       {3,3}   }, // stud. 5
            { {5,4},   {4,4,5},       {5,3}   }, // stud. 6
        };

        String[] pom = new String[students.length];
        int count = 0;
          // over students
        for (int s = 0; s < grades.length; ++s) {
            double ave = 0;
            int    num = 0;
              // over subjects for student s
            for (int c = 0; c < grades[s].length; ++c) {
                num += grades[s][c].length;
                  // over grades for student s, subject c
                for (int g = 0; g < grades[s][c].length;++g)
                    ave += grades[s][c][g];
            }
            ave /= num;
            if (ave > 4) pom[count++] = students[s];
        }

        String[] result = new String[count];
        for (int i = 0; i < count; ++i)
            result[i] = pom[i];

        System.out.print("Best students of the group:");
        for (String s : result)
            System.out.print(" " + s);
        System.out.println();
    }
}
```

List of listings

Index