

Java - lecture notes

with example programs from archive [*java_examples*](#)

Tomasz R. Werner

Warsaw, October 10, 2018 13:59

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:

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

Contents

	Page
1 General introduction	1
1.1 Computers and programming languages	1
1.2 What is Java?	2
2 Compiling and running a Java program	4
3 Types, variables, literals	6
3.1 Primitive types	6
3.2 Object types	6
3.3 Variables and literals	7
3.4 Conversions	9
4 Quick introduction to I/O	11
5 Instructions and operators	13
5.1 Operators	13
5.1.1 Assignment operator	13
5.1.2 Arithmetic operators	14
5.1.3 Conditional and relational operators	15
5.1.4 Bit-wise operators	17
5.1.5 Precedence and associativity	19
5.2 Instructions	22
5.2.1 Conditional statements	22
5.2.2 Switch statement	23
5.2.3 Loops	26
6 List of listings	32
Index	33

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 ἀριθμός (which means number). [By the way, the term *algebra* comes from 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 page¹ — 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 online².

¹<https://www.oracle.com/technetwork/java/javase/downloads/index.html>

²<https://docs.oracle.com/javase/10/docs/api>

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

```

1  /*
2   *   Program Hello
3   *   It prints "Hello, World"
4   */
5
6  public class Hello { /* Entry class must be public
7                        File name = class name!    */
8      public static void main(String[] args) {
9          System.out.println("Hello, World");
10     }
11     // signature of 'main' always like this
12 }
```

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

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**.

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 end 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).

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;
```

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). Characters that are not present on our keyboard may be entered like this

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

by writing, after a backslash and letter '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

```
1 public class Literals {
2     public static void main(String[] args) {
3         System.out.println(22);           // decimal
4         System.out.println(022);           // octal
5         System.out.println(0x22);          // hexadecimal
6         System.out.println(0b1001);        // binary
7         System.out.println(22.22);         // double
8         System.out.println(2.22e-1);       // "scientific"
9         System.out.println(1/3 );          // this is 0 !
10        System.out.println(1/3.);          // one third
11        System.out.println(1/3D);          // 3D -> double
12        System.out.println(2147483648L);    // long
13        System.out.println(2147483647 + 1 ); // oops!
14        System.out.println(2147483647L + 1 );
15        System.out.println('A');           // char
16        System.out.println('A'+2);         // char
17        System.out.println((char)('A'+2));
18        System.out.println('\u0042');      // also char
19        System.out.println("Hello, World");
20        System.out.println("\u017b\u00F3\u0142w");
21        System.out.println("number = " + 2+2);
22        System.out.println("number = " + (2+2));
23        System.out.println(false);
24        System.out.println(2*3 == 6);
25        System.out.println("\"TAB\"s and 'NL'\n"+
26                               "a\tb\tc\te\tf\n\tg\tth\ti\tj");
27        System.out.println("C:\\Program Files\\java");
28    }
29 }
```

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

Listing 3

AAL-Variables/Variables.java

```
1 public class Variables {
2     public static void main(String[] args) {
3         int    ifour = 4;
4         double xhalf = 0.5;
5         double four  = ifour;    // automatic conversion
6         // int badFour = 4.0;    // WRONG
7         int k = 1, m, n = k + 3;
8         m = 2;
9         final double two = xhalf*ifour;
10        // two = two + 2;        // WRONG
11        boolean b = true;
12        if (b) System.out.println(
13            "k=" + k + ", m = " + m + ", n = " + n +
14            "\nSum by 4 is equal to " + (k+m+n)/ifour);
15        String john,           // does string john exist?
16            mary = "Mary";
17        john = "John";
18        System.out.println(john + " and " + mary);
19        john = mary;
20        // reference copied, string "John" lost!
21        System.out.println(john + " and " + mary);
22    }
23 }
```

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 **doubles** 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.

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

```

1  import java.util.Scanner;
2  import java.util.Locale; // see below
3
4  public class ReadKbd {
5      public static void main(String[] args) {
6
7          // If locale is Polish, floating point numbers
8          // have to be entered with c o m m a as the
9          // decimal separator. Locale can be changed to,
10         // e.g., American, by uncommenting the line below:
11         //     Locale.setDefault(Locale.US);
12         // (then the dot is is used as decimal separator).
13         // When reading data, any nonempty sequence of
14         // white characters is treated as a separator.
15
16         Locale.setDefault(Locale.US);
17         Scanner scan = new Scanner(System.in);
18
19         System.out.println("Enter an int, a string " +
20                             "and two double's");
21         int    k = scan.nextInt();
22         String s = scan.next();
23         double x = scan.nextDouble();
24         double y = scan.nextDouble();
25
26         System.out.println("\nEnter data:\n\nint      = " +
27                             k + "\nString  = " + s + "\ndouble1 = " +
28                             x + "\ndouble2 = " + y + "\n");
29         scan.close();
30     }
31 }

```

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

```
1 import javax.swing.JOptionPane;
2
3 public class ReadGraph {
4     public static void main(String[] args) {
5
6         // simple form of showInputDialog...
7         String s = JOptionPane.showInputDialog(
8             null, "Enter an integer");
9         // parsing string to get an int
10        int k = Integer.parseInt(s);
11
12        // parsing string to get a double
13        s = JOptionPane.showInputDialog(
14            null, "Enter a double");
15        double x = Double.parseDouble(s);
16
17        s = JOptionPane.showInputDialog(
18            null, "Enter a string (spaces OK)");
19        JOptionPane.showMessageDialog(null,
20            "Data entered: int=" + k + ", double=" +
21            x + ", string=\"" + s + "\"");
22        System.exit(0); // kills JVM
23    }
24 }
```

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. *Preincrementation* or *decrementation* (`++a` or `--a`) are 'immediate', while *postincrementation* 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 always 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

```
1 public class RelOps {
2     public static void main (String[] args) {
3         int a = 1, b = 8, d = 8;
4         System.out.println(
5             "Is d in [a,b]: " +
6             ( a <= d && d <= b ));
7         System.out.println(
8             "Is d in (a,b): " +
9             ( a < d && d < b ));
10        System.out.println(
11            "Is d outside (a,b): " +
12            ( d < a || d > b ));
13        System.out.println(
14            "Is d outside (a,b): " +
15            ( !(d >= a || d <= b) ));
16        System.out.println(
17            "Is d == b: " + (d == b));
18        System.out.println(
19            "Is d != b: " + (d != b));
20    }
21 }
```

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 **ints**):

```
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,

```
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

```
1 import java.util.Scanner;
2 import java.util.Locale;
3
4 public class Largest {
5     public static void main(String[] args) {
6         Scanner scan = new Scanner(System.in);
7         scan.useLocale(Locale.US);
8
9         System.out.print(
10             "Enter three numbers (with " +
11             "DOT as dec. separator!) -> ");
12
13         double a = scan.nextDouble();
14         double b = scan.nextDouble();
15         double c = scan.nextDouble();
16
17         double largest = b > a ? b : a;
18         largest = c > largest ? c : largest;
19
20         double smallest = b < a ? b : a;
21         smallest = c < smallest ? c : smallest;
22
23         System.out.println("Number " + largest + " is " +
24             "the largest and " + smallest +
25             " is the smallest");
26     }
27 }
```

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 (\ll) 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 (\gg) 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 (\gg) 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 (\mid), 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 (\wedge), 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 \wedge b	0 0 1 0 0 0 1 0

As can be expected, negating operator (\sim) just reverses the bits

a	1 0 1 0 0 1 0 0
\sim a	0 1 0 1 1 0 1 1

Listing 8

BAA-Bits/Bits.java

```
1 public class Bits {
2     public static void main (String[] args) {
3         int a = 0b11111111;    // 255 or 0xFF
4         System.out.println("a = " + a);
5         int b = 0x7F;          // 127
6         System.out.println("b = " + b);
7
8         a = 3; // 00...011
9         System.out.println(a + " " + (a << 1) + " " +
10                             (a << 2) + " " + (a << 3));
11
12         a = -1;
13         int firstByte = a & 255;
14         int secondByte = (a >> 8) & 0xFF;
15         System.out.println("-1: " + secondByte + " " +
16                             firstByte);
17
18         a = 0b1001;
19         b = 0b0101;
20         System.out.println("AND: " + (a & b) + "; " +
21                             "OR: " + (a | b) + "; " +
22                             "XOR: " + (a ^ b) + "; " +
23                             " ~a: " + (~a) + "; " +
24                             " ~b: " + (~b));
25     }
26 }
```

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 `<`, then 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	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	cond ? expr1 : expr2
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

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

```
1 public class BasicOps {
2     public static void main(String[] args) {
3         int x = 2, y = 2*x, z = x + y;
4         // precedence, associativity
5         System.out.println(x + x + z * y - z / 3); // 26
6         System.out.println(x + (x + z) * (y - z) / 3); // -3
7
8         // div and mod
9         int u = 13;
10        System.out.println(
11            "u = " + u + ", u%7 = " + u%7 + ", u/7 = " +
12            u/7 + ", 7*(u/7)+u%7 = " + (7*(u/7)+u%7));
13        System.out.println();
14
15        u %= 7; // compound assignment
16
17        System.out.println("1. u="+u+", x="+x+", y="+y);
18        x = ++u;
19        System.out.println("2. u="+u+", x="+x+", y="+y);
20        y = x--;
21        System.out.println("3. u="+u+", x="+x+", y="+y);
22        u = (x--y);
23        System.out.println("4. u="+u+", x="+x+", y="+y);
24        u = x = y = (int)(9.99*y);
25        System.out.println("5. u="+u+",x="+x+",y="+y);
26        u = 6 << 2;
27        x = u >> 1;
28        y = 7 >> 1;
29        System.out.println("6. u="+u+",x="+x+",y="+y);
30        u = '\u0041';
31        x = 'a';
32        char c = (char)98;
```



```

33     System.out.println("7. u="+u+",x="+x+",c="+c);
34     System.out.println("Unicode of "+c+" is "+(int)c);
35 }
36 }

```

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

```
1 import java.util.Scanner;
2
3 public class LeapYear {
4     public static void main(String[] args) {
5         Scanner scan = new Scanner(System.in);
6         System.out.print("Enter a year (integer) -> ");
7         int year = scan.nextInt();
8         scan.close();
9
10        boolean is_leap;
11
12        if (year%400 == 0)
13            is_leap = true;
14        else if (year%100 == 0)
15            is_leap = false;
16        else if (year%4 == 0)
17            is_leap = true;
18        else
19            is_leap = false;
20        // ?: operator used below!
21        System.out.println("Year " + year + " is " +
22                           (is_leap? "" : "not ") + "a leap year");
23    }
24 }
```

5.2.2 Switch statement

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

```
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

```

1  import java.util.Scanner;
2
3  public class SimpleSwitch {
4      public static void main (String[] args) {
5          Scanner scan = new Scanner(System.in);
6          System.out.print("Enter an initial: ");
7          char initial = scan.next().charAt(0);
8          scan.close();
9
10         switch (initial) {
11             case 'A':
12             case 'a':
13                 System.out.println("Amelia");
14                 break;
15             case 'B':
16             case 'b':
17                 System.out.println("Barbra");
18                 break;
19             case 'C':
20             case 'c':
21                 System.out.println("Cindy");
22                 break;
23             case 'D':
24             case 'd':
25                 System.out.println("Doris");
26                 break;
27             default:

```

```

28         System.out.println("Invalid input");
29     }
30 }
31 }

```

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

Listing 12

ACD-Switch/Switch.java

```

1  import java.util.Scanner;
2
3  public class Switch {
4      public static void main(String[] args) {
5          Scanner scan = new Scanner(System.in);
6
7          System.out.print(
8              "Enter a single hex digit -> ");
9          char ch = scan.next().charAt(0);
10         scan.close();
11
12         // toLower by hand...
13         if (ch >= 'A' && ch <= 'Z')
14             ch = (char)(ch + 'a' - 'A');
15
16         int num;
17
18         switch (ch) {
19             case '0': case '1': case '2':
20             case '3': case '4': case '5':
21             case '6': case '7': case '8':
22             case '9':
23                 num = ch - '0';
24                 break;
25             case 'a': case 'b': case 'c':
26             case 'd': case 'e': case 'f':
27                 num = 10 + ch - 'a';
28                 break;
29             default:
30                 num = -1;
31         }
32
33         System.out.println("Character '" + ch + "' is " +
34             (num >= 0 ? "" : "not ") + "a hex digit. " +
35             "Its numerical value: " + num);
36     }
37 }

```

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

```
1 import java.util.Scanner;  
2  
3 public class Prime {  
4     public static void main(String[] args) {  
5         Scanner scan = new Scanner(System.in);  
6  
7         while (true) {  
8             System.out.print(  
9                 "Enter natural number (0 to exit) -> ");  
10            int n = scan.nextInt();  
11  
12            if (n == 0) break;  
13  
14            boolean prime = true;  
15  
16            if (n == 1) {  
17                System.out.println("Number 1 is neither " +  
18                    "prime nor composite");  
19                continue;  
20            } else if (n > 2 && n%2 == 0) {  
21                prime = false;  
22            } else {  
23                int p = 3;  
24                while (p*p <= n) {  
25                    if (n%p == 0) {  
26                        prime = false;  
27                        break;  
28                    }  
29                    p++;  
30                }  
31            }  
32            if (prime) System.out.println(n + " is prime");  
33            else System.out.println(n + " is not prime");  
34        }  
35    }  
36 }
```

```

28         }
29         p += 2;
30     }
31 }
32     System.out.println("Number " + n + " is " +
33         (prime ? "prime" : "composite"));
34 }
35     scan.close();
36 }
37 }

```

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

```

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

```

1  import java.util.Scanner;
2
3  public class Prime {
4      public static void main(String[] args) {
5          Scanner scan = new Scanner(System.in);
6
7          MAIN_LOOP:
8          while (true) {
9              System.out.print(
10                  "Enter natural number (0 to exit) -> ");
11              int n = scan.nextInt();
12
13              if (n == 0) break;
14
15              if (n == 1) {
16                  System.out.println("Number 1 is neither " +

```

```

17         "prime nor composite");
18         continue;
19
20     } else if (n > 2 && n%2 == 0) {
21         System.out.println("Number " + n +
22             ", being even, is composite");
23         continue;
24
25     } else {
26         int p = 3;
27         while (p*p <= n) {
28             if (n%p == 0) {
29                 System.out.println("Number " + n +
30                     " is composite");
31                 continue MAIN_LOOP;
32             }
33             p += 2;
34         }
35         System.out.println("Number " + n +
36             " is prime");
37     }
38 }
39 scan.close();
40 }
41 }

```

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

```

1 public class Dice {
2     public static void main (String[] args) {
3         int a, b;
4         do {

```

```

5      a = 1 + (int)(Math.random()*6);
6      b = 1 + (int)(Math.random()*6);
7      System.out.println("a=" + a + " b=" + b);
8  } while (a != 6 || b != 6);
9  }
10 }
```

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:

```

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 one or several comma-separated expressions;
- *condition*: expression with a value of type **boolean**;
- *incr*: 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

```

1  import java.util.Scanner;
2
3  public class Stars {
4      public static void main (String[] args) {
```



```

5      Scanner scan = new Scanner(System.in);
6      System.out.print("Enter a positive odd number: ");
7      int n = scan.nextInt();
8      scan.close();
9
10     for (int len=1, sp=n/2; len <= n; len+=2, --sp) {
11         for (int i = 0; i < sp; ++i)
12             System.out.print(" ");
13         for (int i = 0; i < len; ++i)
14             System.out.print("*");
15         System.out.println();
16     }
17 }
18 }

```

For example, after inputting 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

```

1  import java.util.Scanner;
2  /*
3   * Finding and printing values of Euler's totient
4   * function, i.e., number of positive integers up
5   * to a given integer n that are relatively prime to n.
6   */
7
8  public class ForWhileEuler {
9      public static void main(String[] args) {
10         Scanner scan = new Scanner(System.in);
11
12         while (true) {
13             System.out.print(
14                 "\nEnter a natural number (0 to exit) -> ");
15             int n = scan.nextInt();
16
17             if (n == 0) break;
18
19             int count = 0;
20             for (int p = 1; p <= n; ++p) {
21                 int a = n, b = p;

```

```

22         // Euclid's algorithm for GCD
23         while (a != b) {
24             if (a > b) a -= b;
25             else      b -= a;
26         }
27         if (a == 1) ++count;
28     }
29     System.out.print("\u03c6(" + n + ") = " + count);
30 }
31 }
32 }

```

List of listings

1	AAC-HelloWorld/Hello.java	4
2	AAG-Literals/Literals.java	8
3	AAL-Variables/Variables.java	9
4	AAI-ReadKbd/ReadKbd.java	11
5	AAJ-ReadGraph/ReadGraph.java	12
6	ABY-RelOps/RelOps.java	16
7	ACB-CondOp/Largest.java	17
8	BAA-Bits/Bits.java	19
9	AAP-BasicOps/BasicOps.java	21
10	ABX-Ifs/LeapYear.java	23
11	ACC-SimpleSwitch/SimpleSwitch.java	24
12	ACD-Switch/Switch.java	25
13	AFA-While/Prime.java	26
14	AFB-WhileBis/Prime.java	27
15	AFE-DoWhileDice/Dice.java	28
16	AFH-ForPyram/Stars.java	29
17	AFJ-ForWhileEuler/ForWhileEuler.java	30

Index

- << operator, 18
- = operator, 13
- >> operator, 18
- >>> operator, 18
- & operator, 18
- && operator, 15
- | operator, 18
- || operator, 15
- ! operator, 15
- ^ operator, 16, 19

- algorithm, 2
- AND operator, 15
- argument, 13
- arithmetic operators, 14
- assignment operator, 13
- associativity, 20

- binary operator, 13
- bit, 1
- Bit-wise operators, 17
- block, 22
- Boolean type, 6
- byte, 1
- byte code, 5

- casting, 9
- class, 4, 7
 - name of, 4
 - Scanner, 11
- class file, 5
- comment, 4
- compiler, 2
- compound assignment operator, 13
- compound instruction, 22
- concurrency, 3
- conditional operator, 16
- conditional statement, 22
- conversion, 9

- do-while loop, 28

- Euclid, 2
- executable, 1
- expression, 22, 29

- floating point type, 6
- for loop, 29
- function
 - main, 4

- garbage collector, 7
- Gosling J., 2
- graphical user interface, 3
- GUI, 3

- hexadecimal notation, 1

- instruction, 1, 22
 - compound, 22
- instruction set, 1
- integral type, 6
- interpreter, 2

- Java Virtual Machine, 3, 5
- JIT, 5
- just-in-time compilation, 5
- JVM, 3, 5

- labeled loop, 27
- logical type, 6
- loop
 - do-while, 28
 - for, 29
 - labeled, 27
 - named, 27
 - while, 26

- machine code, 1
- main, 4
- modulus operator, 14
- multithreading, 3

- named loop, 27
- network programming, 3
- NOT operator, 15

- object type, 6
- operand, 13
- operating system, 1
- operator, 13
 - <<, 18
 - =, 13
 - >>, 18
 - >>>, 18
 - &, 18
 - &&, 15
 - |, 18

<p> , 15 !, 15 ^, 16, 19 AND, 15 arithmetic, 14 assignment, 13 associativity, 20 binary, 13 bit-wise, 17 compound assignment, 13 conditional, 16 modulus, 14 NOT, 15 OR, 15 precedence, 19 remainder, 14 shift, 18 short-circuited, 15 ternary, 13 unary, 13 XOR, 16 OR operator, 15 pointer, 6 precedence of operators, 19 primitive type, 6 processor, 1 program, 1 reference type, 6 </p>	<p> register, 1 remainder operator, 14 Scanner (class), 11 shift operator, 18 short-circuited operator, 15 statement, 22 conditional, 22 switch, 23 switch statement, 23 ternary operator, 13 type, 6 boolean, 6 casting, 9 conversion, 9 floating point, 6 integral, 6 logical, 6 object, 6 primitive, 6 reference, 6 unary operator, 13 variable, 6, 7 while loop, 26 XOR operator, 16 </p>
---	--