Java - lecture notes

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

Tomasz R. Werner

Warsaw, January 19, 2023 9:37

RULES: Points for the LAB: classes (tasks, activity) — 15 points, 3 tests — $3 \times 25 = 75$ points, assignments (homework) — 10 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 75 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 40 means a failure... If the score for the exam is at least 40, the total score will be calculated as half of the sum of the results obtained for the LAB and for the exam (i.e., max. will be 100).
The final note will be then claculated as follows:
GRADES:

$[50 - 60) \Rightarrow 3.0$ $[60 - 70) \Rightarrow 3.5$ $[70 - 75) \Rightarrow 4.0$ $[75 - 85) \Rightarrow 4.5$ $[85 - 100] \Rightarrow 5.0$

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
    3.1.1 Integral types . . . 6
    3.1.2 Floating point types . . . 9
  3.2 Object types . . . 10
  3.3 Variables and literals . . . 10
  3.4 Conversions . . . 14
  3.5 The stack and the heap . . . 15
4 Quick introduction to I/O . . . 16
5 Instructions and operators . . . 18
  5.1 Operators . . . 18
    5.1.1 Assignment operator . . . 18
    5.1.2 Arithmetic operators . . . 19
    5.1.3 Conditional and relational operators . . . 21
    5.1.4 Bit-wise operators . . . 24
    5.1.5 Casting operator . . . 26
    5.1.6 Precedence and associativity . . . 26
  5.2 Instructions . . . 29
    5.2.1 Conditional statements . . . 29
    5.2.2 Switch statement and expression . . . 30
    5.2.3 Loops . . . 35
6 Arrays . . . 43
  6.1 Creating arrays . . . 43
  6.2 Arrays of references to objects . . . 46
  6.3 Multi-dimensional arrays . . . 46
7 Static functions . . . 51
8 Classes . . . 59
  8.1 Basic concepts . . . 59
  8.2 Classes and objects . . . 59
  8.3 Access to classes and their members . . . 62
  8.4 Constructors and methods . . . 63
  8.5 Static members . . . 68
  8.6 Initializing blocks . . . 72
  8.7 Singleton classes . . . 75
9 Basic data structures . . . 77
  9.1 Singly linked lists . . . 77
  9.2 Stacks . . . 80
  9.3 Queues . . . 82
10 *String*s and *StringBuilder*s . . . 85
  10.1 Class *String* . . . 85
  10.2 Class *StringBuilder* . . . 88
11 Introduction to inheritance . . . 90
  11.1 Importance of equal and hashCode methods . . . 96

12 Exceptions 99
12.1 *try-catch* blocks 100
12.2 *finally* block 101
12.3 Propagating exceptions 102
12.4 Throwing exceptions 102
12.5 Examples 103
12.6 Assertions 109
13 IO streams primer 110
13.1 Binary and text IO streams 110
13.2 *StreamTokenizer* class 118
13.3 IO cheat sheet 119
13.3.1 Reading/writing binary files byte-by-byte 119
13.3.2 Reading a binary file into an array of bytes 120
13.3.3 Reading/writing text files line-by-line 120
13.3.4 Reading/writing text files character-by-character 121
14 Regular expressions 122
14.1 Basic concepts 122
14.1.1 Classes 122
14.1.2 Predefined classes 123
14.1.3 Special locations 123
14.1.4 Quantifiers 123
14.2 Regular expressions in methods of class *String* 124
14.3 Patterns and Matchers 125
14.4 Capturing groups 127
14.5 Option flags 130
14.5.1 *MULTILINE* 130
14.5.2 *DOTALL* 131
14.5.3 *CASE_INSENSITIVE* and *UNICODE_CASE* 131
14.5.4 *UNICODE_CHARACTER_CLASS* 132
14.6 Some examples 132
15 List of listings 141
Index 143

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 them into machine code which can then be passed to the processor. Sometimes the result is not an executable, but has 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, or it can be viewed online[2].

[1]https://www.oracle.com/technetwork/java/javase/downloads/index.html
[2]https://docs.oracle.com/en/java/javase/19/docs/api/java.base/module-summary.html

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 to avoid vexing trouble, 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 not yet 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, there are no references to variables of primitive types in Java). In C/C++ such variables are called **pointers**. [Note that C++ also uses references, but they do not have much in common with what we call a reference in Java.]

3.1.1 Integral types

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:

$$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

This can be interpreted as a number in the binary system, so consecutive digits (from the right) are coefficients at consecutive powers of 2. There is a quirk, however: in order to be able to represent also negative numbers, the term with the highest power of 2 is taken with negative sign

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

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 — for numbers represented by one byte it would be

    01111111

which is 127_{10}. Expressing groups of four bits as hexadecimal digits (see below) the same number is 7F. The smallest (negative) **byte** will have 1 at the negative part and all zeros at the positive ones

    10000000

what in hexadecimal notation would be -80 (it corresponds to -128_{10}). This reasoning applies to the remaining integral types, except **char** — here we count all terms (**char** in Java is 16 bits long) with plus sign.

There are special operators that operate on integers treating them not as numbers but rather as sequences of bits which can convey some information (not necessarily numerical). We will show these operators in Sec. 5.1.4 on page 24. For now just few examples.

If we have two integer numbers (we will use only eight-bit numbers for simplicity), we can AND them: the result will be a sequence of bits where there is 1 whenever there is 1 in both numbers at the corresponding position, and 0 otherwise — we can interpret it as the logical conjunction bit-by-bit with 1 corresponding to true and 0 to false:

```
   0 1 1 0 1 1 0 0
   0 1 0 1 0 1 0 1
   ---------------
&  0 1 0 0 0 1 0 0
```

The symbol of AND is '&', so interpreting these sequences of bits as numbers, we get $108 \ \& \ 85 = 68$.

Similarly, ORing corresponds to alternative, or logical disjunction (denoted by a vertical bar) — to be true, at least one operand must be true:

```
   0 1 1 0 1 1 0 0
   0 1 0 1 0 1 0 1
   ---------------
|  0 1 1 1 1 1 0 1
```

what corresponds to $108 \mid 85 = 125$.

Another operation is XORing (exclusive or), denoted by '^'. Here we get true only if two operands are *different*:

```
   0 1 1 0 1 1 0 0
   0 1 0 1 0 1 0 1
   ---------------
^  0 0 1 1 1 0 0 1
```

or $108 \ \hat{} \ 85 = 57$. XORing has several interesting (and useful) features. For example, let's consider XORing a number with 1's:

```
    0 1 1 0 1 1 0 0
    1 1 1 1 1 1 1 1
    ---------------
 ^  1 0 0 1 0 0 1 1
```

A we can see, the result is like the original value, but with all bits flipped: $0 \to 1$, $1 \to 0$. Now let's XOR the same number with 0's:

```
    0 1 1 0 1 1 0 0
    0 0 0 0 0 0 0 0
    ---------------
 ^  0 1 1 0 1 1 0 0
```

Now the original number has been exactly reproduced. So, XORing with 1 flips the bit, XORing with 0 – leaves it intact. It follows, that XORing any bit with the same bit-value (whether it's 0 or 1) *twice* always reproduces the original value:

```
    0 1 1 0 1 1 0 0
    0 1 0 1 0 1 0 1
    ---------------
 ^  0 0 1 1 1 0 0 1
```

and we XOR the result again with 0 1 0 1 0 1 0 1:

```
    0 0 1 1 1 0 0 1
    0 1 0 1 0 1 0 1
    ---------------
 ^  0 1 1 0 1 1 0 0
```

what reproduces the original sequence of bits 0 1 1 0 1 1 0 0.

Negation (NOT) of a sequence of bits yields the same sequence but with all bits reversed (flipped):

```
   ~0 1 1 0 1 1 0 0   ->   1 0 0 1 0 0 1 1
```

Other useful operations on sequences of bits are shifts — to the left or to the right by a given number of bits. When we shift bits to the left (symbol <<), all bits coming out on the left edge just disappear and zeros come in from the right side:

```
    0 1 1 0 1 1 0 0 << 2   ->   1 0 1 1 0 0 0 0
```

while the opposite holds for right shift (>>>):

```
    0 1 1 0 1 1 0 0 >>> 2   ->   0 0 0 1 1 0 1 1
```

There is also >> operator — here, what comes in from the left is the value of the highest (leftmost) bit: if that is 0 (was corresponds to positive numbers), then zeros come in while if its is 1 (negative numbers), then ones will come in. More details can be found in Section 5.1.4 on p. 24.

3.1.2 Floating point types

The representation of floating point numbers (that represent real numbers known from mathematics) is completely different. Let us consider **float** (although it is not much used). Numbers of this type are stored in 4 bytes, i.e., 32 bits. The highest bit is just the sign bit: 0 for positive values, 1 for negative. Then we have eight bits of the so called exponent, and 23 bits of the mantissa

```
seeeeeeeefffffffffffffffffffffff
```

Eight bits of the exponent are collectively denoted by E and described by the number from the range $[0, 255]$ that these eight bit represent in the binary system. The 23 bits of the mantissa are denoted collectively as F. Then the value is

$$V = (-1)^s \times 2^{E-127} \times 1.F$$

The 127 (the so called *bias*) is needed, because otherwise it would be impossible to represent numbers smaller than one. The first significant digit of any number expressed in binary system must be 1, so this 1 is implicitly added at the beginning, before the binary dot, and is followed by 23 bits of the mantissa. Consequently, we have not 23, but 24 significant binary digits, what corresponds to precision of $24 \cdot \log_{10} 2 \approx 7.2$ decimal digits.

For example, in

```
00111110001000000000000000000000
```

we have $s = 0$, $E = 01111100_2 = 124_{10}$, $F = 01000000000000000000000$, so

$$V = (-1)^0 \times 2^{124-127} \times 1.01_2$$

which, in decimal system, is

$$1.25 \cdot 2^{-3} = \frac{5}{8 \cdot 4} = \frac{5}{32} = 0.15625$$

If $E = 255$ (all ones in binary system), then the interpretation is special: if also $F = 0$, then the number represents $\pm\infty$ (depending on sign bit). If $F \neq 0$ however, then it is a NaN (not-a-number — this will be the result, for example, of taking the logarithm or the square root of a negative number).

Numbers with $E = 0$ are also treated differently (these are the called *subnormal numbers*):

$$V = (-1)^s \times 2^{E-126} \times 0.F$$

As there is no implicit 1 added before the mantissa part, such numbers may have fewer significant digits, so their precision is worse than that of 'normal' numbers.

The basic floating point type is **double**, though, not **float**. It occupies 8 bytes (64 bits): the sign bit, 11 bits of the exponent, 52 bits of mantissa. The bias is 1023. Adding the implicit 1, we have 53 significant binary digits, what corresponds to precision of $53 \cdot \log_2 10 \approx 16$ decimal digits.

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 object types should always start with an upper case letter (it's not enforced by the compiler, but is a convention we should always observe).

Objects of these types (variables) cannot be created locally on the stack and never have names. They are created on the heap and can be 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 value: the object may contain several numbers, Boolean values, and references (addresses) to other objects (e.g., of type **String**); moreover, the class also defines operations that may act upon all this data.

Let us emphasize again that a variable (called *object*) of an object type is *always* anonymous — there is no way to give it any name. Only variables of reference types (pointers), which hold *addresses* of objects as their values, may have names (identifiers).

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;
```

For local variables, instead of declaring a type explicitly, one can use a special keyword **var**: then the compiler will figure out the type itself. Of course, we have to give it a chance to do so, so the variable being defined must be initialised. For example, the definitions above could have been written as

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

because the initialisers on the right tell the compiler that `a` and `b` should be of type **int**.
We can add a letter 'L' (or lower-case 'l', but that could be easily confused with digit 1) 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 wrong, 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 = 189, b = 0275, c = 0xBD, d = 0b10111101;
```

Variables `a`, `b`, `c` and `d` all have the same value 189_{10}, because (taking digits from right to left)

$$189 = 9 \cdot 10^0 + 8 \cdot 10^1 + 1 \cdot 10^2 = 9 + 80 + 100 = 189$$
$$0275 = 5 \cdot 8^0 + 7 \cdot 8^1 + 2 \cdot 8^2 = 5 + 56 + 128 = 189$$
$$0xBD = 13 \cdot 16^0 + 11 \cdot 16^1 = 13 + 176 = 189$$
$$0b10011101 = 1{\cdot}2^0{+}1{\cdot}2^2{+}1{\cdot}2^3{+}1{\cdot}2^4{+}1{\cdot}2^5{+}1{\cdot}2^7 = 1{+}4{+}8{+}16{+}32{+}128{+} = 189$$

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. When writing such literal values, leading zeros may be omitted: instead of 0b0000000000101010, one can write just 0b101010. Additionally, you can insert underscores between digits: they will be ignored by the compiler, but improve readability. The number 1_123_343_198 is much easier to read for human than 1123343198.

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

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

or

```
var x = 1.5;
var 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 (and other symbols) in almost all languages to be represented. Characters that are not present on our keyboard may be entered like this

```java
    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) form-feed (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('\u0042');    // 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");
    }
}
```

which prints

```
22
18
34
9
22.22
0.222
0
0.3333333333333333
0.3333333333333333
2147483648
-2147483648
2147483648
A
67
C
B
Hello, World
Żółw
number = 22
number = 4
false
true
"TAB"s and 'NL'
a       b       c       e       f
        g       h       i       j
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);
    }
}
```

which prints

```
k=1, m = 2, n = 4
Sum by 4 is equal to 1
John and Mary
Mary 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 on 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. Creating a value of one type based on a value of another type 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 and assign it to x. 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.

3.5 The stack and the heap

Let us briefly explain what the stack and the heap are.

Both are parts of memory that are available to the program at runtime. The stack is simpler: when a new value (e.g., of a variable) is to be added, it is always added at the top of the stack. We say that the value is 'pushed' onto the stack. Whenever we create a new variable inside a block of code (e.g., inside a function), its value is pushed onto the stack. Then every time the flow of control leaves the block (e.g., a function exits), all the variables pushed onto the stack in this block are freed (deleted) in the reverse order as they were pushed, and stack is, as we say, rewound — the top of the stack is again at the position it had before entering the block. This means that all variables declared inside a block are lost forever when we leave it: they are all *local*. The process of removing values from the stack is called 'popping off the stack'.
The address of the current top of the stack is always available at runtime: actually, there is a special register of the processor dedicated only to store information on the current location of the top of the stack.
The advantage of using the stack to store variables is that memory is managed for you automatically. We don't have to allocate memory by hand, or free it once we don't need it any more. Also, the CPU organizes stack memory very efficiently: reading from and writing to stack is very fast.

The heap is a region of memory that is *not* managed automatically. When the program asks the system to store some data on the heap, the system searches for a free space there, writes this data in this location, marks this region as occupied and return its address — this address can be stored in a variable of a reference type (which itself may be on the stack). Note that if the value of this variable is lost, for example, because it was a local variable in a block, the data on the heap it referred to is no longer available, as we have lost its address! In Java, such unavailable object on the heap may be eventually freed automatically by the process of the so called *garbage collector*.
Unlike the stack, variables created on the heap are accessible not only locally, but wherever their address is known. Heap memory is slower to be read from and written to, because one has to use pointers (which contain addresses) and follow them to access memory on the heap. Also, the process of allocating memory on the heap is rather complicated and expensive.

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

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*. Other examples:

```
      // shift operator
    a <<= 2;
    a = a << 2;

      // mod (remainder) operator
    a %= 7;
    a = a % 7;

      // bitwise ANDing
    a &= 0xFF;
    a = a & 0xFF;
```

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'. The value of `a` is incremented (decremented) and the value of the expression `++a` or `--a` will be equal to this already incremented or decremented value.

However, when a variable (say, `a`) is *post*incremented (or *post*decremented), the value of the variable `a` is also modified, but the value of the *expression* `a++` (or `a--`) is equal to the 'old', unmodified value of `a`.

For example, after

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

`c` will be 7, as on the right-hand side `a` has been incremented and the value of `++a` is equal to this incremented value (which is 6 in the example), while `b` is decremented, but the value of the expression `b--` is equal to the 'old', not decremented, value of `b` (i.e., still 1). However, after

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

`c` will be 6, as values of `++a` and `--b` will reflect 'new' values of these variables. 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 (those, that require two operands) are placed *between* operands they act upon.
Important: All binary arithmetic operations are always performed on

- two values of type **int**, or
- two values of type **long**, or
- two values of type **float**, or
- two values of type **double**,

and the result is the value of the same type as the type of operands.

In other cases, the values of one or both operands will be implicitly converted to match one of these cases. Therefore:

```
int i1 = 7;
char c1 = 'A', c2 = 'x';
short s1 = 3;
long l1 = 9;
double d1 = 3.5;

int    r1 = i1 + s1;  // short -> int
long   r2 = l1 + c1;  // char  -> long
double r3 = l1 + d1;  // long  -> double
int    r4 = c1 + c2;  // char  -> int (both operands)
int    r5 = c1 + s1;  // char  -> int, short -> int
```

For example, two values of type **byte** added together give **int**, *not* **byte**. Therefore,

```
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.

Everybody knows addition, subtraction, multiplication and division. 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)
19%6 is 1 (as 19 is 3*6+1)

etc.

Note that in mathematics remainders are never negative. In most programming languages, however, the following is assumed to be always true:

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

This, combined with the convention that integer division always truncates towards 0, leads to the following rule:

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
```

An important practical conclusion is to never use

```
if (a % 2 == 1) ...
```

to check, if `a` is odd (because if it's odd but negative, the remainder will be -1). Use rather

```
if (a % 2 != 0) ...
```

Instead of remembering all these rules about negative operands, it is better just not to use negative numbers in remainder operations.

Note also that modulus operator works also for floating point types; for example `7.75 % 0.5` is 0.25.

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 (de Morgan's law).

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 very 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

```
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):

```
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

```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!) -> ");

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

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

        var 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 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 (flips) the bits

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

Let us notice here, that XORing has an interesting and useful feature. Let us see the result of XORing a bit sequence `a` with all ones:

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

We see that XORing the value `a` with all ones gives negation of `a` — wherever there was 1 in `a`, we get 0, and *vice versa*.
Now let us try to XOR our `a` with all zeros:

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

This time what we got is exactly the same as `a`! So, XORing a bit with 1 flips its value, XORing it with zero – reproduces the same value.

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 Casting operator

Some values can be, in a natural way, converted to values of a different type. If such conversion does not lead to a loss of information, it can be performed by the compiler automatically (*implicit conversion*), as in

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

If the conversion is possible, but would lead to a loss of information, we have to enforce it explicitly

```
double x = 7.5;
int a = (int)x;
```

(see also sec. 3.4 on page 14). Of course, not every conversion makes sense, for example

```
double x = 7.5;
String s = (String)x; // WRONG
```

is impossible. Generally, conversions apply to numeric types; they also can be used to references, but we will postpone this discussion to the next chapters.

Unlike in many other languages, there are no conversions between numeric (and reference) types and logical values.

5.1.6 Precedence and associativity

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 not make any sense. If in doubt, just add parentheses to make your intentions clear (in this particular case, $<<$ is 'stronger').

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;        // WRONG
```

would not compile, because we are assigning uninitialized value of `a` to `c`.
The ternary (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";    // WRONG
```

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

```
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);
    }
}
```

The table below illustrates precedences and associativity of Java operators; → denotes associativity form left to right, while ← from right to left. Operators are shown in decreasing order of their precedence.

Table 1: Precedence of operators

Operator type	**Operators**
access (→)	[] () .
postfix	expr++ expr--
unary (←)	++expr --expr +expr −expr ∼ !
cast, object creation (←)	(type) new
multiplicative (→)	* / %
additive (→)	+ −
shift (→)	<< >> >>>
relational	< > <= >= instanceof
equality (→)	== !=

Table 1: Precedence of operators (cont.)

Operator type	**Operators**
bitwise AND (→)	&
bitwise XOR (→)	^
bitwise OR (→)	|
logical AND (→)	&&
logical OR (→)	||
ternary (←)	cond ? expr1 : expr2
assignment (←)	= += -= *= /= %= &= ^= |= <<= >>= >>>=

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 (probably very long) 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**, or *expression statements*.

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 and expression

There are two kinds of **switch** instructions in Java. First, let us consider the "classic" form of this instruction: **switch statement**. It 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 either of an integral type (but not **long**), or the reference to a **String**, or an enumeration constant (which we will introduce later). Symbols val stand for values with which the value of expr will be compared (in the specified order). They have to be literal values (not variables). If the value of expr is equal to any of vals, the code in the corresponding arm is executed and then execution continues (*falls through*) with all the arms 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 **default** arm, where we don't make any comparisons: this will be the arm 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(0-9 or a-z lower or upper case). Here, we take advantage of the "fall through" feature; note that if `ch` is any lower case letter from the range $[a, f]$, due to "falling through", we will end up in line 24 and then break out. The same applies to digits (see lines 26-27):

Listing 12 ACD-Switch/Switch.java

```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         System.out.print(
 7             "Enter a single hex digit -> ");
 8         char ch = scan.next().charAt(0);
 9         scan.close();
10 
11           // toLower by hand...
12         if (ch >= 'A' && ch <= 'Z')
13             ch = (char)(ch + 'a' - 'A');
14 
15         int num;
16 
17         switch (ch) {
18             case 'a':
19             case 'b':
20             case 'c':
21             case 'd':
22             case 'e':
23             case 'f':
24                 num = 10 + ch - 'a';
25                 break;
26             case '0': case '1': case '2': case '3': case '4':
27             case '5': case '6': case '7': case '8': case '9':
28                 num = ch - '0';
29                 break;
30             default:
```

```
                num = -1;
        }

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

Starting from Java 14, instead of the colon, one can use the "arrow", i.e., the two-character '->' symbol (but one cannot mix colons and arrows in one **switch** statement). Then

- the "fall through" feature is "turned off";
- to the right of the arrow, one can only use:
  1. single statement or expression,
  2. a block (enclosed in braces),
  3. a **throw** statement.

For example in the following example, we don't use any **break** statements:

Listing 13 ACE-SwitchArrow/SwitchArrow.java

```java
import java.util.Scanner;

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

        String part1 = k + " is of the form ";
        String part2 = null;

        switch (k % 4) {
            case 0 -> part2 = "4n";
            case 1 -> part2 = "4n+1";
            case 2 -> part2 = "4n+2";
            case 3 -> part2 = "4n+3";
            default-> {
                part2 = " ... probably negative";
                System.out.println("You were expected to " +
                        "enter a *natural* number!");
            }
        }
        System.out.println(part1 + part2);
    }
}
```

Also starting from the Java 14 version, one can supply more than one values (comma separated) in one arm of the switch

Listing 14 ACF-SwitchMult/SwitchMult.java

```java
public class SwitchMult {
    public static void main(String[] args) {
        String n = "BMW", country = null;
        switch (n) {
            case "BMW", "Opel", "Audi" -> country = "Germany";
            case "Peugeot", "Citroen"  -> country = "France";
            default                    -> country = "unknown";
        }
        System.out.println(n + " -> " + country);
    }
}
```

Another kind of switch is the so called *switch expression*. It is also available since Java 14. This is an *expression*, what means that the switch as the whole has a value. Therefore, each arm of the switch expression must itself be an expression, i.e., have a value (of the same type, or at least the types must be convertible to one common type). It can be just an expression, or a block ending with yield value – then the value indicated after the yield statement will be the value of the corresponding arm. As the switch expression must have a value, it has to be **exhaustive**, i.e., for all possible values there must be a matching switch label. For integral types and Strings the number of possible values is practically infinite, so there is no way to avoid the default arm, as in the example below:

Listing 15 ACG-SwitchExpr/SwitchExpr.java

```java
public class SwitchExpr {
    public static void main (String[] args) {
        int i = 7;
        var res = switch(i) {
           case  1,  2,  3 -> "First quarter";
           case  4,  5,  6 -> "Second quarter";
           case  7,  8,  9 -> {
               System.out.println("And here a block...");
               System.out.println("still we need a value...");
               yield "Third quarter";
           }
           case 10, 11, 12 -> "Fourth quarter";
               // must be exhaustive
           default -> "Something wrong";
        };
        System.out.println("res = " + res);
    }
}
```

We will talk about enumerations later, but here we will just mention that for them the number of values is usually small, so we just have to ensure that they all have been taken into account:

Listing 16 ACH-SwitchEnum/SwitchEnum.java

```java
public class SwitchEnum {
    enum Country {FRANCE, GERMANY, MEXICO, CANADA, CHINA};
    public static void main (String[] args) {
        Country c = Country.CANADA;

        var continent = switch(c) {
           case  FRANCE, GERMANY -> "Europe";
           case  MEXICO, CANADA  -> "America";
           case CHINA            -> "Asia";
        };
        System.out.println("Continent: " + continent);
    }
}
```

5.2.3 Loops

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

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

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).
The following loop will print square roots of integers read from the keyboard, but only of positive ones; the loop ends when the user enters 0:

```java
Scanner scan = new Scanner(System.in);
while (true) {
    int n = scan.nextInt();
    if (n == 0) braek;
```

```
        if (n < 0) {
            System.out.println("Negative - try again!");
            continue;
        }
        System.out.println("square root of " + n +
                " is " + Math.sqrt((double)(n)));
    }
```

Note that assignment is an expression — it has a value. Therefore, we could have rewritten the above snippet as

```
    Scanner scan = new Scanner(System.in);
    int n;
    while ((n = scan.nextInt()) != 0) {
        if (n < 0) {
            System.out.println("Negative - try again!");
            continue;
        }
        System.out.println("square root of " + n +
                " is " + Math.sqrt((double)(n)));
    }
```

In the following example, we read numbers and print the information if they are prime or not:

Listing 17 AFA-While/Prime.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

```
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 18 AFB-WhileBis/Prime.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();
    }
}
```

Suppose now that the user enters numbers until he/she enters zero; we want to find the sum of all these numbers and the number of them:

```
int sum = 0, num = 0, n;
while ( (n = scanner.nextInt()) != 0) {
    sum += n;
    ++num;
}
```

after the loop we have what we were intersted in in variables sum and num.

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;      // optional
      // ...
    if (cond2) continue;   // optional
      // ...
} 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.

Let us consider an example. We will use here the **random** from class **Math**. The expression `Math.random()` returns a (pseudo) random value from the half-open interval $[0, 1)$. This is sufficient to generate random values from any range, not necessarily $[0, 1)$. Suppose we want a random integer from the interval $[a, b]$. There are $b - a + 1$ consecutive values in this interval. Multiplying `Math.random()` by this number, we get a **double** from the half-open interval $[0, b - a + 1)$. Casting it to **int** will give us a number from the interval $[0, b - a]$ (we will never get $b - a + 1$, as the interval $[0, 1)$ was half-open). Adding a to the result, we get an **int** from the closed interval $[a, b]$, as desired. So, to generate a pseudo random integer from the closed interval $[a, b]$, we can write

```
a + (int)(Math.random()*(b-a+1))
```

It is important to use parentheses around the product `Math.random()*(b-a+1)` because casting has higher precedence than multiplication. Without the parentheses

```
a + (int)Math.random()*(b-a+1)   // WRONG
```

casting to **int** would be applied to `Math.random()` before multiplication, but this is always zero, as `Math.random()` is always less than 1.
The program below illustrates what we have just said demonstrating also the **do-while-loop**: here, we roll two dice until two sixes are thrown:

Listing 19 AFE-DoWhileDice/Dice.java

```java
public class Dice {
    public static void main (String[] args) {
        int a, b;
        do {
            a = 1 + (int)(Math.random()*6);
            b = 1 + (int)(Math.random()*6);
            System.out.println("a=" + a + " b=" + b);
        } while (a + b != 12);
    }
}
```

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
```

As we can see, the main difference between *while* and *do-while* loop is that in the first case the *condition* is checked before executing the body of the loop, while in the second case — after.

for loop

For loop looks like this:

```
for ( init ; condition ; incr ) {
      // ...
      // ...
}
```

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

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

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

The execution of a loop proceeds as follows:

1. The *init* part will be executed once only, before entering the loop. If it is a declaration of one or more variables of the same type, they will exist only within the body of the loop — they are not visible (in fact, they don't even exist) when the flow of control goes out of the loop.
2. The *condition* part will be evaluated next (if left empty, it will be assumed **true**). If it evaluates to **false**, the loop ends and the flow of control goes out of the loop. If it evaluates to **true** then the body of the loop is executed.
3. When execution of of the body of the loop is completed, the *inc* part is executed, and then we go back to item 2 (evaluation of the *condition*).

For example, the following snippet calculates and prints the sum of all natural numbers from the range $[1, 1000]$

```
int sum = 0;
for (int i = 1; i <= 1000; ++i) {
    sum += i;
}
System.out.println("Sum is " + sum);
```

In the example above, the braces could have been omitted, because the body of the loop consists of only one statement:

```
int sum = 0;
for (int i = 1; i <= 1000; ++i)  sum += i;
System.out.println("Sum is " + sum);
```

Note that we cannot declare sum in the *init* part of the loop

```
    for (int i = 1, sum = 0; i <= 1000; ++i)
        sum += i;
    System.out.println("Sum is " + sum);      // no 'sum' here!
```

because it would not exist after exiting the loop, so we would not be able to print its value.

In the following example we have 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 20 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, if the input is 9, the program prints

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

In the next example, we use while- and for-loops to calculate Euler's totient function $\varphi(n)$ — this is the number of positive integers up to a given integer n that are relatively prime to n (for example, $\varphi(10) = 4$, as there are 4 natural numbers that are smaller than 10 and relatively prime to it: 1, 3, 7, and 9.

Listing 21 AFJ-ForWhileEuler/ForWhileEuler.java

```java
import java.util.Scanner;
/*
```

```java
 * 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);
        }
    }
}
```

The program prints, for example,

```
$ Enter a natural number (0 to exit) -> 54
φ(54) = 18
$ Enter a natural number (0 to exit) -> 10
φ(10) = 4
$ Enter a natural number (0 to exit) -> 111222
φ(111222) = 35856
$ Enter a natural number (0 to exit) -> 0
$
```

Section 6

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).

6.1 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.
- Individual elements of an array can be accessed by their indices — if arr is the reference to an array, the expression arr[i] denotes its ith element; remember, however, that **indexing starts from zero!**
- 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's sometimes called a 'for-each loop' and 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 the consecutive element of the array.

For example, this is how we can create an 10-element array of **char**s and fill it with random capital Latin letters (note the keyword **new**). Then we print its elements, separated by spaces, in one line.

```
char[] arr = new char[10];
for (int i = 0; i < arr.length; ++i) {
    arr[i] = (char)('A' + (int)(Math.random()*26));
}
for (char c : arr)
    System.out.print(c + " ");
System.out.println();
```

Note that the condition in the **for**-loop is `i < arr.length` with strict inequality, as the index `arr.length` (i.e., 10) would be illegal: indexing starts with 0, so the last element has index 9. Note also the use of the for-each loop to print the elements.

This is how we can create an array and initialize it at the same time; note that we don't specify its size, because the compiler will be able to infer it from the initializer in braces; after creation, we print it in reverse order:

```
int[] arr = {1, 2, 8, 9};
for (int i = arr.length-1; i >= 0; --i)
    System.out.print(arr[i] + " ");
System.out.println();
```

Next example shows still another way to create and initialize an array:

```
int[] a0 = new int[]{1, 2, 3};
int[] a1 = {4, 5, 6};
int[] arrsum = new int[a0.length]; // contains zeros
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
```

The example below shows how to find the value of the maximum element of an array

```
int[] arr = {1, -6, 9, -2, 7};
int mx = arr[0];
for (int i = 1; i < arr.length; ++i)
    if (arr[i] > mx) mx = arr[i];
System.out.println("Maximum element is " + mx);
```

Sometimes what we want is rather the index of the maximum element:

```
int[] arr = {1, -6, 9, -2, 7};
int indmax = 0;
for (int i = 1; i < arr.length; ++i)
    if (arr[i] > arr[indmax]) indmax = i;
System.out.println("Maximum element has index " + indmax);
```

How to reverse the order of elements in an array `arr` of, say, **int**s? It's simple:

```
for (int i = 0, j = arr.length-1; i < j; ++i, --j) {
        int tmp = arr[i];  // swapping two elements
        arr[i] = arr[j];
        arr[j] = tmp;
}
```

Note that the condition `i < j` cannot be replaced by `i < arr.length` — we would then reverse the order of elements twice thus restoring the original order!
And now – how to randomly shuffle elements of an array?

```
int[] arr = {1, 2, 3, 4, 5, 6};
  // shuffling
for (int i = 0; i < arr.length-1; ++i) {
      // random index from interval [i, arr.length-1]
```

```
        int r = i + (int)((arr.length-i)*Math.random());
          // swapping
        int tmp = arr[i];
        arr[i] = arr[r];
        arr[r] = tmp;
    }
```

Let us consider the following example. Note that **printArr** is a function which just prints the content of an array passed to it as the first argument (just after the opening parenthesis); it will also print the string passed as the second argument. More about functions in the next section (Sec. 7, p. 51).

Listing 22 CYD-BasicArray/BasicArr.java

```
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");
    }


    /**
     *  prints an array of integer numbers
     */
    private static void printArr(int[] a, String message) {
        System.out.print(message + ": [");
        for (int i : a)
            System.out.print(" " + i); // <-- i unmodifiable
        System.out.println(" ]; size = " + a.length);
    }
}
```

The program prints

```
Array a1: [ 1 2 3 ]; size = 3
Array a2: [ 4 5 6 ]; size = 3
After a1=a2, a1 is: [ 4 5 6 ]; size = 3
After modifications a1 is: [ 44 5 66 ]; size = 3
After modifications a2 is: [ 44 5 66 ]; size = 3
Ad hoc array: [ 7 8 9 ]; size = 3
```

6.2 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**:

```
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();
    }
}
```

6.3 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*.

Note also, that all elements of a 'two-dimensional' array are references to 'normal' arrays of the same type, but *not* necessarily of the same length (as shown in the above example). Such arrays, where rows are of different lengths, are called **jagged arrays** (or 'ragged arrays'). However, very often they *do have* the same length: we call such 2D arrays **rectangular arrays**, as we can visualize them as 'rectangles' of elements (called *matrices* in mathematics):

```
int[][] arr = { {  1,  2,  3,  4},
                { 11, 22, 33, 44},
                { -1, -2, -3, -4} };
```

Individual arrays ({1,2,3,4}, {11,22,33,44} and {-1,-2,-3,-4} in this example) are called **rows** of this array — they correspond to arr[0], arr[1], arr[2]. Elements with the same *second* index are called **columns**: in the above example elements `arr[i][1]` where i=0, 1, 2, are the second (with index 1) column of the array (these are numbers 2, 22, -2).

In particular, the number of columns in a rectangular array may be equal to the number of rows — it is then a **square array**. For example, the following array is a square array 4×4:

```
int[][] squ = { {  1,  2,  3,  4},
                {  5,  6,  7,  8},
                {  9, 10, 11, 12},
                { 13, 14, 15, 16} };
```

For square arrays it makes sense to talk about its diagonals. The **main diagonal** (or just 'diagonal') consists of the elements for which the row and column indices are the same, or in other words these are elements on the diagonal going from the upper-left corner to the lower-right one (in the example above, these are elements 1, 6, 11, 16). The **antidiagonal** are elements for which the sum of row and column indices is fixed and equal to size-1, where size is the number of columns (equal to the number of rows). In other words, these are elements on the diagonal going from the upper-right corner to the lower-left one (in the example above, these are elements 4, 7, 10, 13).
Let us consider another example:

Listing 23 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)
```

```java
                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)
            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();
        }
    }
}
```

Note that the array `b` has rows of different lengths, so, in the loop which prints its elements, we have to use, in the inner loop, `col < b[row].length`.

Note also the array `f`. It is a 2D array, that is *one-dimensional* array of references to one-dimensional arrays. Therefore, in order to create it, we have to specify only number of its elements (that is number of 'rows'); all these elements have initially the value **null**, and their type is *reference to an array of* **int***s*. Of course, we can later assign to them references to any arrays of integers (of any lengths).
In the next example, we use a three-dimensional array of **int**s: the first index corresponds to a student, the second to a course he/she is taking and the third to the grades of this student and this course:

Listing 24 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
38                 for (int g = 0; g < grades[s][c].length;++g)
39                     ave += grades[s][c][g];
40             }
41             ave /= num;
42             if (ave > 4) pom[count++] = students[s];
43         }
44
45         String[] result = new String[count];
46         for (int i = 0; i < count; ++i)
47             result[i] = pom[i];
48
49         System.out.print("Best students of the group:");
50         for (String s : result)
51             System.out.print(" " + s);
52         System.out.println();
53     }
54 }
```

The variable grades is really a reference (pointer) to an array of references to two-dimensional arrays. Each element of the array grades corresponds to one student, so, for example, grades.length is 7. Therefore, grades[1] is the reference to a two-dimensional array corresponding to the second (as indexing starts from 0) student. This array is really an array of references to arrays of ints, each of which corresponds to one course. For example, grades[1][1] is the reference to the array of grades of the second student and of the second course (the one with elements $5, 2, 3, 3, 4$). Therefore, grades[1][1].length is 5. Finally, grades[1][1][4] is of type int and has value 4.

The loop in lines 31-43 is a nested loop: we iterate over students (index s) and for each student we calculate the average of his/her grades — in num we will collect number of grades and in ave the sum of grades. If the average of all this student's grades is grater than 4, we add his/her name to the array pom. This array is created before the loop; as we don't know in advance what the number of 'good' students will be, we create it with size grades.length and then, when we are done (line 45) and we know this number, we create another array, result of the correct size and copy elements from pom to results, which is then printed

```
Best students of the group: Jim Jack
```

In a similar way, having such a three-dimensional array, we could ask other questions, like

- find the index of the student with highest average of grades;
- find the index of the student with highest average of grades of the course number c;
- find the number of all students who got at least one grade 2;
- find the number of all students who never got a 2 grade;
- create an array containing averages of all grades for each student;
- and so on...

Section 7

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.
The definition of a static function is of the form

```
[access] 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 — we will be talking about them in Sec. 8, p. 59). 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.
- 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 of the appropriate type (as declared as the type of parameters) lying on the stack that can be associated with its parameters. In particular, we can use literals as arguments — copies of their values will be then pushed on the stack.
- All variables defined *inside* a function are *local* to this function — they are also located on the stack (in the example below, mx is such a local variable). This region of the program stack, associated with an invocation of a function, is called this function's **stack frame** or **activation record**. Note that names of local variables are arbitrary and unrelated to local variables of the same name appearing 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 'unwound' or '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.

An example of a rather trivial function would be

```
static double maxOf3(double a, double b, double c) {
    double mx = a;
    if (b > mx) mx = b;
    if (c > mx) mx = c;
    return mx;
}
```

and then, somewhere in **main** or another function

```
double u = 1, v = 2, w = 2;
// ...
double result = maxOf3(u, v+1, w-1);
```

As we can see, the arguments do not need to be variables — what matters are their *values*, *copies* of which will be put on the stack and will be available for the function under names `a`, `b` and `c` (and will 'disappear' when the function exits).

It is very important to realize how the arguments are passed to the function. For example, suppose we have a function

```
public static int fun(int a) {
    a = a + 99;
    return a;
}
```

Then, after (somewhere in another function)

```
int a = 1;
int res = fun(a);
```

the value of `res` will be 100, but the value of `a` will still be 1, as only the copy of its value was accessible to the function; this copy *was* modified, but it disappeared after the return anyway.

All this applies also to passing objects, for example arrays (which *are* objects.) There is one important fact that we have to remember, though. Variables declared as arrays (or variables of any other object type) are really pointers (in Java called *references*) to anonymous objects representing these arrays (or other objects). Their values are *addresses* of objects. This means, in particular, 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 (or another object) itself. This copy will be put on the stack and will disappear after the function returns, so modifying it usually doesn't make much sense. However, the value of this copy *is* the address of the original object (e.g., of an array); consequently, having this address, functions which receive it *can* modify the original object (as they 'know where it is').
Examples can be found in the following program:

Listing 25 CYS-PassArr/PassArr.java

```java
public class PassArr {

    public static void main(String[] args) {
          // array returned from a function
        int[] a = getArr(5);
        printArr(a, "Array returned from function");

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

    /**
     *  prints an array of integer numbers
     */
    private static void printArr(int[] a, String message) {
        System.out.print(message + ":\n    [");
        for (int i : a) System.out.print("  " + i);
        System.out.println(" ]; size = " + a.length);
    }

    /**
     *  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;
    }

    /**
     *  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;
        }
    }
}
```

Let us now 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 invoking, for each number from the range, the first one.

Listing 26 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);
    }
}
```

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 27 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**!)

The next example illustrates the well known algorithm of sorting arrays, the so called *selection sort*. The idea is simple: for each consecutive element of the array find the smallest element in the portion of the array to the right of the element chosen and, if it is smaller than this element, just swap these two elements. We can implement this trivial algorithm both iteratively or recursively (usually the iterative version will be more efficient):

Listing 28 AFR-SelSort/SelSort.java

```java
public class SelSort {
    private static void selSortIte(int[] a) {
        for (int i = 0; i < a.length-1; ++i) {
            int indmin = i;
            for (int j = i+1; j < a.length; ++j)
                if (a[j] < a[indmin]) indmin = j;
            int temp = a[i];
            a[i] = a[indmin];
            a[indmin] = temp;
        }
    }

    private static void selSortRec(int[] a, int from) {
        if (from == a.length-1) return;
        int indmin = from;
        for (int i = from+1; i < a.length; ++i)
            if (a[i] < a[indmin]) indmin = i;
        int temp = a[from];
        a[from] = a[indmin];
        a[indmin] = temp;
        selSortRec(a, from+1);
    }

    private static void printArray(int[] a) {
        System.out.print("[");
        for (int i = 0; i < a.length; ++i)
            System.out.print(" " + a[i]);
        System.out.println(" ]");
    }

    public static void main (String[] args) {
        int[] a1 = {4, 8, -3, 5, 2, 8, 4, 6};
        int[] a2 = {4, 8, -3, 5, 2, 8, 4, 6};
        selSortIte(a1);
        printArray(a1);
        selSortRec(a2, 0);
        printArray(a2);
    }
}
```

Functions — both static and non-static — can be **overloaded**. This means that in one class we can define several functions with the same name. When an invocation of one of these functions is encountered in the program, the compiler must know which one of them is meant: we say that it has to *resolve the method call*. For this to be possible, the overloaded functions have to differ in number and/or types of parameters in such a way, that just by 'looking' at the arguments of an invocation, the compiler can determine unambiguously the method to be called.

In the example below, there are three functions with the name **sum**.

Listing 29 AFM-Overload/Overload.java

```java
public class Overload {
    static int sum(int a, int b) {
        return a + b;
    }
    static int sum(int a, int b, int c) {
        return a + b + c;
    }
    static String sum(String s, int a) {
        return s + a;
    }
    public static void main (String[] args) {
        int a = 4, b = 3, c = 5;
        String s = "a=";
        System.out.println(
                "Sum of   a,b: " + sum(a, b) +
              "\nSum of a,b,c: " + sum(a, b, c) +
              "\nSum of   s,a: " + sum(s,a));
    }
}
```

The first two overloads differ in number of parameters/arguments. The first and the third have both two parameters, but the first of these parameters is of different type (**int** and **String**, respectively) Therefore, the compiler will always know which function to call just by 'looking' at number and types of arguments. The program prints

```
Sum of   a,b: 7
Sum of a,b,c: 12
Sum of   s,a: a=4
```

The types of an argument and its corresponding parameter do not have to match exactly. The compiler considers all overloads and looks for the best match. The first criterion is simple: the number of arguments must be equal to the number of parameters (we disregard the so called *var-args* here). Then types are analyzed. If there is no perfect match, the compiler tries to find one by *widening* the type of arguments: for example, the value of an argument of type **byte** may be widened to **short**, **short** to **int** etc. This is illustrated by the following program, which defines three overloads of function **f**: with the parameter of type **short**, **int** and **double**.

Listing 30 AFN-Resol/Resol.java

```java
public class Resol {
    static void f(short a)  {System.out.println("s=" + a);}
    static void f(int a)    {System.out.println("i=" + a);}
    static void f(double a) {System.out.println("d=" + a);}

    public static void main (String[] args) {
        byte   ab =  65;
        char   ac = 'A';
        short  as =  65;
        int    ai =  65;
        long   al = 65L;
        double ad = 65D;
        System.out.print("byte   -> "); f(ab);
        System.out.print("char   -> "); f(ac);
        System.out.print("short  -> "); f(as);
        System.out.print("int    -> "); f(ai);
        System.out.print("long   -> "); f(al);
        System.out.print("double -> "); f(ad);
    }
}
```

which prints

```
byte   -> s=65
char   -> i=65
short  -> s=65
int    -> i=65
long   -> d=65.0
double -> d=65.0
```

As one can see, **byte** has been widened to **short** (as there is no overload for **byte**s). There is no overload for **char**s either, so **char** has been widened to **int**. But not to **short**! Both types have the same size, but **char** is unsigned, so there are values of this type that are not representable as **short**s. Note also that, as there is no overload for **long**s, it was widened to **double**, although it changes the type from integral to floating point.

Precise rules of method call resolution are a bit more complicated, as they have to take into account also inheritance, the so called boxing/unboxing, *var-args*, etc. — we will cover these topics later.

Section 8

Classes

8.1 Basic concepts

In object oriented programming we deal with **objects** that we can think of as aggregates of some pieces of information together with actions which can be performed on them. Similar objects (sharing the same type of information and the same actions) are described by the definition of a **class** which encapsulates properties of all object of this class that will be created in our program. Each object of a class can contain some information in the form of **fields** — fields have fixed types and names and will be present in any object of the class; of course their values are usually different in different objects. All these values, which can be subject to modifications during the execution of a program, define current **state** of an object. Values of fields in an object which are accessible to the user, at least for reading, are sometimes called its **attributes**.

Actions are represented by functions (called **methods**) which operate on objects — methods can return information about the state of an object, modify it, etc. They are always invoked on a specific object and have direct access to all fields of this particular object they were invoked on, and also to all fields of *any* object of the same class. Invoking a method on an object is sometimes described as sending a message to this object.

Fields and methods of a class are collectively called its **members**, or, strictly speaking, its non-static members; there are also static members: static fields and static functions (methods).

8.2 Classes and objects

Classes define new types. A class can contain

- fields (non-static);
- static fields;
- constructors;
- methods (non-static);
- static functions (methods);
- static and non-static initializer blocks;
- definitions of other classes (the so called *inner* classes) or enumerations.

Objects (instances) of classes are always created on the heap (in *free memory*). They are always anonymous — there is no way to give a name to an object. It is also not possible to create an object locally on the stack — only references can be local and have names. Operator **new** creates an object and returns a *reference* (which in C/C++ corresponds to a pointer, i.e., address) to the object created. We can store this reference (address) in a named reference variable. If there are no references to an object left, the object may be removed by the garbage collector sub-process of the JVM (although we don't know if and when it will happen).

In the example below, we define a new type (class) **TrivPoint**, which is supposed to describe points on a plane:

- `x` and `y` are (non-static) fields. Each object of the class will contains two such **int**s — of course their values may be different for each object (point).

- **translate**, **scale**, **getX**, **getY**, **setX**, **setY** and **info** are (non-static) methods;
- **infoStatic** is a static function (static method).

There are no static fields here. Also, there is no constructor defined, but in fact there is one, called *default constructor*, created by the compiler (more about constructors in a moment).

Listing 31 BGO-TrivPoint/TrivPoint.java

```java
public class TrivPoint {
    public int x, y;

    public void translate(int dx,int dy) {
        x += dx;
        y += dy;
    }

    public void scale(int sx,int sy) {
        x *= sx;
        y *= sy;
    }

      // setters
    public void setX(int x) {
        this.x = x;      // this required
    }
    public void setY(int yy) {
        y = yy;          // this.y assumed
    }

      // getters
    public int getX() {
        return this.x;
    }
    public int getY() {
        return y;        // this assumed
    }

      // static
    public static void infoStatic(TrivPoint p) {
        System.out.println("[" + p.x + "," + p.y + "]");
    }

      // non-static
    public void info() {
        System.out.println("[" + this.x + "," + y + "]");
    }
}
```

The class **Main** (below) by itself doesn't serve any purpose — it is just a wrapper for the **main** function which must be defined (with a signature exactly as shown) somewhere and is the entry point to any Java application. In **main** we create an object of type **TrivPoint**; it will contain fields x and y — we could have created several objects of this type: each would contain 'its own' fields x and y, independent of x and y of any other object of the same class. Note the syntax used to create an object: we have to write round parentheses, as when calling a function. In parentheses, we can pass arguments to a constructor. In our case, there is only the default constructor (by definition, it doesn't take any arguments) created by the compiler, as we haven't defined any custom constructor ourselves. Note that p is *not* the name of any object; it is the name of a separate *local* reference variable whose value is the *address* of the object proper (which itself is anonymous). In C++ we would call such a variable a pointer; in Java it is called a *reference* (although is has nothing in common with references in C++.)

Listing 32 BGO-TrivPoint/Main.java

```
 1  public class Main {
 2      public static void main(String[] args) {
 3          TrivPoint p = new TrivPoint();
 4          p.x = 1;
 5          p.y = 2;
 6          p.info();
 7          p.setX(3);
 8          p.info();
 9          System.out.println("x=" + p.getX() + "; " +
10                             "y=" + p.getY());
11          TrivPoint.infoStatic(p);
12          p.infoStatic(p);         // not recommended!
13
14          p.scale(2,3);
15          p.info();
16          p.translate(1,-3);
17          p.info();
18      }
19  }
```

Let us briefly explain the difference between static functions and methods (non-static). The method **scale** seems to have two parameters. However, it's a *method* (there is no **static** keyword in its declaration). This means that it has one additional parameter, not shown in the list of parameters (as it would be, e.g., in Python). This 'hidden' parameter is of type **TrivPoint**, i.e., it is a reference to an object of this type. Therefore, **scale** has in fact *three* parameters and hence, invoking it, we have to specify three arguments. Let's look at its invocation in line 10

```
    p.scale(2, 3);
```

Two arguments are given explicitly and they correspond to parameters sx and sy of the method. What about the third? It will be the reference to the object on which the method is invoked, in our case the value of the variable p which holds the reference (address) to the object created in line 3. So, conceptually, the invocation is equivalent to something like

```
    TrivPoint::scale(p, 2, 3);
```

Therefore, methods are always called on a specific object (which, of course, must exist). The reference to this object is passed (pushed on stack) to the method as one of the arguments. Inside the body of a method we can refer to it — but what is its name there? As it was not mentioned in the list of parameters, we were not able to give it any name. Therefore, the name is fixed once for all and is **this**.

Now look at the definition of the method **scale**. We use the name `x` there. There are variables `sx` and `sy` declared there, but no `x`, so compilation should crash. In such situation, however, the compiler will automatically add **this** in front of undeclared variable (`x` in this case) to obtain `this.x`. But this means 'field `x` of the object referenced to by **this**' — this is exactly the field `x` in the object the method was invoked on (i.e., the one referenced to by `p` in the **main** function). The same, of course, applies to `y`.

The situation is different for static functions. Here, we don't have any hidden parameters (therefore **this** doesn't exist inside static functions). If a static function is to have access to an object of type **TrivPoint**, the reference to this object must be passed explicitly — as in function **infoStatic** in our example.

More details and more examples will be presented in the following sections.

8.3 Access to classes and their members

Generally, fields of a class should be somehow protected from unrestricted access from other classes (we call it **hermetization**, or **encapsulation**). They should be manipulated only by methods, which can ensure that they are operated upon in a safe and consistent way. We have control over accessibility of members by adding, at the beginning of their declaration, an appropriate keyword: **private**, **protected**, **public** or, not specifying any of these, we get the default. They have the following meaning:

- default (or **package private**) — no special keyword — the field can be accessed from functions (static or non-static) of the same class and also from functions of all classes belonging to the same package;
- **private** — the field can be accessed *only* from functions of the same class;
- **protected** — the field can be accessed from functions of the same class, form classes in the same package, and also from classes which extend this class (inherit from it) even if they belong to a different package (inheritance will be covered later);
- **public** — the field can be accessed from functions of all classes where our class is visible.

The same rules apply to all members: also constructors and functions. It is recommended to declare all *fields* (data members) as **private** (or **protected**). But then a problem arises: if we do not provide public methods which modify the fields, it will never be possible to assign any useful value to them! Of course, there is a way to do it — by defining *constructors* (see the next section).

We can also declare whole classes as public or having default accessibility. If a class is declared with the default accessibility, its name will be visible only in other classes, but *only* those from the same package. There also exist the so called **inner classes**, i.e., classes defined inside definition of other classes, which will be covered later. Such classes may also be declared as **private** or **protected**.

The 'entry' class, the one which contains **main**, must always be declared as **public** (and the **main** function itself must also be **public**).

8.4 *Constructors and methods*

So let us make the fields of a class **private**. For example:

Listing 33 BGI-VerySimple/VerySimple.java

```java
public class VerySimple {
    private int    age;
    private String name;

      // constructor
    public VerySimple(int age, String n) {
        this.age = age;
        name = n;
    }
      //getter
    public int getAge()  {
        return age;
    }
      // setter
    void setAge(int a) { // package private
        age = a;
    }
      // getter (with no corresponding setter)
    public String getName() {
        return name;
    }
}
```

and in **Main** we can create objects of this class, and even modify them (because both constructor and **setAge** are *not* **private** or **protected**. Of course, in our **main** function, which is a function of another class, we cannot directly access the fields of class **VerySimple**, but we can use public constructors and methods, which, being members of the class **VerySimple**, *do* have access to all members *of all* objects of this class. In particular, **constructor** is a very special function:

- its name must be the same as the name of the class;
- it does not declare any return type (even **void**);
- it will be automatically invoked *only* at the very end of the process of constructing an object — there is no way to invoke it on an object which has already been created before.

Normally, constructor are used to initialize fields of the object being created, but generally they can do whatever we wish.

Listing 34 BGI-VerySimple/Main.java

```java
public class Main {
    public static void main(String[] args) {
        VerySimple alice = new VerySimple(23,"Alice");
        VerySimple bob   = new VerySimple(21,"Bob");
        alice.setAge(18);

        System.out.println(
                alice.getName()  + " " + alice.getAge());
        System.out.println(
                bob.getName()    + " " + bob.getAge());
    }
}
```

To use a specific constructor during creation of an object, we just write, after the name of the class, arguments — as many of them as expected by the constructor, and of appropriate types. The program, as the previous one, prints

```
Alice 18
Bob 21
```

In a class, one can define many methods or constructors with the same name (in case of constructors there is no way to avoid it, as *all* constructors have to be named as the class they are defined in). This is called **overloading**. However, overloaded functions must differ in number of parameters and/or these parameters' types, so when they are invoked, the compiler knows (by looking at arguments) which one is meant. The precise rules used by the compiler to resolve which method or constructor should be used in a given context are rather complicated. However, there should be no problem if the differences are sufficiently obvious (different number of parameters, difference in types like between String and int etc.).

If we don't define any constructor, the compiler will add one — parameterless and doing nothing. Then all fields will be initialized with their default values: zero for numeric types, null for references, false for booleans. A parameterless constructor, whether it is created by the compiler or defined by ourselves, is called **default constructor**. However, the compiler will *not* create any default constructor if there is at least one constructor in a class defined by us.

When a method is invoked, it must always be invoked on an object (in the example, it was the object pointed to by reference alice). As we have already mentioned, the reference (address) to this object is passed to the function as an additional, 'hidden' argument. As it is not mentioned on the parameter list of the function, its name is once for all fixed: this. Inside the function, when we refer to a name (e.g., age) without specifying of which object it is a member, it is understood that it is this.age that is meant.

Any constructor can invoke — *but only in its first line* — another constructor of the same class using this with arguments, as if it were the name of a method. Such a constructor is said to be a delegating constructor. We can see an example in the following program:

Listing 35 BGP-Point/Point.java

```java
public class Point {
    private int x, y;

    public Point(int x, int yy) {
        System.out.println("Point(int,int) with " +
                              x + " and " + y);
        this.x =  x;
        y      = yy;
    }

    public Point(int x) {
        this(x,0);
        System.out.println("Point(int) with " + x);
    }

    public Point() {
        this(0);
        System.out.println("Point()");
    }

    public Point translate(int dx,int dy) {
        x += dx;
        y += dy;
        return this;
    }

    public Point scale(int sx,int sy) {
        x *= sx;
        y *= sy;
        return this;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    /**/
    @Override
    public String toString() {
        return "[" + x + "," + y + "]";
    }
    /**/

    public static void main(String[] args) {
        System.out.println("\n*** Creating point p1 (1,2)");
        Point p1 = new Point(1,2);

        System.out.println("\n*** Creating point p2 (1)");
        Point p2 = new Point(1);

        System.out.println("\n*** Creating point p3 ()");
        Point p3 = new Point();

        p3.translate(4,4).scale(2,3).translate(-1,-5);

        System.out.println("\np1: [" + p1.getX() + "," +
                                         p1.getY() + "]");

        System.out.println("\np1: " + p1 + "  p2: " + +
                                    p2 + "  p3: " + p3);
    }
}
```

After executing another constructor, the control flow returns to the invoking (delegating) constructor. It can be seen from the output of the above program

```
*** Creating point p1 (1,2)
Point(int,int) with 1 and 0

*** Creating point p2 (1)
Point(int,int) with 1 and 0
Point(int) with 1

*** Creating point p3 ()
Point(int,int) with 0 and 0
Point(int) with 0
Point()

p1: [1,2]

p1: [1,2]  p2: [1,0]  p3: [7,7]
```

This program illustrates a special rôle of the **toString** method. It is defined in class **Object**. As we will learn later, any class *inherits* (is an extension) of this special class. Usually, in our classes we **override** (redefine) this method — it is then called automatically if an object is provided, but a string describing it is needed (for example, it will be called automatically by function **println**.) As it exists in all classes, either because we redefined it or, if not, we inherited it from class **Object**, the method **toString** can be safely invoked on any object. The version from class **Object** doesn't return a very useful string, but, what is important, it *does* exists and returns a string. Another example

Listing 36 BGR-BasicClass/Person.java

```java
public class Person {

    private String name;
    private int birth_year;

    public Person(String name, int r) {
        this.name = name;
        birth_year = r;
    }

    public String getName() {
        return name;
    }

    public int getYear() {
        return birth_year;
    }
    /**/
    @Override
    public String toString() {
        return name + " (b. " + birth_year + ")";
    }
    /**/

    public boolean isOlderThan(Person other) {
        return birth_year < other.birth_year;
    }
}
```

and the class **Main** which uses **Person**:

Listing 37 BGR-BasicClass/Main.java

```java
import javax.swing.JOptionPane;

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

        Person john = new Person("John", 1980);
        Person mary = new Person("Mary", 1985);

        System.out.println(
            "Two Persons created: " +john + " and " + mary);

        Person older = mary.isOlderThan(john) ? mary : john;
        System.out.println("Older: " + older.getName() +
                " born in " + older.getYear());

        String s = older + " is older";

        // JOptionPane.showMessageDialog(null,s);
        System.out.println(s);
        System.exit(0);
    }
}
```

The program prints

```
Older: John born in 1980
John (b. 1980) is older
```

8.5 Static members

Any class can also declare **static** members — both fields (data) and methods (but *not* static constructors!) We can imagine a static members as belonging the class as a whole, not to objects. As such, they can be used even if we haven't created any object of our class — they are brought into existence and initialized when the class is loaded by the JVM (if the class happens to be an enum, *after* all the enum values have been created).

Inside a static function there is no **this**, which normally exists and is the reference to the object the function was invoked on — static functions are *not* invoked on any object and consequently cannot use **this**. From the outside of a class, we refer to its static members using just the name of the class. We *can*, although it's very confusing, use reference to *any* object of this class for the same purpose. For example, in the last line of

```java
public class AnyClass {
    public static int stat;
    // ...
}

// somewhere else

AnyClass.stat = 7;
AnyClass a = new AnyClass();
// ...
a.stat = 28;
```

we refer to static member stat just by putting a before the dot, but the compiler will use only the information about the type of a — which particular object of class **AnyClass** is used in this context is completely irrelevant. Looking at this line alone, one is not able to tell whether stat is a static or non-static member of the class; for this reason it is recommended to always use the first syntax, with the name of a class, as here, it is obvious that stat must be static.

In the following example:

Listing 38 BHE-StatEx/StatExample.java

```java
public class StatExample {
    private static double rate =  1;
    private static char     ID = 'A';

    private double amount;
    private char       id;

    public static   void setRate(double r) { rate = r; }
    public static double getRate() { return rate; }

    StatExample(double a) {
        id = ID++;
        amount = a;
    }

    @Override
    public String toString() {
        return "I'm " + id + ", I have $" + amount +
            " = " + rate*amount + " PLN";
    }

    public static void main (String[] args) {
       StatExample.setRate(4.1);
       StatExample sa = new StatExample(10);
       StatExample sb = new StatExample(16);
       StatExample sc = new StatExample(20);
       System.out.println(sc + "\n" + sb + "\n" + sa);
    }
}
```

ID is a static member whose value is assigned in the constructor to non-static field id and then incremented by one; in this way each object will get its unique id. Also rate is static, as it represents a rate which is common for all objects and used by them to convert one currency into another. Consequently, function **setRate** is static, because it only affects rate and does not even need any object of the class to exist; as we can see, it is invoked before creating any objects. The program prints

```
I'm C, I have $20.0 = 82.0 PLN
I'm B, I have $16.0 = 65.6 PLN
I'm A, I have $10.0 = 41.0 PLN
```

Both methods and static functions can be recursive, i.e., they can invoke themselves; of course, you have to ensure that the chain of recursive invocations stops at some point…

Listing 39 AFL-FunRecur/SimpleRec.java

```
public class SimpleRec {

      // should be called with from=0
    static void printArrRec(int[] arr, int from) {
        if (from == arr.length) {
            System.out.println();
            return;
        }
          // first print then invoke next
        System.out.print(arr[from] + " ");
        printArrRec(arr,from+1);
    }

      // should be called with from=0
    static void printArrRecReverse(int[] arr, int from) {
        if (from == arr.length) return;
          // first invoke next then print
        printArrRecReverse(arr,from+1);
        System.out.print(arr[from] + " ");

```

```java
        if (from == 0) System.out.println();
    }

      // should be called with from=0, to=arr.length
    static void revArrayRec(int[] arr, int from, int to) {
        if (to-from <= 1) return;
        int temp = arr[from];
        arr[from] = arr[to-1];
        arr[to-1] = temp;
        revArrayRec(arr,from+1,to-1);
    }

      // should be called with from=0
    static int maxElemRec(int[] arr, int from) {
        if (from == arr.length-1) return arr[arr.length-1];
        return Math.max(arr[from],maxElemRec(arr,from+1));
    }

      // should be called with from=0
    static void selSortRec(int[] arr, int from) {
        if (from == arr.length-1) return;
        int indmin = from;
        for (int i = from+1; i < arr.length; ++i)
            if (arr[i] < arr[indmin]) indmin = i;
        int temp = arr[from];
        arr[from] = arr[indmin];
        arr[indmin] = temp;
        selSortRec(arr,from+1);
    }

    static int gcd(int a, int b) {
        return  b == 0 ? a : gcd(b, a%b);
    }

    static int fact(int n) {
        return  n <= 1 ? 1 : n*fact(n-1);
    }

      // must be called with k=2
    static boolean isPrime(int n, int k) {
        if (k*k > n)  return true;
        if (n%k == 0) return false;
        return isPrime(n,k+1);
    }

    static long counter = 0;
      // stupid way of calculating Fibonacci numbers
    static long fibo(int n) {
        ++counter;
        return n <= 1 ? (long)n : fibo(n-2) + fibo(n-1);
    }

    public static void main(String[] args) {
          // Arrays
        int[] a = {13,3,55,7,9,11};
        printArrRec(a,0);
        revArrayRec(a,0,a.length);
        printArrRec(a,0);
        selSortRec(a,0);
        printArrRec(a,0);
        printArrRecReverse(a,0);
        System.out.println("Max. in a: "+maxElemRec(a,0));

          // GCD
        System.out.println("Greatest common divisor of" +
                " 5593 and 11067 is " + gcd(5593,11067));

          // Factorials
        System.out.println("10! = " + fact(10));
        System.out.println("12! = " + fact(12));
          // NO ERROR BUT WRONG!!!
        System.out.println("13! = " + fact(13) + " WRONG!");

          // Primes
        System.out.println("Primes up to 100");
        for (int n = 2; n <=100; ++n)
            if (isPrime(n,2)) System.out.print(n+" ");
        System.out.println();

          // Fibonacci numbers
        for (int n = 40; n <= 46; n += 2) {
            counter = 0;
            long r = fibo(n);
            System.out.println("Fibo(" + n + ") = " + r +
                               "; counter = " + counter);
        }
    }
}
```

The program prints

```
13 3 55 7 9 11
11 9 7 55 3 13
3 7 9 11 13 55
55 13 11 9 7 3
Max. in a: 55
Greatest common divisor of 5593 and 11067 is 119
10! = 3628800
12! = 479001600
13! = 1932053504 WRONG!
```

```
Primes up to 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Fibo(40) = 102334155; counter = 331160281
Fibo(42) = 267914296; counter = 866988873
Fibo(44) = 701408733; counter = 2269806339
Fibo(46) = 1836311903; counter = 5942430145
```

8.6 Initializing blocks

Inside the definition of a class, we can put static and non-static initialization blocks. They have the form of a block of code enclosed in curly braces; in case of static initializer block we add keyword `static` if front of it. The block must be put outside of any functions!

The body of the non-static initializer block will be executed in the process of creating any object after the object itself has been created and after its members have been initialized with default values but *before* entering a constructor. Therefore, we can put into the initializer a code which should be executed at the beginning of any of the overloaded constructors.

Static initializer block is executed only once: when a class is loaded by the JVM (after creating enum constants, if the class happens to be an enum.) If a class contains also static fields, initialization of static fields comes first, then initializer blocks are executed in the order they appear in the definition. In a static block we can initialize fields whose proper initialization requires some code to be executed.

The order of initialization can be seen from the output of the following program:

Listing 40 BHG-StatOrd/Stats.java

```java
public class Stats {
    private static int sino;
    private static int siyes = 2;
    private static final int fin;

    {           // nonstatic initialization block
        show("nonstatic init");
        sino = 1;
    }

    static {    // static initialization block
        show("   static init");
        fin = 3;
    }

    public Stats() {
        show("   constructor");
    }

    private static void show(String mes) {
        System.out.println(mes + ":" +
                " sino=" + sino +
                " siyes=" + siyes +
                " fin=" + fin);
    }
}
```

with main in class **Main**:

Listing 41 BHG-StatOrd/Main.java

```java
public class Main {
    public static void main(String[] args) {
        Stats e = new Stats();
    }
}
```

which outputs

```
       static init: sino=0 siyes=2 fin=0
    nonstatic init: sino=0 siyes=2 fin=3
       constructor: sino=1 siyes=2 fin=3
```

In another, a little more complicated example

Listing 42 BHF-StatBlocks/StatBlocks.java

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.Locale;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.stream.Collectors;
import javax.swing.JOptionPane;

import static java.nio.charset.StandardCharsets.UTF_8;

public class StatBlocks {
    static private double rateUSD = 1;
    static private double rateUAH = 1;

      // static initializer block
    static {
        try {
            URL nbp = new URL(
                "http://www.nbp.pl/kursy/xml/LastA.xml");
            BufferedReader bw =
                new BufferedReader(
                    new InputStreamReader(
                        nbp.openStream(),UTF_8));
            String txt =
                bw.lines().collect(Collectors.joining(" "));
            bw.close();
            Matcher m =
                Pattern.compile(".*USD.*?(\\d+,\\d+)" +
                                ".*UAH.*?(\\d+,\\d+).*")
                       .matcher(txt);
            m.matches();
            rateUSD = Double.parseDouble(
                             m.group(1).replace(",","."));
            rateUAH = Double.parseDouble(
                             m.group(2).replace(",","."));
        } catch (Exception e) {
            if (JOptionPane.showConfirmDialog(
                    null, "Fetching data failed; continue" +
                    " anyway with default rates = 1?",
                    "WARNING! FETCHING DATA FAILED!",
                    JOptionPane.YES_NO_OPTION
                    ) != JOptionPane.YES_OPTION
                ) System.exit(1);
        }
    }

    private static int ID = 1;

    private final int id;

      // non-static initializer block
    {
        if (ID % 100 == 13) ++ID;
        id = ID++;
    }

    private double amount;

    public StatBlocks(double a) { amount = a; }
    public StatBlocks()         { this(20);   }

    public static double getRateUSD() { return rateUSD; }
    public static double getRateUAH() { return rateUAH; }

    @Override
    public String toString() {
        return String.format(Locale.US,
                "My id = %d. I have %5.2f " +
                "PLN = $%5.2f = %6.2f UAH",id,
                amount, amount/rateUSD,amount/rateUAH);
    }

    public static void main (String[] args)
                          throws Exception {
        double rUSD = StatBlocks.getRateUSD();
```

```java
        System.out.println("Current rates are:\n" +
            "    USD/PLN = " + rateUSD + '\n' +
            "    UAH/PLN = " + rateUAH);
        StatBlocks sa = new StatBlocks();
        StatBlocks sb = new StatBlocks(40);
        StatBlocks sc = new StatBlocks(80);
        System.out.println(sc + "\n" + sb + "\n" + sa);
    }
}
```

initializing static fields rateUSD and rateUAH requires getting connection with the site of the National Bank of Poland and fetching the current rates. It is done in static initializer block. If fetching rates from the internet fails for some reason, we display a warning asking the user if he/she wants to continue with all rates set to default values 1. As in the previous example, each created object gets unique identifier id; this time we do it in non-static initializing block. Here, to avoid identifiers ending with 13, what might attract a disaster, we skip such numbers, what requires some code to be executed. As we do it in the initializer block, we don't have to repeat the same code in all constructors, what could be easily forgotten about when adding other constructors. The program prints something like (the result depends on current rates):

```
Current rates are:
    USD/PLN = 4.5066
    UAH/PLN = 0.1258
My id = 3. I have 80.00 PLN = $17.75 = 635.93 UAH
My id = 2. I have 40.00 PLN = $ 8.88 = 317.97 UAH
My id = 1. I have 20.00 PLN = $ 4.44 = 158.98 UAH
```

8.7 Singleton classes

Very often we encounter the situation when we have a class of which at most one object should ever be created; such objects are called **singletons**. There are at least two approaches to produce such objects. If the object in question is expensive to create and it is possible that it will not be needed at all, we can use lazy evaluation approache:

Listing 43 BGX-Singlet/Connect.java

```java
public class Connect {

    private static Connect connection = null;

    private Connect()
    { } // private - no one can create any other object

    public static Connect getInstance() {
          // lazy evaluation, no multithreading
        if(connection == null){
            connection = new Connect();
        }
        return connection;
    }
}
```

Or, if we know that such an object almost surely will be required, one can use eager evaluation:

Listing 44 BGX-Singlet/Config.java

```java
public class Config {
      // eager evaluation
    private final static Config config = new Config();

    private Config() {  // no one can create another object
        // ...
    }

    public static Config getInstance() {
        return config;
    }
}
```

The program convinces us that there is only *one* object of each of the classes. Note, however, that this will be a lot more complicated in the face of multithreading!

Listing 45 BGX-Singlet/Main.java

```java
public class Main {
    public static void main(String[] args) {
        Connect con1 = Connect.getInstance();
        Connect con2 = Connect.getInstance();
        if (con1 == con2) System.out.println("con1==con2");
        Config cnf1 = Config.getInstance();
        Config cnf2 = Config.getInstance();
        if (cnf1 == cnf2) System.out.println("cnf1==cnf2");
    }
}
```

and the output is

```
con1==con2
cnf1==cnf2
```

Section 9

Basic data structures

Computer programs are all about processing information. The amount of information that has to be handled is very often really huge and grows rapidly from year to year. Therefore, it is very important to organize the data in such a way that processing it is fast and efficient. Generations of scientists have been working (and are still working) on that problem. Generally, designing a way in which the data is structurized is related to the way in which it will be used; both aspects are studied by the branch of computer science called Algorithms and Data Structures.

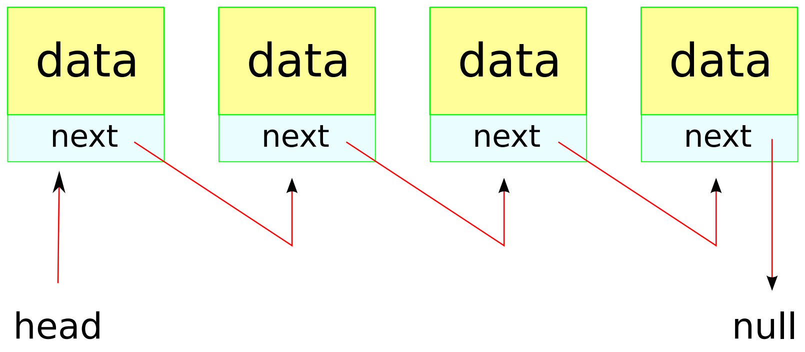
One important, and very useful, data structure is already known to us: arrays. Representing data in the form of arrays has many advantages, perhaps the most important being the speed of access to individual elements (using indices), which is "immediate" ($\mathcal{O}(1)$) and does not depend on the size of the array. The downside, however, is the fact that the size of arrays must be fixed at their creation and no elements can be then "deleted" or added.

In this section, we will mention just a few most fundamental data structures based on lists (ordered sequences of elements): lists as such, queues and stacks. Of course, there are many other equally useful structures – like dictionaries (maps), trees, graphs etc. — you will study them later, although we will say a few words about maps in the section on collections in Java.

9.1 Singly linked lists

A singly linked list represents a sequence of pieces of data of a certain type (in an extreme case, references to object of type **Object** which may represent anything). Each such piece of data is "wrapped" in an object of some type (conventionally called **Node**) as its field, the other field being the reference to the next node of the list. In this way, we can build a chain of nodes, where each node contains information about the next. We need some kind of a marker which will mark the last node as being the last: for example, we can adopt the convention that the last node's next field is **null**.

Thus, the situation looks like this:



Note that having the reference to the first node (conventionally called **head**), we can access all the nodes, because in each of them we will find the reference to the next one.

Let us consider an example. The class **Node** represents a node containing data – in this case just an **int** – and a reference to the next node:

Listing 46 BGU-SingList/Node.java

```java
   // could be private static inner class of MyList
public class Node {
    int data;
    Node next;
    Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }
    Node(int data) {
        this(data,null);
    }
}
```

The class representing the whole list will contain only one field: the head of the list, i.e., the reference to its first node. We will add methods which add new nodes at the beginning (**addFront**) and at the end (**addBack**) of the list. Notice that adding a node at the end requires traversing the whole list. Also, to count elements of the list (the **size** method), we have to traverse the whole list.

The **showListReversed** method prints the list in the reverse order by calling private, *recursive* function **showRev**. The main idea here is that the function *first* calls itself for the next node (if it exists), and only then, when the flow of control returns to it, prints information on *this* element.

Listing 47 BGU-SingList/MyList.java

```java
public class MyList {
    private Node head;

    public MyList() {
        head = null;
    }

    public void addFront(int data) {
        head = new Node(data,head);
    }

    public void addBack(int data) {
        if (head == null) {
            addFront(data);
            return;
        }
        Node tmp = head;
        while(tmp.next != null)
            tmp = tmp.next;
        tmp.next = new Node(data);
    }
    public void showList() {
        System.out.print("[ ");
        Node tmp = head;
        while(tmp != null) {
            System.out.print(tmp.data + " ");
            tmp = tmp.next;
        }
        System.out.println("]");
    }
    public void showListReversed() {
        System.out.print("[ ");
        showRev(head);
        System.out.print("]");
    }
    private void showRev(Node h) {
        if (h.next != null) showRev(h.next);
        System.out.print(h.data + " ");
    }
    public int size() {
          // inefficient!
        int count = 0;
        Node tmp = head;
        while(tmp != null) {
            ++count;
            tmp = tmp.next;
        }
        return count;
    }
    public boolean empty() {
        return head == null;
    }
}
```

In **Main**, we build a list and then print its contents

Listing 48 BGU-SingList/Main.java

```java
public class Main {
    public static void main(String[] args) {
        MyList list = new MyList();
        list.addBack(4);
        list.addBack(5);
        list.addFront(3);
        list.addFront(2);
        list.addFront(1);
        list.showList();
        list.showListReversed();
        System.out.println("\nsize = " + list.size());
    }
}
```

The program prints

```
[ 1 2 3 4 5 ]
[ 5 4 3 2 1 ]
size = 5
```

Of course, for our list to be useful, we would have to add more methods: for example to insert new node in the middle of the list, to remove its elements etc.

9.2 Stacks

A **stack** is a very simple but extremely useful data structure. It provides three operations

- **push** — adding new element;
- **pop** — removing one element (and getting its value);
- **empty** — checking if the stack is empty.

However, adding (pushing) and removing (popping) elements must be organized in such the way that what we pop is always what we pushed most recently. For example, if we push first A, then B and then C, three consecutive "pops" will give us first C, then B and lastly A. A structure organized in this way is called **LIFO** (*last in, first out*).

A stack may be easily implemented as a singly linked list: we always add (push) an element at the front (so it becomes the new head) and pop elements also from the front. Note that traversing the list will never be necessary!

Let us consider an example. Wrapping class **Node** looks the same as before

Listing 49 BGW-Stack/Node.java

```java
    // could be private static inner class of MyStack
public class Node {
    private int data;
    private Node next;
    Node(int data, Node next) {
        this.data = data;
        this.next = next;
    }
    Node(int data) {
        this(data,null);
    }
    int  getData() { return data; }
    Node getNext() { return next; }
}
```

and the implementation is very simple

Listing 50 BGW-Stack/MyStack.java

```java
public class MyStack {
    private Node head = null;
    public void push(int data) {
        head = new Node(data,head);
    }
    public int pop() {
        int d = head.getData();
        head = head.getNext();
        return d;
    }
    public boolean empty() {
        return head == null;
    }
}
```

In Main, we test our stack:

Listing 51 BGW-Stack/Main.java

```java
public class Main {
    public static void main(String[] args) {
        MyStack st = new MyStack();
        st.push(4);
        st.push(5);
        st.push(6);
        while (!st.empty())
            System.out.println("popping " + st.pop());
    }
}
```

The program prints

```
popping 6
popping 5
popping 4
```

The output shows that, indeed, what we have pushed last (value 6) has been popped first.

Another popular implementation of the stack is based on an array. This is illustrated in the program below. In the constructor, we allocate an array (with fixed size) and a field top which corresponds to the index of the first free 'slot' in the array (0 at the beginnng). When pushing a new value, we insert in at into the array at the position top and we increment top by 1. When popping, we first decrement top and we return the value from this position (which will then be treated as the first free 'slot'). The downside of this implementation is the fact that the capacity of the stack will be limited.

Listing 52 BGZ-ArrStack/ArrStack.java

```java
public class ArrStack {
    private int[] elems;
    private int top = 0;
    public ArrStack(int maxSize) {
        elems = new int[maxSize];
    }
    public void push(int e) {
            // throws if stack full
        elems[top++] = e;
    }
    public int pop() {
            // throws if stack empty
        return elems[--top];
    }
    public boolean empty() {
        return top == 0;
    }
}

```

InMain, we create the stack and test its behavior:

Listing 53 BGZ-ArrStack/Main.java

```java
public class Main {
    public static void main (String[] args) {
        ArrStack stack = new ArrStack(5);
        for (int i = 1; i <= 5; ++i)
            stack.push(i*i);
        while (!stack.empty())
            System.out.print(stack.pop() + " ");
        System.out.println();
    }
}
```

and we get

```
25 16 9 4 1
```

Note that we do *not* handle the situations when an attempt is made to pop from an empty stack or to push onto a full stack: an exception will be thrown anyway, as an out-of-bounds index will then be used.

9.3 Queues

A **queue** is similar to a stack, but with the reversed order of "pops": now, what we pop is not the most recent element pushed, but the "oldest" — this is called **FIFO** (*first in, first out*). For example, if we enqueue first A, then B and then C, three consecutive "dequeues" will give us first A, then B and finally C.

A very simple implementation is again based on a singly linked list: we always add elements at the end and remove them from the beginning. This will not require traversing the list, if we always keep the reference to the last element (called tail in the example below).

Let us consider an example. The **Node** class is now a nested class: it is declared as static and *private* class *inside* the **MyQueue** class, so it will be invisible for the user but can be used inside the enclosing class **MyQueue** (we could have done it in previous examples as well):

Listing 54 BGY-SimpQueue/MyQueue.java

```java
public class MyQueue {
      // nested class
    private static class Node {
        String data;
        Node next = null;
        Node(String d) { data = d; }
    }

    private Node head, tail;

    public MyQueue() {
        head = tail = null;
    }
    public void enqueue(String s) {
        if (head == null)
            head = tail = new Node(s);
        else
            tail = tail.next = new Node(s);
    }
    public String dequeue() {
        String s = head.data;
        if ((head = head.next) == null) tail = null;
        return s;
    }
    public boolean empty() {
        return head == null;
    }
}
```

In **Main**, we test our queue:

Listing 55 BGY-SimpQueue/Main.java

```java
public class Main {
    public static void main (String[] args) {
        MyQueue qS = new MyQueue();
        for (double d = 0.5; d < 5; d += 1)
            qS.enqueue("" + d);
        while (!qS.empty())
            System.out.print(qS.dequeue() + " ");
        System.out.println();
    }
}
```

the program prints

```
0.5 1.5 2.5 3.5 4.5
```

so the elements are printed in the same order in which they have been enqueued.

Section 10

Strings and *StringBuilders*

10.1 Class String

The type String is an object type (*not* a primitive type). This means that there is no way go give a string a name — we always operate on strings using references to strings (variables holding, as their values, addresses of objects allocated on the heap).

There are several ways of creating objects of type String (and references to them). As String is so often used, there exist some syntactic simplifications when dealing with objects of this type. Normally, objects are created using new operator, but for Strings we can use:

```
String s = "Pernambuco";
```

or, if we have an int named a,

```
String s = "a = " + a;
```

Of course, we can also use 'normal' form, for example

```
String s = new String("Pernambuco");
```

A string can also be created from an array of characters:

```
char[] arr = {'A', 'B', 'C'};
String s = new String(arr);
```

and in a few other ways (see documentation).

Objects of type String are *objects* (as opposed to variables of primitive types), so we can invoke *methods* on them. They will never modify an existing string (in particular the one a method is invoked on), but many such methods return the reference to a newly created objects, created on the basis of the original one. Examples — out of many, see documentation — include (we assume that s is the reference to a String object):

- `s.length()` — returns the size (number of characters) of s (`"ABC".length()` is 3) — note that this is a *method*, not a field as for arrays;
- `s.equals(anotherstring)` — returns true if and only if the string s is equal (contains the same characters) as anotherstring; this is the correct way to compare strings for equality — note that `s == anotherstring` or `s != anotherstring` compares *addresses* of objects, not their contents;
- `s.equalsIgnoreCase(anotherstring)` — as equals, but the case of characters is ignored ('A' and 'a' are considered equivalent);
- `s.trim()` — returns the reference to a newly created string which is as s but with all leading and trailing white spaces removed (`"   ABC   "` → `"ABC"`);
- `s.toLowerCase()`, `s.toUpperCase()` — return the reference to a newly created string which is as s but with all upper- (lower-) case letters replaced by lower- (upper-) case ones — non-letters are not modified (`"aBC".toLowerCase()` → "abc");
- `s.substring(from, to)` — returns the reference to a newly created string which is a substring of s from character with index from (inclusive) to character with index to *exclusive* (indexing starts with 0), for example `"abcd".substring(1,3)` → "bc";

- `s.substring(from)` — equivalent to `s.substring(from, s.length());`
- `s.charAt(ind)` — returns (as value of type **char**) the character with index ind (indexing starts with 0), for example `"abcd".charAt(2)` $\to$ 'c';
- `s.contains(anotherstring)` — returns **true** if s contains as its substring the string anotherstring, and **false** otherwise;
- `s.startsWith(anotherstring)`, `s.endsWith(anotherstring)` — returns **true** if s starts (ends) with the substring anotherstring, and **false** otherwise;
- `s.indexOf(anotherstring)`, `s.indexOf(char)` — returns the first index where the substring anotherstring (character char) occurs in s, or -1 when such substring (character) doesn't occur in s (`"abcdcd".indexOf("cd")` $\to$ 2);
- `s.lastIndexOf(anotherstring)`, `s.lastIndexOf(char)` — returns the last index where the substring anotherstring (character char) occurs in s, or -1 when such substring (character) doesn't occur in s (`"abcdcd".lastIndexOf("cd")` $\to$ 4);
- `s.indexOf(anotherstring, ind)`, `s.indexOf(char, ind)` — as `s.indexOf(anotherstring)` and `s.indexOf(char)`, but searching starts at index ind;
- `s.lastIndexOf(anotherstring, ind)`, `s.lastIndexOf(char, ind)` — as `s.lastIndexOf(anotherstring)` and `s.lastIndexOf(char)` – searching starts at index ind;
- `s.toCharArray()` — returns the reference to an array of characters (**char[]**) of length s.length() and containing individual characters of the string s;
- `s.split(regex)` — returns the reference to an array of strings (**String[]**) containing substrings of s where regex is treated as a separator between elements. This is a regex (regular expression) which we don't know about yet: very often you can use a string containing a single character, or `"\\s+"` which means *any nonempty sequence of white characters*. For example, if s is the reference to string `"aa:bb:cc"`, then `s.split(":")` will give us an array of **String**s containing three elements: `"aa"`, `"bb"` and `"cc"`.

Some of these methods are used in the program below

Listing 56 BHH-Strings/Strings.java

```java
public class Strings {
    private static void pr(String m, String a, String b) {
        System.out.println(m + ": \"" + a +
                     "\" -> \"" + b + "\"");
    }
    public static void main (String[] args) {
          // length()
        String a = " Shakespeare   ";
        String b = a.trim();
        System.out.println("a.length() -> " + a.length());
        System.out.println("b.length() -> " + b.length());
        pr("        trim()", a, b);

          // substring
        a = "abcdefgh";
        b = a.substring(3,6);
        pr("substring(3,6)", a, b);
        b = a.substring(3);
        pr("  substring(3)", a, b);

          // toUpperCase, toLowerCase
        b = a.toUpperCase();
        pr(" toUpperCase()", a, b);

          // split
        String[] arr = "ONE:TWO:THREE".split(":");
        for (String d : arr)
            System.out.print(d.toLowerCase() + " ");
        System.out.println();

        arr = "one   two       three".split("\\s+");
        for (String d : arr)
            System.out.print(d.toUpperCase() + " ");
        System.out.println();

          // charAt
        String ny = "New York";
        for (int i = 0; i < ny.length(); ++i)
            System.out.print(ny.charAt(i) + " ");
        System.out.println();

          // toCharArray
        char[] ca = ny.toCharArray();
        for (int i = ca.length-1; i >= 0; --i)
            System.out.print(ca[i]);
        System.out.println();

    }
}
```

which prints

```
a.length() -> 15
b.length() -> 11
        trim(): " Shakespeare   " -> "Shakespeare"
substring(3,6): "abcdefgh" -> "def"
  substring(3): "abcdefgh" -> "defgh"
 toUpperCase(): "abcdefgh" -> "ABCDEFGH"
one two three
ONE TWO THREE
N e w   Y o r k
kroY weN
```

10.2 Class StringBuilder

The fact that Strings are immutable has many advantages but the downside is that manipulating strings always involves creation of new objects of type String. For example, when we want to append something to a string using `s=s+"something"`, we actually do *not* append anything to an existing string; we just create a completely new object and the reference to this new object assign back to `s` erasing its previous contents (but the string that was referenced to by `s` before still exists on the heap in memory).

Problems related to string being immutable are addressed by the class StringBilder. Object of this type are similar to String objects but *are* modifiable. Internally, they allocate an array of characters and operate on it; when it becomes too small, another, twice as big array, is allocated and all existing elements are copied to it. As at each reallocation much bigger array is allocated, its size grows rapidly and hence not many such reallocations will ever be required. Probably the most useful methods of String-Bilder are **append**, which appends to a string another string, **insert**, which inserts a string on an arbitrary position into an existing string; as the mater of fact almost anything can be inserted, in particular numbers and also objects (in that case, the result of invocation of **toString** will be inserted). Also, the method **delete** is often useful: it can remove a fragment of the string. Invoking **toString** on a StringBuilder object gives us the final, immutable String object. Many of the methods mentioned above return this object, so their invocations may be chained, as in the example below:

Listing 57 BHI-StrBuilder/StrBuilder.java

```java
public class StrBuilder {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("three");
        sb.append(",four")
          .insert(0,"twoxx,")
          .insert(0,"one,")
          .delete(sb.indexOf("x"), sb.indexOf("x")+2);
        System.out.println("sb = " + sb.toString());

        final int NUMB = 50_000;
        long startTime;

        startTime = System.nanoTime();
        String s = "0";
        for (int i = 1; i <= NUMB; ++i)
            s = s + i;
        long elapsedS = System.nanoTime() - startTime;
        System.out.printf(" String: %10d ns = %.3f sec%n",
                                  elapsedS, elapsedS*1e-9);

        startTime = System.nanoTime();
        StringBuilder builder = new StringBuilder("0");
        for (int i = 0; i <= NUMB; ++i)
            builder.append(i);
        long elapsedB = System.nanoTime() - startTime;
        System.out.printf("Builder: %10d ns = %.3f sec%n",
                                  elapsedB, elapsedB*1e-9);

        System.out.printf("elapsedS/elapsedB = %.2f%n",
                             (double)elapsedS/elapsedB);
    }
}
```

The program prints

```
sb = one,two,three,four
 String:  735807986 ns = 0,736 sec
Builder:    2263932 ns = 0,002 sec
elapsedS/elapsedB = 325,01
```

which shows that operations on **StringBilder** objects can be orders of magnitude faster than analogous operations on **String**s.

Section 11

Introduction to inheritance

Inheritance applies to situations when one type describes objects which behave as objects of another ('base') type but in a different way or they possess some additional properties. For example dog is an animal, so a class describing dogs may inherit from (extend) class describing animals. This is called 'is-a' relation: dog *is-a* animal (but not necessarily the other way around — not all animals are dogs). By using inheritance, we can reuse the code written in the base class (Animal) and not repeat it in class Dog. What is more important, however, everywhere where an animal is expected (any animal: maybe dog, maybe cat), we can use a dog, because dog *is-a* animal.

In particular, you can create an object of the derived class (Dog) and assign the address (reference) obtained from new to a reference variable declared as having the type of the base class (Animal). In such a situation, we say that the *static* (declared) type of the refernce is Animal, but the dynamic ('real') type is Dog. Through this reference, the compiler will allow you to use all methods from Animal, but not those which exist in Dog but are not inherited from Animal. However, if, in class Dog, you have **overridden** (redefined) a method inherited from Animal, then at runtime this redefined (coming from Dog) method will be used even though you refer to the object by a reference whose static type is Animal — this is called "late binding" and is the essence of **polymorphism**. Methods with such behavior are called **virtual** — in Java, therefore, all methods *are* virtual. The only exception is a method declared as final: any attempt to override such final method will fail.

Any class can directly extend (inherit from) *only one* base class (also called its **superclass**). If, defining a class, we do not specify its superclass, it is assumed that it inherits from a special class Object; it follows, that *every* class inherits directly or indirectly, from Object. Class Object defines a few methods which are therefore inherited (and, consequently, *exist*) in any class. One important example of such a method is toString (shown in the example below). Its purpose is to return a String describing somehow the object it was invoked on. Usually, we redefine toString in our classes: otherwise the inherited version will be used, what will always succeed but may return a rather useless result. When we redefine (override) an inherited method, it is recommended (although not required) to annotate this redefinition with Override, as in the example below; this makes our intention explicit to the compiler which can therefore verify if indeed the declaration of the method is correct. Also, note the use of instanceof operator:

```
a instanceof AClass
```

answers (at runtime) the question *is the object referred to by* a *of type* AClass *or any of its subclasses*. We can also find or compare types of objects refereed to by a reference using getClass method (which is defined in Object, so all classes inherit it). It returns an object of class Class which represents the real type of the object it was invoked on: such a *single* object is created for each class loaded by the JVM. As there is only one such object for any class, one *can* use == operator to compare types of any two objects by using the objects of class Class representing their types.

Let us consider an example: we define a simple class Point

Listing 58 DJV-Inherit/Point.java

```java
public class Point {

    protected int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Point(int x) {
        this(x,0);
    }

    public Point() {
        this(0,0);
    }

    public Point translate(int dx,int dy) {
        x += dx;
        y += dy;
        return this;
    }

    public Point scale(int sx,int sy) {
        x *= sx;
        y *= sy;
        return this;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    @Override
    public String toString() {
        return "[" + x + "," + y + "]";
    }
}
```

and then class **Pixel** which extends **Point** (all pixels *are* points, but they have also a color):

Listing 59 DJV-Inherit/Pixel.java

```java
import java.awt.Color;

public class Pixel extends Point {

    private Color color;

    public Pixel(int x, int y, Color color) {
        super(x,y);
        this.color = color;
    }

    public Pixel(int x, int y) {
        this(x,y,Color.BLACK);
    }

    public Pixel(Color color) {
        this(0,0,color);
    }

    public Pixel() {
        this(0,0,Color.BLACK);
    }

      // n o t  inherited
    public Color getColor() {
        return color;
    }

    @Override
    public String toString() {
        return super.toString() + "(r=" + color.getRed() +
               ",g=" + color.getGreen() + ",b=" +
               color.getBlue() + ")";
    }
}
```

We then check static and dynamic types of various references

Listing 60 DJV-Inherit/Main.java

```java
import java.awt.Color;

public class Main {
    public static void main(String[] args) {
        Point pp = new Point(2,1);
        Point pt = new Pixel(1,2);
        Pixel px = new Pixel(new Color(255,51,102));

        System.out.println("is 'pp' a Point? " +
                (pp instanceof Point));
        System.out.println("is 'pp' a Pixel? " +
                (pp instanceof Pixel));
        System.out.println("is 'pt' a Point? " +
                (pt instanceof Point));
        System.out.println("is 'pt' a Pixel? " +
                (pt instanceof Pixel));
        System.out.println("is 'px' a Point? " +
                (px instanceof Point));
        System.out.println("is 'px' a Pixel? " +
                (px instanceof Pixel));
        System.out.println("class of pp: " +
                pp.getClass().getName());
        System.out.println("class of pt: " +
                pt.getClass().getName());
        System.out.println("class of px: " +
                px.getClass().getName());
          // Pixel is a Point; we can translate or scale it
        px.translate(5,4).scale(2,3).translate(-1,-3);
        System.out.println("pp: " + pp);
        System.out.println("pt: " + pt);
        System.out.println("px: " + px);
        System.out.println("Color px : " + px.getColor());
          // casting required!
        System.out.println("Color pt : " +
                           ((Pixel)pt).getColor());
    }
}
```

The program prints

```
is 'pp' a Point? true
is 'pp' a Pixel? false
is 'pt' a Point? true
is 'pt' a Pixel? true
is 'px' a Point? true
is 'px' a Pixel? true
class of pp: Point
class of pt: Pixel
class of px: Pixel
pp: [2,1]
pt: [1,2](r=0,g=0,b=0)
px: [9,9](r=255,g=51,b=102)
Color px : java.awt.Color[r=255,g=51,b=102]
Color pt : java.awt.Color[r=0,g=0,b=0]
```

Note that we got that pt *is* an instance of **Pixel** although its static type is **Point**. This is because the **instanceof** operator is dynamic (polymorphic): it detects, at runtime, the 'real' type of the object pointed to by a reference. However, note that we *cannot* call **getColor** on pt: it refers to a **Pixel**, but its static type is **Point** and only the static type is taken into consideration at compile time — as there is no **getColor** method in **Point**, the compilation would fail. Therefore, we have to *cast* the `pt` reference (see the last line of the program). By writing `((Pixel)pt)`, we are saying to the compiler 'this is just a **Point** for you, but believe me, I know and I promise you that its real type is **Pixel**'. The compiler cannot check it, but at runtime the program will crash if it turns out that the object pointed to by `pt` is in fact *not* a **Pixel**.

Let us consider another example: we define an **Animal** class

Listing 61 DJW-InherAnimal/Animal.java

```java
public class Animal {
    protected String name;
    protected double weight;
      // no default constructor!
    public Animal(String n, double w) {
        name = n;
        weight = w;
    }
    public String getVoice() {
        return "?";
    }
    public static void voices(Animal[] animals) {
        for (Animal a : animals)
            System.out.println(a + " " + a.getVoice());
    }
    @Override
    public String toString() {
        return name + "(" + weight + ")";
    }
}
```

which defines the **getVoice** method. It also defines a static method **voices** which traverses an array of (references to) **Animal**s (not knowing what the real type of objects pointed to by the elements of the array are). However, when calling **toString** or **getVoice**, the methods from the real type will be selected (therefore, it is a *polymorphic* invocation).
We then create two classes which extend **Animal**: class **Dog**

Listing 62 DJW-InherAnimal/Dog.java

```java
public class Dog extends Animal {
    public Dog(String name, double weight) {
        super(name, weight);
    }
    @Override
    public String getVoice() {
        return "Bow-Wow";
    }
    @Override
    public String toString() {
        return "Dog " + super.toString();
    }
}
```

and class **Cat**

Listing 63 DJW-InherAnimal/Cat.java

```java
public class Cat extends Animal {
    public Cat(String name, double weight) {
        super(name, weight);
    }
    @Override
    public String getVoice() {
        return "Miaou-Miaou";
    }
    @Override
    public String toString() {
        return "Cat " + super.toString();
    }
}
```

In both classes, we *override* (redefine) the **getVoice** method. Note that there is no default constructor in the base class **Animal** – therefore, using **super** in constructors of the derived classes is obligatory.
Now in another class

Listing 64 DJW-InherAnimal/Main.java

```java
public class Main {
    public static void main (String[] args) {
        Animal[] animals = {
                new Dog("Max", 15), new Cat("Batty", 3.5),
                new Dog("Ajax", 5), new Cat("Minnie", 4)
        };
        Animal.voices(animals);
    }
}
```

we create an array of various animals — some are dogs, some are cats — and pass it to the **voices** method from **Animal**. As we can see from the output

```
Dog Max(15.0) Bow-Wow
Cat Batty(3.5) Miaou-Miaou
Dog Ajax(5.0) Bow-Wow
Cat Minnie(4.0) Miaou-Miaou
```

invocations in `System.out.println(a + " " + a.getVoice())` inside the **voices** method are polymorhic: both **toString** and **getVoice** have been taken from the real type of objects pointed to by elements of the array animals, although the declared type of these elements was **Animal** and there was no way for the compiler to know their real type.

11.1 Importance of equal and hashCode methods

Class **Object** defines — among others (in particular

```
public String toString()
```

that we already know) two other important methods:

- `public boolean equals(Object ob)` — which is used to check if two objects are, according to some criterion, equal: for two references `ob1` and `ob2`, the expression `ob1.equals(ob2)` compares the objects and yields **true** or **false**. But according to what criterion? In class **Object** there is no data to be compared, so the default implementation just compares addresses of the two objects. Very often, this is *not* what we want. When we have two objects of class **Person** and these persons have identical names, dates of birth, passport numbers etc., we rather want to consider the two objects as representing exactly the same person, in other words we want to consider these two objects equal. To get this behavior, we thus have to redefine (override) **equals** in our class.
  The **equals** method should implement an equivalence relation on non-null object references. This means that for any non-null references `a`, `b` and `c` `a.equals(a)` should always be **true** (reflexivity), `a.equals(b)` should have the same value (**true** or **false**) as `b.equals(a)` (symmetry), and if `a.equals(b)` and `b.equals(c)` are **true** then `a.equals(c)` must also be **true** (transitivity).
  Reflexivity and symmetry are usually obvious, but transitivity — not always. Suppose, we consider two two-element sets equal if they have at least one common element. Then $A = \{a, b\}$ is equal to $B = \{b, c\}$ and B is equal to $C = \{c, d\}$, but A and C have no common element and are not equal.
- `public int hashCode()` — which is used to calculate the so called **hash code** of an object. This is necessary if we want to put objects of our class in collections implemented as hash tables (e.g., **HashSet** or keys in a **HashMap**). The implementation of **hashCode** method from the **Object** class uses just the address of the object to calculate its hash code. However, very often this is not desirable as it would lead to situations when two objects that are considered equal (according to **equals**) have different hash codes: as a consequence the collections of such objects would be invalidated. Therefore, we have to override also this method remembering to do it consistently: whenever two objects are equal according to **equal**, their hash codes should be exactly the same.

The following example illustrates **toString** and **equals** methods:

Listing 65 AAR-EquToString/EquToString.java

```java
public class EquToString {
    public static void main(String[] args) {
        Person johny = new Person("John",1980);
        Person john  = new Person("John",1980);
        Osoba  jasio = new Osoba("Jan",1980);
        Osoba  jan   = new Osoba("Jan",1980);

        if (johny.equals(john))
```

```java
            System.out.println("johny==john");
        else
            System.out.println("johny!=john");

        if (jasio.equals(jan))
            System.out.println("jasio==jan");
        else
            System.out.println("jasio!=jan");

        System.out.println("johny: " + johny);
        System.out.println("jasio: " + jasio);
    }
}
```

The program prints

```
johny == john
billy != bill
johny: John(1980)
billy: PersonBad@659e0bfd
```

As we can see, with **toString** and **equals** methods *not* overriden (as in class **PersonBad**), the program works but uses the versions of these methods from class **Object**. To get the expected results, we have to override these methods, as we did in class **PersonGood**.

The next example illustrates the effect of overriding the **hashCode** method:

Listing 66 HUM-HashEquals/Person.java

```java
public class Person {

    private String name;
    private String idNumber;

    public Person(String name, String idNumber) {
        this.name     = name;
        this.idNumber = idNumber;
    }

    /*
    @Override
    public boolean equals(Object other) {
        if (other == null ||
            getClass() != other.getClass()) return false;
        Person p = (Person)other;
        return idNumber.equals(p.idNumber) &&
                   name.equals(p.name);
    }
    /**/

    /*
    @Override
    public int hashCode() {
        return 17*name.hashCode() + idNumber.hashCode();
    }
    /**/

    @Override
    public String toString() {
        return name + "(" + idNumber + ")";
    }
}
```

with **main** as below

Listing 67 HUM-HashEquals/AHash.java

```
import java.util.HashMap;
import java.util.Map;

public class AHash {
    public static void main(String[] args) {
        Map<Person,String> map = new HashMap<>();

        map.put(new Person("Sue","123456"),"Sue");

          // new object, but should be equivalent to
          // the one which has been put into the map
        Person sue = new Person("Sue","123456");

        if (map.containsKey(sue))
            System.out.println(sue + " has been found");
        else
            System.out.println(sue + " has NOT been found");
    }
}
```

As can be easily checked, the program will print

```
Sue(123456) has been found
```

only when both **hashCode** and **equals** are consistently overridden in the class of keys of the map (i.e., **Person**).

Section 12

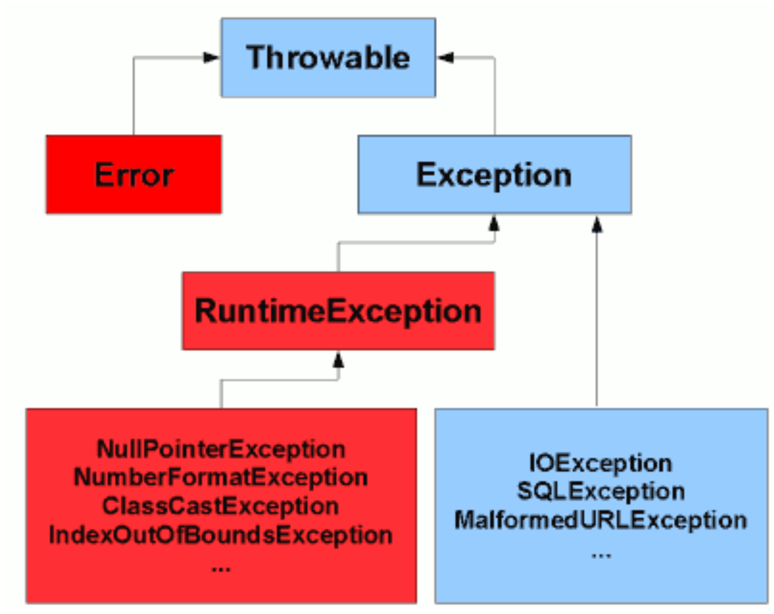
Exceptions

Exceptions are situations when program does not know what to do next. For example, we say *read data from this file*, but there is no file to read from. Or, we try to print an element of an array with a given index, but this index is negative or bigger then the size of our array, so the required element doesn't exist. In all such situations, normal flow of control is disrupted and an exception is **thrown**: the JVM creates an object representing the error and checks, if we have supplied a code to somehow handle this kind of error (we will see in a moment, how to supply such code).

All possible exceptions fall into one of two categories:

- Checked exceptions: these are exceptions that *must* be somehow dealt with by our program, otherwise the program will not even compile. We can deal with checked exceptions
  1. by using `try-catch` blocks;
  2. by declaring the function in which a given exception may be thrown as throwing it — then the caller of the function will have to handle it.
- Unchecked exceptions: these are exceptions that we can, but not have to, handle. If we don't handle it and this kind of exception occurs, the program terminates (after printing some information about the exception that has been encountered).

Exceptions are represented by objects of types extending `Throwable` (the top of the hierarchy tree). It has two 'children', one of them, `Exception`, represents *checked* exceptions, except the subtree rooted at `RuntimeException`, which are unchecked. The second child subtree, rooted at `Error`, represents unchecked exceptions only used internally by JVM; normally, they shouldn't (although they can, if we insist) be handled at all, as they indicate serious problems beyond our control. The general structure of the `Throwable` class hierarchy looks like this (image taken from javamex page[3]), where red color denotes subtrees of *unchecked* exceptions:



[3]http://www.javamex.com

12.1 try-catch blocks

How do we handle exceptions? We enclose a fragment of our code where an exception may, at least potentially, occur in a **try** block of the **try-catch** construct:

```
try {
    // ...
} catch(ExcType1 ex) {
    // handling the exception of type ExcType1
} catch(ExcType2 ex) {
    // handling the exception of type ExcType2
}
// ... other catch blocks
```

The code in the **try** block is executed. If everything goes smoothly and there is no exception, the **catch** blocks are ignored. However, suppose that at some point when executing the code in the **try** block, an exception occurs. Then

- execution of the code in the **try** block is disrupted;
- object of a class corresponding to the type of the exception is created;
- the type of this object is compared with **ExcType1**; if this type is **ExcType1** or its subclass (i.e., a class extending it directly or indirectly), then the flow of control enters the corresponding **catch** block, the exception is deemed to be already handled and the code in the **catch** block is executed (inside the block, `ex` will be the reference to the object representing the exception; such objects carry a lot of information and we can invoke useful methods on them). After that, no other **catch** block will be considered;
- if the type of the exception is not **ExcType1** or its subclass, then, in the same way, the next **catch** block is tried;
- ...and so on, until a **catch** block with an appropriate type is found.

Note that it doesn't make sense to place a **catch** block corresponding to exceptions of type **Type2** *after* the **catch** block corresponding to **Type1**, if type **Type2** extends **Type1**. This is because exceptions of type **Type2** would be 'caught' by the first **catch**, making the second unreachable. For example, there is a class **FileNotFoundException** which is derived (inherits, extends) from **IOException**. Something like

```
try {
    // ...
} catch(FileNotFoundException e) {
    // ...
} catch(IOException e) {
    // ...
}
```

does make sense: if **FileNotFoundException** occurred, then the first **catch** will be executed, if it was any other exception derived from **IOException** — the second one. However

```
try {
    // ...
} catch(IOException e) {
    // ...
} catch(FileNotFoundException e) { // NO!!!
    // ...
}
```

is wrong, because FileNotFoundException and all other exceptions derived form IOException will be caught by the first catch; the second is unreachable and hence useless.

It is possible for one catch block to handle two (or more) types of unrelated exceptions (they are unrelated if none of them is a subtype of the other). We just specify these types separating their names with a vertical bar (|); for example in

```
    // ...
    catch (IOException | SQLException e) {
        // ...
        throw e;  // rethrowing exception
    }
}
```

the catch will catch exceptions of type IOException and of type SQLException as well (and their subclasses). This example also shows that handling an exception, we can, after doing something, **rethrow** the same exception (or, as a matter of fact, another one) to be handled elsewhere.

12.2 finally block

After all catch blocks, we can (but we don't have to) use a **finally block**

```
    try {
        // ...
    } catch(ExcType1 e) {
        // ...
    } catch(ExcType2 e) {
        // ...
    } finally {
        // ...
    }
```

The code in finally block will always be executed. If there is no error in the try block, then all catch blocks will be ignored, but finally block will be executed; if there is an error in the try block, then the corresponding catch block (if any) will be executed and, afterwards, the finally block (even if there is a return statement in try and/or catch blocks!) Finally blocks are very useful when we want to ensure that some resources (open files, open internet or data-base connections, locks etc.) are released always — whether an exception occurred or not. It is even possible to use a finally block when no exception is anticipated. For example, the code below can be dangerous

```
    // get resources
    // ...
    if ( condition1 ) return;
    // ...
    if ( condition2 ) return;
    // ...
    // release resources
```

because if any of conditions holds, the resources acquired at the beginning will not be released. However, using finally

```
    try {
        // get resources
        // ...
        if ( condition1 ) return;
        // ...
        if ( condition2 ) return;
        // ...
    } finally {
        // release resources
    }
```

we can ensure that the resources will be released no matter what.

12.3 Propagating exceptions

Sometimes we know that an exception (in particular, a checked one) can occur but we don't know how to handle it. Then we can declare the whole function as throwing this kind of exception (or even, comma separated, exceptions of two or more types):

```
    void fun( /* parameters */ ) throws ExcType1, ExcType2 {
        // here we do  n o t  handle exceptions
        // of types ExcType1, ExcType2
    }
```

In this way we propagate the exception 'up the stack' to the caller (the function which calls our fun), so there the invocation of fun will have to be handled

```
    void caller( /* parameters */ ) {
        // ...
        try {
            fun( /* args */ );
        } catch(ExcType1 e) {
            // ...
        } catch(ExcType2 e) {
            // ...
        }
        // ...
    }
```

Of course, the caller function can itself be declared as throwing and then exceptions will be propagated to the caller of caller, and so on. In this way, we can even propagate exceptions up to the main function, which, although it is not recommended, itself may be declared as throwing (propagating exceptions to the caller of main, i.e., the JVM itself).

12.4 Throwing exceptions

In some situations, we can throw exceptions by ourselves. Suppose we implement, in class Person, a method setting the person's age

```
    void setAge(int age) {
        this.age = age;
    }
```

We may decide that trying to set a negative age is a user's error. We can signal such invalid attempt by throwing an exception of some type. Java defines many types of exceptions in its standard library — in this particular case IllegalArgumentException would be probably most appropriate. We thus rewrite out method like this

```
    void setAge(int age) {
        if (age < 0)
            throw new IllegalArgumentException("age<0");
        this.age = age;
    }
```

This exception is unchecked, so we don't have to handle it or to declare the method as throwing it. Nevertheless, it is a good practice to document the fact that such an exception might be thrown by adding it to the declaration of the method

```
    void setAge(int age) throws IllegalArgumentException {
        if (age < 0)
            throw new IllegalArgumentException("age<0");
        this.age = age;
    }
```

As we said, many classes representing exceptions have already be written and are part of the standard library. However, we can also create such classes ourselves. Normally, it is very simple; all we have to do is to define a class extending a library class: a subclass of RuntimeException if we want an unchecked exception or a subclass of Exception if we want our class to represent a checked exception. All library exception classes have two constructors: the default one and one taking a string representing a message that can be queried on the object. So we can write

```
    public class MyException extends Exception {
        public MyException() { }
        public MyException(String message) {
            super(message);
        }
    }
```

and that's basically enough, although, of course, we can add more constructors, fields and methods if we find it appropriate for our purposes. This is seldom necessary, because the main message passed by an exception is just its type.

12.5 Examples

In the example below, we handle possible exceptions related to input/output operations. In particular, opening a file and reading from it using a Scanner can throw a *checked* exception (a subtype of IOException) that we have to handle. [Note that reading from System.in does *not* throw checked exceptions.]

Listing 68 AXW-ScanExcept/ScanExcept.java

```java
import java.io.IOException;
import java.nio.file.Paths;
import java.util.Scanner;

public class ScanExcept {
    public static void main (String[] args) {
          // no checked exception here
        Scanner console = new Scanner(System.in);

        Scanner  scfile = null;
        try {
            scfile = new Scanner(Paths.get("pangram.txt"),
                                "UTF-8");
            int count = 0;
            while (scfile.hasNextLine())
                System.out.printf("%2d: %s%n", count,
                                  scfile.nextLine());
        } catch(IOException e) {
            System.out.println("Message: " + e +
                               "\n**** Now the stack:");
            e.printStackTrace();
            System.out.println("**** CONTINUING...");
        }

        System.out.println("Enter anything...");
        String s = console.next();
        System.out.println("After try/catch: " +
                      "read from console: " + s);

          // active only when run with '-ea' option!
          // should be used for debugging only.
          // No side effects!
        assert scfile != null : "apparently no file";

        console.close();
        scfile.close();
    }
}
```

When the input file ***file.txt*** exists, reading it succeeds, the **catch** clause is not entered, but the **finally** clause is executed:

```
 1: Line 1
 2: Line 2
 3: Line 3
FINALLY executed
Enter anything...
something
Read from console: something
```

However, when the input file (*file.txt*) is missing, the **FileNotFoundException** is thrown, variable scfile remains **null** and the **catch** clause is entered. The exception is now considered handled, so the program continues (and the **finally** clause is executed anyway):

```
Exception: java.nio.file.NoSuchFileException: file.txt
**** Now the stack:
java.nio.file.NoSuchFileException: file.txt
        at java.base/sun.nio.fs.UnixException
            .translateToIOException(UnixException.java:92)
        at java.base/sun.nio.fs.UnixException
            .rethrowAsIOException(UnixException.java:106)
        at java.base/sun.nio.fs.UnixException
            .rethrowAsIOException(UnixException.java:111)
        at java.base/sun.nio.fs.UnixFileSystemProvider
            .newByteChannel(UnixFileSystemProvider.java:218)
        at java.base/java.nio.file.Files
            .newByteChannel(Files.java:380)
        at java.base/java.nio.file.Files
            .newByteChannel(Files.java:432)
        at java.base/java.nio.file.spi.FileSystemProvider
            .newInputStream(FileSystemProvider.java:422)
        at java.base/java.nio.file.Files
            .newInputStream(Files.java:160)
        at java.base/java.util.Scanner
            .makeReadable(Scanner.java:622)
        at java.base/java.util.Scanner.<init>(Scanner.java:759)
        at java.base/java.util.Scanner.<init>(Scanner.java:741)
        at ScanExcept.main(ScanExcept.java:13)
**** CONTINUING...
FINALLY executed
Enter anything...
something
Read from console: something
Exception in thread "main"
        java.lang.AssertionError: apparently no file
        at ScanExcept.main(ScanExcept.java:34)
```

At the end of the program, we used an **assertion statement**; we will say a few words about this topic in a moment.

In the example below, we read data from the user and use (unchecked) exception of type **NumberFormatException** to ensure that the data is valid:

Listing 69 AXX-TryCatch/TryCatch.java

```java
import javax.swing.JOptionPane;

public class TryCatch {
    public static void main(String[] args) {
        boolean noGood = true;
        int     number = 0;

        while (noGood) {
            String s = JOptionPane.showInputDialog(
                null,"Enter an integer","Entering data",
                JOptionPane.QUESTION_MESSAGE);

            if (s == null)       break;
            if (s.length() == 0) continue;
            try {
                number = Integer.parseInt(s);
                noGood = false;
            } catch(NumberFormatException e) {
                JOptionPane.showMessageDialog(
                    null,"<html>This is not an integer!" +
                         "<br />Try again!!!",
                    "ERROR",JOptionPane.ERROR_MESSAGE);
            }
        }

        System.out.println(
            noGood ?
                "Program cancelled" : "Entered: " + number);
    }
}
```

The following example demonstrates the use of two different unrelated exceptions:

Listing 70 AYC-Excpts/Excpts.java

```java
import javax.swing.JOptionPane;

public class Excpts {
    public static void main(String[] args) {
        boolean noGood = true;
        int     number = 0;

        while (noGood) {
            String s = JOptionPane.showInputDialog(
                null,"Enter two integers","Entering data",
                JOptionPane.QUESTION_MESSAGE);

            if (s == null) break;
            s = s.trim();
            if (s.length() == 0) continue;

            int spac = s.indexOf(' ');
            if (spac < 0) {
                JOptionPane.showMessageDialog(
                    null,"<html>Wrong data!" +
                        "<br />Try again!!!",
                    "ERROR",JOptionPane.ERROR_MESSAGE);
                continue;
            }

            try {
                int n1 = Integer.parseInt(
                                 s.substring(0,spac));
                int n2 = Integer.parseInt(
                          s.substring(spac+1).trim());
                number = n1/n2;
                noGood = false;

            } catch(NumberFormatException e) {
                JOptionPane.showMessageDialog(
                    null,"<html>This were not integers!" +
                         "<br />Try again!!!",
                    "ERROR",JOptionPane.ERROR_MESSAGE);
                System.err.println("EXCEPTION " +
                    e.getMessage() + '\n' + "STACK TRACE:");
                e.printStackTrace();

            } catch(ArithmeticException e) {
                JOptionPane.showMessageDialog(
                    null,"<html>Division by zero!" +
                         "<br />Try again!!!",
                    "ERROR",JOptionPane.ERROR_MESSAGE);
                System.err.println("EXCEPTION " +
                    e.getMessage() + '\n' + "STACK TRACE:");
                e.printStackTrace();
            }
        }

        System.out.println(
            noGood ?
                "Program cancelled"
              : "Result of division: " + number);
    }
}
```

The last example illustrates custom (user defined) exceptions: we define our own type of exception:

Listing 71 AYN-CheckedExc/MyUncheckedException.java

```
public class MyUncheckedException
                    extends IllegalArgumentException {
    MyUncheckedException() {
        super();
```

```java
    }
    MyUncheckedException(String message) {
        super(message);
    }
}
```

and then we use it

Listing 72 AYN-CheckedExc/CheckedExc.java

```java
public class CheckedExc {

    public static void main(String[] args) {

        try {
            goSleep(3*1000);
        } catch(InterruptedException ignored) {
            System.err.println("Interrupted");
        } finally {
            System.err.println("AFTER SLEEP");
        }

          // handling all exceptions
        try {
            goSleep(-1);
        } catch(InterruptedException e) {
            System.err.println("Interrupted");
        } catch(Exception e) {
            System.err.println("Handling exception: " +
                                           e.getMessage());
        } finally {
            System.err.println("GOING ON");
        }

          // here MyUncheckedException is  n o t  handled
          // so the program  w i l l  crash (but 'finally'
          // clause will be executed anyway)
        try {
            goSleep(-1);
        } catch(InterruptedException e) {
            System.err.println("Interrupted");
        } finally {
            System.err.println("QUITTING");
        }
    }

    private static void goSleep(int time)
                                throws InterruptedException {
        if (time < 0)
```

```
                throw new MyUncheckedException("Negative time");

            System.out.println(
                    "Going to sleep for " + time +"ms");
            Thread.sleep(time);
            System.out.println("Waking up");
        }
    }
```

The program prints:

```
Going to sleep for 3000ms
Waking up
AFTER SLEEP
Handling exception: Negative time
GOING ON
QUITTING
Exception in thread "main" MyUncheckedException: Negative time
        at CheckedExc.goSleep(CheckedExc.java:40)
        at CheckedExc.main(CheckedExc.java:29)
```

As we can see, the string `"QUITTING"` was printed by the **finally** clause even though there was an unhandled exception and the program crushed.

12.6 Assertions

Another way of dealing with possible errors in our programs is by using *assertions* (line 34 of the listing 68 on page 104). After the keyword **assert**, we specify a condition (something with **boolean** value) and, after a colon, a message (which is in fact optional)

```
    assert boolExp : message;
```

At runtime, the value of boolExp is evaluated and if found **false**, the program will print the message and terminate by throwing an exception of type **AssertionError**. In our example, this is what will happen when the file is missing and the variable scfile is **null**. By default, assertions are disabled at runtime, but can be enabled by `-ea` switch[4] when running a program. Because assertions are sometimes *on* and sometimes *off*, there should be no side effects of using them. Normally, assertions check conditions that we are almost sure should always hold. If, however, the condition is *not* met, and an *unchecked* exception of type **AssertionError** *is* thrown, we should never even try to handle it, because it indicates a serious flaw in the program that just must be corrected.

[4]It is also possible to enable/disable assertions selectively for certain packages or individual classes.

Section 13

IO streams primer

13.1 Binary and text IO streams

An IO stream can be viewed as a sequence of data (ultimately, these are always just bytes) 'flowing' from a 'source' to a 'sink' (destination). All IO streams fall into two categories

- output streams: data from our programs (values of variables, objects, arrays etc.) are the "source" while the destination is a file, a socket, the console or even a region in memory;
- input streams: our programs (variables, arrays etc.) is now the destination, while the source might be a file, a socket, the keyboard, etc.

Java represents characters by their two-byte Unicode codes (the so called *code points*). Therefore, there is a problem with reading and writing texts: any text is written using some encoding, so Java must translate a byte or a sequence of bytes corresponding to a character into a two-byte code point and *vice versa*. Therefore, we have to distinguish byte (binary) streams, where bytes are treated just as bytes, without any modifications, and text (also called *character*) streams where bytes are subject to some transformations, dependent of the encoding in use. Hence, we end up with four types of IO streams:

- output byte (binary) streams: they all correspond to subclasses of the class **OutputStream** (e.g., **ByteArrayOutputStream**, **FileOutputStream**, **ObjectOutputStream**);
- output text streams: they correspond to subclasses of **Writer** (e.g., **BufferedWriter**, **CharArrayWriter**, **OutputStreamWriter**, **PrintWriter**, **StringWriter**);
- input byte streams: they correspond to subclasses of **InputStream** (e.g., **ByteArrayInputStream**, **FileInputStream**, **ObjectInputStream**, **StringBufferInputStream**);
- input character streams: they correspond to subclasses of **Reader** (e.g., **BufferedReader**, **CharArrayReader**, **InputStreamReader**, **StringReader**);

The main four classes mentioned above are *abstract*, what means that one cannot create objects of these types, only objects of their concrete subclasses.

Any Java process, as other processes, is given by the operating system three standard streams. They are represented by objects which are static fields of the class **System**

- System.out: representing the standard output stream (by default it is connected to the terminal's screen);
- System.in: representing the standard input stream (by default it is connected to the terminal's keyboard);
- System.err: representing the standard error stream (by default it is connected to the terminal's screen).

Both System.out and System.err are of type **PrintStream** which inherits (indirectly) from **OutputStream** (but is designed to handle text output) while System.in is of a type inheriting from **InputStream** (and is a binary stream). The difference between System.out and System.err is that the former is *buffered*, while the latter is not. When we write to a buffered stream, like System.out, we are in fact writing to a buffer which will be eventually flushed to its destination (by default, to the screen) in larger chunks. Sometimes such behavior is not desired, especially when we expect some exception handling or when we write to a socket. Printing to unbuffered streams (like System.err) is immediate — data is not stored in any buffer, it goes directly to the destination of the stream.

All I/O operations[5] may throw exceptions of types extending **IOException** and are *checked*; therefore, when using them, we have to handle possible failures somehow.

Let us consider the following example. In the simple program below, we first write, in a loop, individual bytes of a **long**, starting with the most significant one, to the file. Then we read these bytes back and reconstruct a **long** with the same value. Note the special form of the **try** clause, the so called *try-with-resources*. In the round parentheses, we create I/O streams and references to these streams. Note that they will be in scope of the **try** clause. Normally, we have to remember to close all streams that have been opened. The *try-with-resources* construct, however, takes care of closing streams no matter what happened, even if an exception has been thrown. Therefore, no **finally** clause is needed here.

Listing 73 KEP-WriteBin/WriteBin.java

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class WriteBin {
    public static void main (String[] args) {
        long before = -284803830071168L;
        System.out.println("before = " + before);
        try (
            OutputStream os =
                    new FileOutputStream("WriteBin.bin");
        ) {
            for (int i = 7; i >= 0; --i)
                os.write( (int)(before >> i*8) );
        } catch(IOException e) {
            e.printStackTrace();
            System.exit(1);
        }

        long after = 0;
        try (
            InputStream is =
                    new FileInputStream("WriteBin.bin");
        ) {
```

[5] except those on *PrintStream* objects, which 'consumes' possible exceptions in its methods

```
            for (int i = 0; i < 8; ++i)
                after = (after << 8) | is.read();
        } catch(IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
        System.out.println("after  = " + after);
    }
}
```

The program prints

```
before = -284803830071168
after  = -284803830071168
```

confirming that we have written and read bytes correctly. Note, that the created file has exactly 8 bytes and contains the full information about our **long**. Writing a long in the text form requires up to 20 characters (2.5 times more!). Moreover, conversion of a number to its string representation (and then back) is also a quite expensive operation.
In practice, we don't have to manipulate the bytes of variables to store them on disk in their binary form — as we will see later, there are special tools in the library that make such tasks trivial.

In the next example, we first read from **InputStream** (connected to just System.in). As this is a binary stream, by invoking **read** we get consecutive bytes (as **int**s), which we can cast on **char**. We will get −1, when the end of data is reached (note that −1 doesn't correspond to any legal character). This, however, will never happen with System.in, so we just detect the LF character ('\n') to stop reading. A text may contain characters encoded on two or three bytes — such characters will not be read correctly. However, we can pass the System.in object to the constructor of **InputStreamReader** which is kind of a 'translator' yielding an object behaving as a text stream (given a specific encoding). In principle, we could have passed it further to the constructor of **BufferedReader**: using this object we can read characters much more efficiently; moreover, it also supports the very convenient **readLine** method which allows us to read a whole line at once (as we will see in one of the following examples).

Listing 74 BHJ-BytesChars/BytesChars.java

```
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.Reader;
import static java.nio.charset.StandardCharsets.UTF_8;

public class BytesChars {
    public static void main(String[] args) {
        System.out.print(
                "Type something and press enter ==> ");
        InputStream is = System.in;
        try {
            char c = ' ';
            while (true) {
                int i = is.read();
                c = (char)i;
                if (c == '\r' || c == '\n') break;
                System.out.printf("%3d ('", i);
                System.out.println(c + "') =>" +
                    " digit:" + Character.isDigit(c) +
                    " letter:" + Character.isLetter(c) +
                    " white:" + Character.isWhitespace(c));
            }
        } catch(IOException e) {
            e.printStackTrace();
            return;
        }
          // we don't close 'is' here, as this is the
          // standard input and we will use it later
        System.out.print("Type something again ==> ");
        try (
            Reader rd =
                new InputStreamReader(System.in, UTF_8)
        ) {
            char c = ' ';
            while (true) {
                int i = rd.read();
                c = (char)i;
                if (c == '\r' || c == '\n') break;
                System.out.printf("%#5x ('", i);
                System.out.println(c + "') =>" +
                    " digit:" + Character.isDigit(c) +
                    " letter:" + Character.isLetter(c) +
                    " white:" + Character.isWhitespace(c));
            }
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

Running the program, we can get:

```
Type something and press enter ==> Żółć
197 ('Å') => digit:false letter:true white:false
187 ('»') => digit:false letter:false white:false
195 ('Ã') => digit:false letter:true white:false
179 ('³') => digit:false letter:false white:false
197 ('Å') => digit:false letter:true white:false
130 ('') => digit:false letter:false white:false
196 ('Ä') => digit:false letter:true white:false
135 ('') => digit:false letter:false white:false
```

```
Type something again ==> Żółć
0x17b ('Ż') => digit:false letter:true white:false
0xf3 ('ó') => digit:false letter:true white:false
0x142 ('ł') => digit:false letter:true white:false
0x107 ('ć') => digit:false letter:true white:false
```

As we can see, in the first case multi-byte characters (as, e.g., in Żółć) are not read correctly. This is because (at least under Linux) the text from the console is in UTF-8 encoding, in which a single character may occupy 1, 2, 3, or even four bytes. The stream System.in is, however, a byte (binary) stream, so each **read** consumes one byte, which not necessarily corresponds to any character, as it may be only a part of a multi-byte character.
The situation is different in the second case — here we 'wrap' System.in in **InputStreamReader** (which behaves as a text stream), passing also the correct encoding. Object of this wrapper (*decorator*) class behaves as a *text* stream, so each **read** consumes one *character*, no matter how many bytes it takes.

Let us show now how one can read and write text files, what is a very common task. This can be done in various ways, so consider the program below as an example, not necessarily the best in all situations. Note that when dealing with text files, one should always specify the encoding of all input/output files.

Listing 75 KFE-GrepNew/GrepNew.java

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import static java.nio.charset.StandardCharsets.UTF_8;

public class GrepNew {
    public static void main(String[] args) {
        String iFileName = "alice.txt";
        String oFileName = "grep_alice.txt";
        String wordSearchedFor = "Cheshire";

        Path filein = Paths.get(iFileName);
        if (!Files.exists(filein)     ||
            !Files.isReadable(filein) ||
             Files.isDirectory(filein)) {
            System.out.println("Invalid input file!!!");
            return;
        }

        try (
               // UTF8 is the default in nio classes,
               // but not in older io classes
            BufferedReader br =
                Files.newBufferedReader(filein, UTF_8);
            BufferedWriter bw =
                Files.newBufferedWriter(
                    Paths.get(oFileName), UTF_8))
        {
            String line;
            int lineNo = 0;
            while ( (line = br.readLine()) != null) {
                ++lineNo;
                if (line.indexOf(wordSearchedFor) >=0)
                    bw.write(String.format("Line %3d: %s%n",
                                           lineNo,line));
            }
            System.out.println("Results written to " +
                                 oFileName);
        } catch(IOException e) {
            System.out.println("Something wrong");
            System.exit(1);
        }
    }
}
```

The program creates the file ***grep_alice.txt*** containing

```
Line 1429:   `It's a Cheshire cat,' said the Duchess, `and that's why.
Line 1437:   I didn't know that Cheshire cats always grinned; in fact, I
Line 1561: the Cheshire Cat sitting on a bough of a tree a few yards off.
Line 1567:   `Cheshire Puss,' she began, rather timidly, as she did not at
Line 2234: be a grin, and she said to herself `It's the Cheshire Cat:  now I
Line 2270:   `It's a friend of mine--a Cheshire Cat,' said Alice:  `allow me
Line 2317:   When she got back to the Cheshire Cat, she was surprised to
```

Here, we used classes from the ***java.nio*** package, which is newer than ***java.io*** and usually recommended. Objects of class **Path** represent the names of files in the current file system (not necessarily existing ones). We can create such objects by calling the static factory method **get** and passing (absolute or relative) name of a file. Then one can check if such a file exists, whether it is readable, executable, what its size or last modification time is, etc.

Note also the form of the **try** clause. Here, we used again the *try-with-resources* construct. Before the opening brace, in round parentheses, we create objects representing "resources" — in this case streams. These could be also other types or resources, like data base connections; what is important is that they have to be *closeable* (in other words, they have to implement the **Closeable** interface). If there are more than one, as here, we separate them by a semicolon. As we remember, the benefit of this form of the **try** clause is that we don't have to bother with closing the resources — they will be automatically closed whether an exception has occurred or not. Moreover, this form inserts variables declared in parentheses to the scope of the **try** clause, but *not* to the outer scope, keeping the outer scope unpolluted by variables that are not needed there.

In the loop reading the file, we call **readLine** on the **BufferedReader** object. In this way, we can read the file line by line not bothering about detecting the LF character ourselves. When the end of file is reached, **readLine** returns **null**, otherwise it returns the next line as a string (with the LF character *chopped off*).

For completeness, another version of essentially the same program is shown below. Here, we don't use classes from the ***java.nio*** package, but from an older (but still used and still necessary) ***java.io*** package.

Listing 76 KFD-Grep/Grep.java

```java
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Grep {
    public static void main(String[] args) {
        File filein = new File("alice.txt");
        String wordSearchedFor = "Cheshire";

        if ( !filein.exists()      ||
             !filein.canRead()     ||
              filein.isDirectory()  ) {
            System.out.println("Invalid input file !!!");
            System.exit(1);
        }
        File fileou = new File("grep_" + filein.getName());
        BufferedReader br = null;
        BufferedWriter bw = null;
        String LF=System.getProperty("line.separator");

        try {
            br = new BufferedReader(
                         new FileReader(filein));
            bw = new BufferedWriter(
                         new FileWriter(fileou));
            String line;
            int lineNo = 0;
            while ( (line = br.readLine()) != null) {
                ++lineNo;
                if (line.indexOf(wordSearchedFor) >=0)
                    bw.write(String.format("Line %2d: %s%s",
                                           lineNo,line,LF));
            }

        } catch (IOException e) {
            System.out.println("Problems with reading");
        } finally {
            try { if (br != null) br.close(); }
            catch(IOException ignore) { }
            try { if (bw != null) bw.close(); }
            catch(IOException ignore) { }
            System.out.println("Results written to " +
                                fileou.getAbsolutePath());
        }
    }
}
```

There are some differences to be noted: class **File** which, to some extend, plays the rôle of the class **Path** from the ***java.nio.file*** package. Notice also, that **BufferedReader** must be created in two steps: first we create 'raw' byte stream of type, e.g., **FileInputStream**, then we pass it to the constructor of **InputStreamReader** with, as the second argument, the required encoding, and then this to the constructor of the **BufferedReader**. In the example above there are only two steps: to the constructor of **BufferedReader** we pass directly an object of type **FileReader**, but then there is no way to specify the encoding — default system encoding will be assumed. Note how *try-with-resources* simplifies operations on IO streams; in the program above, we don't use it and therefore the somewhat complicated **finally** clause is needed (as **close** may itself also throw an exception).

It is sometimes useful to a read text file into a single string (for example, to process it with regular exceptions). This is a very common task, and it can be accomplished, for example, as shown below

Listing 77 KFI-File2Str/File2Str.java

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import static java.nio.charset.StandardCharsets.UTF_8;

public class File2Str {
    public static void main(String[] args) {
        String text = null;

        try {
            byte[] bytes =
                Files.readAllBytes(Paths.get("pangram.txt"));
            text = new String(bytes, UTF_8);
        } catch(IOException e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("***** File read (1):\n" + text);

          // simpler way, since java 11
        try {
            text = Files.readString(
                                Paths.get("pangram.txt"), UTF_8);
        } catch(IOException e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }
        System.out.println("***** File read (2):\n" + text);
    }
}
```

13.2 *StreamTokenizer class*

The last example illustrates the **StreamTokenizer** utility class. It reads from a text file and splits the input into 'tokens' – single pieces of information (treating spaces as separators, but this can be changed). After reading a token with the **nextToken** method, the field ttype contains information about the type of this token in the form of a predefined integer constant: TT_NUMBER if the token can be interpreted as a number, TT_WORD if it's a string, TT_EOL if it's the end-of-line character, TT_EOF if the end of file has been reached. When the token is a number or a string, one can get their values from the fields nval (**double**) or sval (**String**), respectively.

For example, for a data file like this

```
Tokyo 38 Delhi
25.7     Shanghai 23.7 SaoPaulo 21 Mumbai 21
```

the following program

Listing 78 HUS-Tokens/Tokenizer.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.StreamTokenizer;
import java.nio.file.Files;
import java.nio.file.Paths;
import static java.io.StreamTokenizer.TT_EOF;
import static java.io.StreamTokenizer.TT_NUMBER;
import static java.io.StreamTokenizer.TT_WORD;

public class Tokenizer {
    public static void main(String[] args) {
        double result = 0;
        StringBuilder sb = null;
        try ( // UTF8 assumed
            BufferedReader br =
                Files.newBufferedReader(
                    Paths.get("Tokenizer.dat")))
        {
            StreamTokenizer sTok = new StreamTokenizer(br);
            sTok.eolIsSignificant(false);
            sTok.slashSlashComments(true);
            sTok.slashStarComments(true);
            sb = new StringBuilder();
            while (sTok.nextToken() != TT_EOF) {
                switch (sTok.ttype) {
                case TT_NUMBER:
                    result += sTok.nval;
                    break;
                case TT_WORD:
                    sb.append(" " + sTok.sval);
                    break;
                }
            }
        } catch (FileNotFoundException e) {
            System.err.println("Input file not found");
            return;
        } catch(IOException e) {
            System.err.println("IO Error");
            return;
        }
        System.out.println("Total population: " + result);
        System.out.println("Cities: " +
                    sb.toString().substring(1));
    }
}
```

will print

```
Total population: 129.4
Cities: Tokyo Delhi Shanghai SaoPaulo Mumbai
```

13.3 IO cheat sheet

Let us summarize, as a reference, basic forms of reading from or writing to IO streams.

13.3.1 Reading/writing binary files byte-by-byte

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;

// ...

    try (
        InputStream  fis = new FileInputStream("fi.bin");
        OutputStream fos = new FileOutputStream("fo.bin")
    ){
        int n;
        while ( (n = fis.read()) != -1) {
            char c = (char)n;
            // ...
            System.out.print(c);
            fos.write(n);
        }
    } catch(IOException e) { /* ... */ }
```

To make reading/writing more efficient, one may also use buffered streams:

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;

// ...

    InputStream fis =
        new BufferedInputStream(
            new FileInputStream("fi.bin"));
    OutputStream fos =
        new BufferedOutputStream(
            new FileOutputStream("fo.bin"))
```

13.3.2 Reading a binary file into an array of bytes

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

// ...

    byte[] ba = null;
    try {
        ba = Files.readAllBytes(Paths.get("fi.bin"));
    } catch(IOException e) { /* ... */ }
    for (byte b : ba) System.out.print((char)b);
```

13.3.3 Reading/writing text files line-by-line

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import static java.nio.charset.StandardCharsets.UTF_8;

// ...

    try (
        BufferedReader br =
            Files.newBufferedReader(
                Paths.get("fi.txt"), UTF_8);
        BufferedWriter bw =
            Files.newBufferedWriter(
                Paths.get("fo.txt"), UTF_8)
    ){
        String line;
        while ( (line = br.readLine()) != null) {
            System.out.println(line);
            bw.write(line);
            bw.newLine();
        }
    } catch(IOException e) { /* ... */ }
```

13.3.4 Reading/writing text files character-by-character

One can use BufferedReader as in the example above but, instead of readLine, call read

```
      // Reads one character and returns it as an int.
      // Returns -1 when the end of file has been reached
    int  n = br.read();
    char c = (char)n;
```

To read text from the keyboard, you can use

```
    import java.io.InputStreamReader;
    import java.io.Reader;
    import java.io.IOException;
    import static java.nio.charset.StandardCharsets.UTF_8;

        try (
            Reader rd =
                new InputStreamReader(System.in, UTF_8)
        ) {
            while (true) {
                int i = rd.read();
                // ... a condition to stop the loop
                c = (char)i;
                System.out.print(c);
            }
        } catch(IOException e) {
            e.printStackTrace();
        }
```

Section 14

Regular expressions

Regular expression (**regex**) is a sequence of characters which defines a pattern that we want to search for in a string (generally, in a text which may be arbitrary long). Regexes are 'compiled' into a form resembling functions and executed by the so called *regular expression engines* — almost all contemporary languages support regular expressions (built into the language or as part of their standard libraries). The theory behind regexes is rather involved and its full understanding requires quite advanced mathematical knowledge; it was developed by an outstanding logician Stephen Cole Kleene (pronounced KLAY-nee) in 1950s and first used in practice in early implementations of Unix text processors and utility programs (Ken Thompson). Almost all modern implementations of regular expression engines are based on Larry Wall's implementation in his Perl programming language (late 1980s).

Details on the Java implementation: see Oracle's documentation.[6]

14.1 Basic concepts

Suppose we are looking for a word, say 'elephant', in a text. Then the regular expression which defines this word will be just `"elephant"`. But what if we have, in our text, the word 'Elephant'? Of course, this won't match, because the first letter differs. Or, we look for 'cat' but only if it is a separate word, not part of another word (like in 'tomcat' or 'caterpillar'). Or, we are looking for numbers (sequences of digits) but we don't know in advance what numbers occur in our text and of what length (number of digits) they are. All these problems can be easily solved with the help of regular expressions which allow us to formulate such requirements as 'a sequence of letters', 'any uppercase letter followed by a dot', 'a sequence of at least four but at most seven digits the first of which is not 0', 'two words separated by one or more spaces or TAB characters', etc.

14.1.1 Classes

Classes define sets of characters. They are specified in square brackets — a hyphen between characters denotes a range, a 'hat' (^) at the beginning denotes negation, the `&&` symbol stands for ANDing. For example:

- `[abc]` — set of three letters: 'a', 'b' and 'c',
- `[a-d]` — set of four letters: 'a', 'b', 'c' and 'd',
- `[a-cu-z]` — set of lowercase letter from ranges `[a-c]` and `[u-z]`,
- `[a-zA-Z]` — set of lower- and uppercase Latin letter,
- `[a-zA-Z0-9_]` — set of all Latin letter and digits, and underscore,
- `[^0-9]` — any character, but *not* a digit,
- `[a-z&&[^i-n]]` — any character in the range `[a-z]` but simultaneously *not* in the range `[i-n]` (therefore equivalent to `[a-ho-z]`).

As we can see, there is a way to AND, but what about ORing? This can also be achieved with symbol `|`: regex `cat|dog` will look for 'cat' OR 'dog'.

[6]https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/regex/Pattern.html

14.1.2 Predefined classes

Some most useful classes are predefined and denoted by special symbols; sometimes it is just one letter after a backslash — in these cases uppercase letter means the same as the corresponding lowercase symbol, but negated. For example:

- `\d` — any digit, `\D` — any non-digit,
- `\s` — any white character (space, tab, new line), `\S` — anything except white character,
- `\w` — any Latin letter, digit or underscore, `\W` — negation of `\w` (also non-Latin letters if UNICODE_CHARACTER_CLASS option is enabled, see sec. 14.5.4),
- `.` — any character except the new line (or including it, if DOTALL option has been selected, see sec. 14.5.2),
- `\p{P}` — any punctuation character, `\P{P}` — anything but not punctuation,
- `\p{L}` — any letter, in any language, `\P{L}` — not a letter,
- `\p{Ll}` — any lowercase letter, in any language,
- `\p{Lu}` — any uppercase letter, in any language,
- etc.,

14.1.3 Special locations

There are symbols which denote not characters but rather special locations in the text being analyzed. For example:

- `\b` — word boundary — just before or just after a word (a word is a sequence of characters matching `\w`),
- `^` — beginning of the text (or of a line, if MULTILINE option is enabled, see sec. 14.5.1),
- `$` — end of the text (or of a line, if MULTILINE option is enabled).

14.1.4 Quantifiers

You can put a so called *quantifier* just after an element of a regex. It then determines a possible number of repetitions of this element. For example:

- `+` — once or more,
- `*` — any number of occurrences (including zero),
- `?` — once or not at all,
- `{n,m}` — number of occurrences in the range $[n, m]$,
- `{n,}` — at least `n` occurrences.

All these quantifiers are by default **greedy**. It means that the regex engine will try to find the *longest* possible match. For example, if our regex is `a.*z` and the text is `"abzczdz"`, then the whole text will be found as the match, even though substrings `"abz"` and `"abzcz"` would be also possible (but are shorter). If this is not what we want, we can make a quantifier **reluctant**, i.e., it will try to find the *shortest* match — in the above example `"abz"` would be found. To make a quantifier reluctant, we just add a question mark (?). Thus, continuing the above example, the regex `a.*?z` would find the shortest match (`"abz"`).

There exist the third kind of quantifiers, the so called **possessive** quantifiers, denoted by a plus symbol (+). It is, in a sense, even more greedy that greedy quantifier, because it never steps back. Normally, for a greedy quantifier, something like `.*` consumes everything and then, if there is no match, the matcher slowly backs off: it makes one step backward to see if there is a match now, if not, it steps back one more character again, and so on.

For example, let us assume that the regex is `"a.*bc"` (so `.*` is, by default, greedy) and the text is `"abcdbc"`. First, after 'a', the `.*` will consume everything. But there is no 'bc' after 'everything', so the matcher steps back one character and now there is 'c' at the end. This is still not 'bc', so the matcher makes one more step back and now it has 'bc' at the end — matching succeeds with the whole text `"abcdbc"` as the match.

Now suppose the quantifier is reluctant: `"a.*?bc"`. After 'a' the matcher consumes the shortest substring matching `.*`, that is nothing. There is no match, because there is no 'bc', so the matcher makes one step *forward* and indeed there is 'b'. After making one more step, there is 'bc', so matching succeeds and the matched substring is `"abc"`.

Now the possessive case: `"a.*+bc"`. The `.*` consumes everything, and there is no 'bc' after that. Possessive matcher never steps back, so the matching fails. Generally, you can live without possessive quantifiers, and in fact in many languages they are not supported at all. However, when they are appropriate, they make the whole process of searching for a match faster (because the matcher does not have to remember intermediate states, as it will never need to go back).

14.2 Regular expressions in methods of class String

There are some very useful methods in class **String** which use regular expressions:

1. method **String::split**;
2. method **String::replaceAll**;
3. method **String::matches**;

The first, **split**, invoked on a string, takes a regex and then splits the string into parts separated by substrings matching the regex — it returns an array of **String**s. Note that the regex specifies *separators*, not what we are looking for! For example,

`"Łódź - 0.7; London - 8.8; Tokyo - 13.6".split("\\P{L}+")`

will produce a three-element array containing the names of the three cities: the separator here is 'non-empty sequence of any non-letters' (note the capital 'P'). Note that there is a separator (sequence of non-letters) at the end of the input string. Therefore, there should be an empty string as the last element of the resulting array of strings. However, trailing empty strings are discarded. This does not apply to leading separator: in such a case, we *will* get an empty string as the first element of the array.

Regexes can also be ORed, as the following example illustrates: here, the separator is defined as non-empty sequence of white characters (between word boundaries) OR a punctuation mark surrounded, perhaps, by sequences of white characters:

Listing 79 GXP-Split/Splitting.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import static java.nio.charset.StandardCharsets.UTF_8;

public class Splitting {
    public static void main(String[] args) {
        try (
            BufferedReader br = Files.newBufferedReader(
                    Paths.get("Splitting.txt"), UTF_8)
        ) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] array = line.split(
                        "(\\b\\s+\\b|\\b\\s*\\p{P}\\s*\\b)");
                System.out.print("|");
                for (String s:array) System.out.print(s+"|");
                System.out.println();
            }
        } catch(IOException e) {
            System.out.println(e.getMessage());
            System.exit(-1);
        }
    }
}
```

If the file contains

```
John,Mary    Charles    ;    Zoe
carrot   parsley:potato
```

then the program prints

```
|John|Mary|Charles|Zoe|
|carrot|parsley|potato|
```

The second method, **replaceAll**, takes a regex and a string and returns a string with all occurrences of substrings that match the regex replaced by the given string, e.g.,

```
"cat, caterpillar, tomcat, cat".replaceAll("\\bcat\\b", "dog")
```

will return `"dog, caterpillar, tomcat, dog"` (note that `cat` in `tomcat` will not be replaced because there is no word boundary before the letter 'c', and similarly for `caterpillar`). Invoking `s.replaceAll(regex, rep)` on a string `s` is equivalent to `Pattern.compile(regex).matcher(s).replaceAll(rep)`. In particular, we can refer to capture groups, specified in the regex string, in the replacement string (see sec. 14.4 on page 127).

The third method, **matches**, takes a regex and answers the question 'does the whole string match the regex'. For example

```
"Madagascar!".matches("\\p{L}+")
```

will return **false**, because the string contains a character which is not a letter, while

```
"Madagascar!".matches("\\p{L}+.*")
```

will return **true**.

14.3 Patterns and Matchers

Regexes, before being used, must be 'compiled' (in the case of the aforementioned methods of class **String**, this is done automatically). This is done by invoking the static method of class **Pattern compile(regex)**: it returns an object of type **Pattern** which represents the 'compiled' form of our regex (we can view it as some sort of a function). Having a compiled regex (pattern), we invoke on it **matcher(text)** where text is the text to be analyzed (as a string). It returns an object of type **Matcher** representing the result:

```
String text = "A text";
String reg  = "a regex";
Pattern pat = Pattern.compile(reg);
Matcher m = pat.matcher(text);
```

or, if we do not need to reuse the pattern:

```
String text = "A text";
String reg  = "a regex";
Matcher m = Pattern.compile(reg).matcher(text);
```

Very often, we want to use the same pattern many times, but for different texts — for example for each line of a text file. We then compile our regex once, and then we can call **matcher** on the same object of type **Pattern** and get matchers corresponding to subsequent lines. Or, we can even create a matcher once only, passing, for example, an empty string as a text, and then reset it for subsequent lines:

```
Matcher matcher = null;
try {
    matcher = Pattern.compile("a regex").matcher("");
} catch(PatternSyntaxException e) {
    System.err.println("Wrong pattern?");
    e.printStackTrace();
    System.exit(1);
}

try (
    BufferedReader br =
            Files.newBufferedReader(
                Paths.get("file.txt"), UTF_8)
) {
    String line;
    while ((line = br.readLine()) != null) {
        matcher.reset(line);
        // ... processing information
        // ... from a single line
    }
} catch (IOException e) {
    // ...
}
```

By invoking matcher's various methods, we can extract the information we need. Two basic methods which actually do the search are **matches** and **find**:

- **matches()** checks if the whole text matches the given regex (the one, on which we have called **matcher**) and returns **true** or **false**; note that it is not enough that the text contains some substrings matching the pattern — it must be the whole text;

- **find()** looks for *substrings* matching the regex (see examples below) and also returns **true** or **false**. It can be called several times; each time we invoke it, it starts searching from the location within the string after the previous match.

Matcher objects always remember the location where the last operation has finished. For successful invocation of **matches**, it will probably be the end of the input text, for an unsuccessful — the place where the matcher 'realized' that **matches** cannot succeed. Also for **find**, the location of the last successful match is remembered, so the next invocation of **find** will start looking for the next match. If we want to start from the beginning (for example, after calling **matches**), we can invoke `matcher.reset()`. The **reset** function is overloaded — a version taking a text makes the matcher 'forget' the old text and sets a new one that will be analyzed.

14.4 Capturing groups

Capturing group allows us to remember a part of text matching a pattern, so that we can use it later. A group is created when a part of a regex is enclosed in a pair of round parentheses. For example, the following regex

```
"(\\p{L}+)\\P{L}+(\\p{L}+)"
```

will match a string containing two words separated by a non-empty sequence of non-letters. However, both words will be remembered by the matcher as group number 1 and group number 2, and after a successful match we can get them by calling `matcher.get(1)` and `matcher.get(2)`, respectively. Groups may be nested and are numbered starting from 1, group number 0 being the whole substring matched, containing all the groups defined inside. Numbers are assigned according to the order in which opening parentheses of the groups are encountered — each group extends to the corresponding closing parenthesis. For example, in the program below, there will be three groups:

- one containing the first word, after, perhaps, some leading spaces, and followed by a sequence of any characters (which will not enter any group);
- then a group with two sequences of digits separated by a hyphen;
- then a group consisting of only the second of these two sequences of digits.

```
String reg = "\\s*(\\w+).*?(\\d+-(\\d+))";
  //             1          2     3
String text = "   Einstein Albert, 1879-1955";
Matcher m = Pattern.compile(reg).matcher(text);
System.out.println("Matches? " + m.matches());
System.out.println("# of groups " + m.groupCount());
for (int i = 1; i <= m.groupCount(); ++i)
    System.out.println(i + ": " + m.group(i));
```

The program prints

```
Matches? true
# of groups 3
1: Einstein
2: 1879-1955
3: 1955
```

Note the use of **groupCount** method — it reports the number of matched groups, *not* counting the special group(0). Note also that in this regex we used the reluctant `.*?` — without the question mark, `.*` would consume the first three digits of the first number!

Instead of numbering groups, we can assign names to them; the syntax is `(?<name>`, where `name` is any unique name. We then refer to such groups by invoking `matcher.group("name")`. Both ways of accessing groups, by their number or by names, are illustrated in the program below:

Listing 80 GXX-RegGroups/RegGroups.java

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegGroups {
    public static void main (String[] args) {
        String str = "12-07-2014 xx 6-6-2010 yy 1-11-2011";
        String pat1 = "(\\d{1,2})-" +
                      "(\\d{1,2})-" +
                      "(\\d{4})";
        String pat2 = "(?<day>\\d{1,2})-" +
                      "(?<month>\\d{1,2})-" +
                      "(?<year>\\d{4})";

        Matcher m = Pattern.compile(pat1).matcher(str);
        System.out.println("Unnamed groups");
        while (m.find()) {
            System.out.println(m.groupCount()+" groups:");
            System.out.print("D: "+m.group(1)+", ");
            System.out.print("M: "+m.group(2)+", ");
            System.out.print("Y: "+m.group(3)+'\n');
        }

        m = Pattern.compile(pat2).matcher(str);
        System.out.println("Named groups");
        while (m.find()) {
            System.out.println(m.groupCount()+" groups:");
            System.out.print("D: "+m.group("day")+", ");
            System.out.print("M: "+m.group("month")+", ");
            System.out.print("Y: "+m.group("year")+'\n');
        }
    }
}
```

which prints

```
Unnamed groups
3 groups:
D: 12, M: 07, Y: 2014
3 groups:
D: 6, M: 6, Y: 2010
```

```
3 groups:
D: 1, M: 11, Y: 2011
Named groups
3 groups:
D: 12, M: 07, Y: 2014
3 groups:
D: 6, M: 6, Y: 2010
3 groups:
D: 1, M: 11, Y: 2011
```

Inside a regex, we can refer to the previously found groups using `\n` expressions, where `n` is the number of the group we are interested in (this is called *backreference*). In the following example we want to find fragments of a text enclosed in apostrophes or double quotes. We thus look for either an apostrophe or a double quote, but we have to ensure that the closing quote is of the same type as the opening one:

```
String reg = "(['\"])[^'\"]*\\1";
String text = "'abc' xx \"def\" yy \"ghi' zz";
Matcher m = Pattern.compile(reg).matcher(text);
while (m.find())
    System.out.println(m.group());
```

The program prints

```
'abc'
"def"
```

Note that the sequence `ghi` is enclosed in non-matching quotes, so, correctly, it has not been found.

Backreferences can also be used in replacement texts when using **String.replaceAll** method. There is also such a method in class **Matcher** and in fact

```
str.replaceAll(regex,text)
```

is equivalent to

```
Pattern.compile(regex).matcher(str).replaceAll(text)
```

For example, suppose we have a file with full names in the order first name and last name separated by at least one space, but what we want is to have them in reversed order: first the last name and then the first name separated by exactly one space. In the replacement text, we refer to groups using `$n` notation, where `n` is the group number, as in the example below:

```
String orig  = "John  Smith, Mary   Brown";
String modif = orig.replaceAll(
        "(\\p{L}+)\\s+(\\p{L}+)", "$2 $1");
System.out.println("Orig : " + orig + '\n' +
                   "Modif: " + modif);
```

The program prints

```
Orig : John  Smith, Mary   Brown
Modif: Smith John, Brown Mary
```

We could have used named groups instead; then in the replacement string we refer to groups using `${name}` notation:

```
String orig  = "John  Smith, Mary   Brown";
String modif = orig.replaceAll(
        "(?<first>\\p{L}+)\\s+(?<last>\\p{L}+)",
        "${last} ${first}");
System.out.println("Orig : " + orig + '\n' +
                   "Modif: " + modif);
```

with the result as before.

14.5 Option flags

There are several options which influence the process of compilation of a pattern. They can be specified in two ways: as an additional argument to **compile** or embedded directly into the regex. Options are defined in class **Pattern** as named static integer fields, which may be ORed, if we want several of them. If we choose to specify them embedded in our regex, we do it by including expression `(?<letters>)` where `<letters>` can be one or more letters denoting different options. For example, suppose we want to turn on the options DOTALL and MULTILINE; we can do it like this

```
import static java.util.regex.Pattern.*;
// ...
String regex = "...";
Pattern p = Pattern.compile(regex, DOTALL | MULTILINE);
```

or like this

```
String regex = "(?sm)...";
Pattern p = Pattern.compile(regex);
```

as the letter 's' denotes DOTALL ('s' because it's called 'single-line' in Perl) and 'm' stands for MULTILINE. Note that enabling options usually induces some performance penalty, so don't do it if it's not necessary.

Some of the most useful options are described below (there are others).

14.5.1 *MULTILINE*

This option enables the so called 'multi-line mode', which means that expressions ^ and $ match just at the beginning of each a line and just at the and of each line, respectively. Normally, these symbols match only at the beginning and at the end of the whole input text; in multi-line mode those are still available as `\A` and `\Z`.
This option can also be enabled by embedding the flag `(?m)`.
For example, the following program

```
String s = "A 123\nD 456";
Matcher m1 = Pattern.compile("^\\w").matcher(s);
Matcher m2 = Pattern.compile("(?m)^\\w").matcher(s);
System.out.print("m1 : ");
while(m1.find())
    System.out.print(m1.group() + " ");
System.out.print("\nm2 : ");
while(m2.find())
    System.out.print(m2.group() + " ");
System.out.println();
```

will print

```
m1 : A
m2 : A D
```

because in the first case we only looked for a letter at the beginning of the entire input, while in the second case — at the beginning of each line separately.

14.5.2 *DOTALL*

This option changes the interpretation of the dot (.). By default, it denotes 'any character except new line', while with this option enabled a dot matches any character, *including* the new line.
This option can be enabled by embedding the flag `(?s)`.
For example, the following program

```
String s = "A 123\n456 B";
boolean b1 = Pattern.compile(
        "\\w.*\\w").matcher(s).matches();
boolean b2 = Pattern.compile(
        "(?s)\\w.*\\w").matcher(s).matches();
System.out.println("b1=" + b1 + "; b2=" + b2);
```

will print

```
b1=false; b2=true
```

because in the first case `.*` consumes everything up to the new line (but not any further) and there is no letter at the end, while in the second case everything, including the new line character, will be consumed by `.*` reaching the final letter 'B'.

14.5.3 *CASE_INSENSITIVE* and *UNICODE_CASE*

Enabling the CASE_INSENSITIVE option makes the lower- and uppercase letters indistinguishable. However, this works for Latin letters only. If it should apply to non-Latin letters too, we have to enable additionally the option UNICODE_CASE.
The options can be enabled by embedding the flags `(?i)` and `(?u)`, respectively, or we can enable both of them by `(?iu)`.
For example, the following program

```
String s = "PaRiS";
boolean b1 = Pattern.compile(
        "paris").matcher(s).matches();
boolean b2 = Pattern.compile(
        "paris", Pattern.CASE_INSENSITIVE)
        .matcher(s).matches();
System.out.println("b1=" + b1 + "; b2=" + b2);
```

will print

```
b1=false; b2=true
```

as in the second case we ignore the case of letters, so `PaRiS` and `paris` are considered equivalent.

14.5.4 *UNICODE_CHARACTER_CLASS*

When this option is enabled, some classes of characters, which normally apply only to ASCII characters, take into account all Unicode characters.
The options can be enabled by embedding the flags `(?U)` (note the *capital* U). Enabling UNICODE_CHARACTER_CLASS implies also UNICODE_CASE.
For example, the following program

```
String s = "Żółć";
boolean b1 = Pattern.compile(
        "\\w+").matcher(s).matches();
boolean b2 = Pattern.compile(
        "\\w+", Pattern.UNICODE_CHARACTER_CLASS)
        .matcher(s).matches();
System.out.println("b1=" + b1 + "; b2=" + b2);
```

will print

```
b1=false; b2=true
```

as in the second case `\w` matches also non-ASCII letters.

14.6 Some examples

In the following example we look for names in file ***input.txt*** (in UTF-8 encoding): we make a rather naïve assumption that a string with the first letter in uppercase and the remaining characters in lowercase must be a name... Then we prints a list of names together with numbers of occurrences and lists of line numbers where a given name appeared. We use an auxiliary class **Name**

Listing 81 GXR-RegExNames/Name.java

```java
import java.util.ArrayList;

public class Name {
    private String name;
    private ArrayList<Integer> list;
    private int total = 0;

    public Name(String name, int lineNo) {
        this.name = name;
        list = new ArrayList<Integer>();
        list.add(lineNo);
        total = 1;
    }

    public void addNum(int lineNo) {
        if (list.get(list.size()-1) != lineNo)
            list.add(lineNo);
        ++total;
    }

    @Override
    public String toString() {
        return name + " (" + total + ") in lines " + list;
    }
}
```

and the program might look like this:

Listing 82 GXR-RegExNames/RegEx.java

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Map;
import java.util.TreeMap;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;
import static java.nio.charset.StandardCharsets.UTF_8;

/*  Reads file "input.txt" (in UTF-8) and selects
 *  names (strings starting with capital letter with
 *  other letters in lowercase. Then prints a list of
 *  names together with numbers of occurences and lists
 *  of line numbers where a given name appeared.
 *  Uses class Name.
*/
public class RegEx {

    private Map<String,Name> map;

    public static void main(String[] args) {
        new RegEx("input.txt");
        System.exit(0);
    }

    public RegEx(String fileName) {
        Path filein = Paths.get(fileName);
        if (!Files.exists(filein)      ||
            !Files.isReadable(filein) ||
             Files.isDirectory(filein)) {
            System.err.println("Invalid input file !!!");
            System.exit(1);
        }
//          File input = new File(filename);
//          BufferedReader br = null;
        map = new TreeMap<String,Name>();
        try (
            BufferedReader br =
                Files.newBufferedReader(filein, UTF_8)
            ) {
            String line, name,
                   patt = "\\b\\p{Lu}\\p{Ll}+\\b";
            Pattern pattern = null;
            try {
                pattern = Pattern.compile(patt);
            } catch(PatternSyntaxException e) {
                System.err.println("Wrong pattern");
                System.exit(1);
            }
            Matcher matcher = pattern.matcher("");

            int lineNo = 0;
            while ((line = br.readLine()) != null) {
                lineNo++;
                matcher.reset(line);
                if (matcher.find()) {
                    do {
                        name = matcher.group();
                        if (!map.containsKey(name))
                            map.put(name,
                                    new Name(name,lineNo));
                        else
                            map.get(name).addNum(lineNo);
                    } while (matcher.find());
                }
            }
        } catch(IOException e) {
            System.err.println("Something wrong - exiting");
            e.printStackTrace();
            System.exit(1);
        }

        for (Map.Entry<String,Name> e : map.entrySet())
            System.out.println(e.getValue());
    }
}
```

The program below asks for a regex and a text (in a loop); it then prints information about matches found:

Listing 83 GXU-Regexes/Regexes.java

```java
import java.io.Console;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

```

```java
public class Regexes {
    public static void main(String[] args){
        Console console = System.console();
        if (console == null) {
            System.err.println("Console unavailable");
            System.exit(1);
        }
        while (true) {
            String reg = console.readLine(
                    "%nRegex ('q' to quit) -> ");
            if ("q".equals(reg)) return;
            String inp = console.readLine(
                      "Input string        -> ");
            Pattern pattern = Pattern.compile(reg);
            Matcher matcher = pattern.matcher(inp);

            boolean found = false;
            while (matcher.find()) {
                found = true;
                console.format("Found '%s' at %d-%d.%n",
                    matcher.group(),
                    matcher.start(),
                    matcher.end());
            }
            if(!found){
                console.format("No match found.%n");
            }
        }
    }
}
```

and a similar program with a graphical interface:

Listing 84 GXW-REExplorer/REExplorer.java

```java
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.regex.PatternSyntaxException;
import javax.swing.AbstractAction;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;
import static javax.swing.JDialog.DISPOSE_ON_CLOSE;
import static javax.swing.JFrame.EXIT_ON_CLOSE;

public class REExplorer {

    public static void main(String[] args) {
        new REExplorer();
    }

    private REExplorer() {
        final JFrame f = new JFrame("RE Explorer");
        f.setDefaultCloseOperation(EXIT_ON_CLOSE);

        final JLabel labx = new JLabel("REGEX:");
        final JTextField rege = new JTextField(40);
        final JButton gobu  = new JButton("Go!");
        labx.setFont(new Font("Dialog",Font.PLAIN,18));
        gobu.setFont(new Font("Dialog",Font.PLAIN,18));
        rege.setFont(new Font("Dialog",Font.PLAIN,18));
        rege.setBorder(BorderFactory.
                        createEmptyBorder(5,5,5,5));
        JPanel pans = new JPanel();
        pans.setLayout(new FlowLayout());
        pans.add(labx);
        pans.add(rege);
        pans.add(gobu);

        final JTextArea text = new JTextArea(10,40);
        text.setFont(new Font("Dialog",Font.PLAIN,18));
        text.setBorder(BorderFactory.
                       createTitledBorder(
                       "Enter text to be searched below"));

        AbstractAction act = new AbstractAction() {
            @Override
            public void actionPerformed(ActionEvent e) {
                showText(getMatches(rege.getText(),
                                    text.getText()));
            }
        };
        rege.addActionListener(act);
        gobu.addActionListener(act);

        JScrollPane scroll = new JScrollPane(text,
            JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
            JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout());
        panel.add(scroll,BorderLayout.CENTER);
        panel.add(pans,BorderLayout.SOUTH);

        f.setContentPane(panel);

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                f.pack();
                f.setLocationRelativeTo(null);
                f.setVisible(true);
            }
        });
    }

    private String getMatches(String regex, String text) {
        StringBuilder sb = new StringBuilder(200);
        String nL = System.getProperty("line.separator");

        sb.append("Pattern: \"" + regex + "\"" + nL +
                  "-- Text from here -----" + nL + text +
                  nL + "-- to here ------------" + nL + nL);
          // Compiling the pattern
        Pattern pattern = null;
        try {
            pattern = Pattern.compile(regex);
        } catch (PatternSyntaxException exc) {
            sb.append(
                  "Error: " + exc.getMessage() + nL);
            return sb.toString();
        }

        Matcher matcher = pattern.matcher(text);

          // does the whole text match the pattern?
        boolean match = matcher.matches();
        sb.append("matches() gives: " +
                  (match ? "YES" : " NO") + nL + nL);
          // groups (if any)
        if (match) {
            int gr = matcher.groupCount();
            sb.append(gr + " groups:" + nL);
            for (int i = 1; i <= gr; ++i)
                sb.append("  " + i + ": " +
                          matcher.group(i) + nL);
            sb.append(nL);
        }

        matcher.reset();

          // looking for matches inside the text
        boolean found = matcher.find();
        if (!found)
            sb.append("find() didn\'t find anything" + nL);
        else
            do {
                sb.append("find() found \"" +
                    matcher.group() + "\" at " +
                    (matcher.start()+1)  + "-" +
                    matcher.end() + nL);
            } while(matcher.find());

        return sb.toString();
    }

    private void showText(String text) {
        final JDialog dg = new JDialog();
        dg.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        JTextArea area = new JTextArea(20,45);
        area.setText(text);
        area.setBorder(BorderFactory.
                       createEmptyBorder(10,5,10,5));
        area.setFont(new Font("Monospaced",Font.PLAIN,20));
        area.setEditable(false);
        dg.add(new JScrollPane(area));
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                dg.pack();
                dg.setLocationRelativeTo(null);
                dg.setVisible(true);
            }
        });
    }
}
```

Another example:

Listing 85 GXY-RegFind/RegFind.java

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

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

        spl("Joe : Mary::Jane", ":");
        spl("Joe : Mary:Jane", "\\s*:\\s*");
```

```java

        mat("Joe : Mary:Jane", "[^:]*(:[^:]*){2}");
        mat("123 xxx ABCD", "\\.*\\d+[^A-Z]*[A-Z]{3,}");
        mat("Jane Crawford", "\\w+\\s+([A-Z]\\.)?\\s*\\w+");

        rep("a   b   c", "\\s+", " ");

        process("kot\\b",
                "Kot kot kotek");
        process("(kot)",
                "Kot kot kotek");
        process("\\b[A-Z][a-z]+\\b",
                "cat Dog hen Cow aHorse Z");
        process("^.*(kot).*(tek)",
                "Kot kot kotek");
        process("(a.*)(\\w).*",
                "a b c d c b x");
        process("(\\d{1,3}\\.){3}\\d{1,3}",
                " 1.1.1.2 12.12.34.231 234 xx 3.21.21.21 zz");
    }

    private static void process(String reg, String str) {
        Pattern p = Pattern.compile(reg);
        Matcher m = p.matcher(str);
        System.out.println("== match and find =======");
        System.out.println("STRING : " + str);
        System.out.println("REGEX  : " + reg);
        boolean gr = m.matches();
        System.out.println("Matches: " + gr);
        if (gr) {
            System.out.println("Groups : " + m.groupCount());
            for (int i = 1; i <= m.groupCount(); ++i) {
                System.out.println("Group " + i +
                           " = '" + m.group(i) + "'");
            }
        }
        m.reset();
        while (m.find()) {
            System.out.println(
                    "Found  " + ": " + m.group() +
                    " at " + m.start() + "-" + m.end());
        }
    }

    private static void spl(String str, String reg) {
        System.out.println("== split ================");
        System.out.println("STRING : " + str);
        System.out.println("REGEX  : " + reg);
        String[] s = str.split(reg);
        System.out.print(s.length + " terms:");
        for (int i = 0; i < s.length; ++i)
            System.out.print(" '" + s[i]+ "'");
        System.out.println();
    }

    private static void mat(String str, String reg) {
        System.out.println("== matches ==============");
        System.out.println("STRING : " + str);
        System.out.println("REGEX  : " + reg);
        System.out.println(str.matches(reg));
    }

    private static void rep(String str, String reg, String with) {
        System.out.println("== replace ==============");
        System.out.println("STRING : " + str);
        System.out.println("REGEX  : " + reg);
        System.out.println("WITH   : '" + with + "'");
        System.out.println(str.replaceAll(reg,with));
    }
}
```

List of listings

1 AAC-HelloWorld/Hello.java . . . 4
2 AAG-Literals/Literals.java . . . 12
3 AAL-Variables/Variables.java . . . 13
4 AAI-ReadKbd/ReadKbd.java . . . 16
5 AAJ-ReadGraph/ReadGraph.java . . . 17
6 ABY-RelOps/RelOps.java . . . 22
7 ACB-CondOp/Largest.java . . . 23
8 BAA-Bits/Bits.java . . . 25
9 AAP-BasicOps/BasicOps.java . . . 27
10 ABX-Ifs/LeapYear.java . . . 30
11 ACC-SimpleSwitch/SimpleSwitch.java . . . 31
12 ACD-Switch/Switch.java . . . 32
13 ACE-SwitchArrow/SwitchArrow.java . . . 33
14 ACF-SwitchMult/SwitchMult.java . . . 34
15 ACG-SwitchExpr/SwitchExpr.java . . . 34
16 ACH-SwitchEnum/SwitchEnum.java . . . 35
17 AFA-While/Prime.java . . . 36
18 AFB-WhileBis/Prime.java . . . 37
19 AFE-DoWhileDice/Dice.java . . . 39
20 AFH-ForPyram/Stars.java . . . 41
21 AFJ-ForWhileEuler/ForWhileEuler.java . . . 41
22 CYD-BasicArray/BasicArr.java . . . 45
23 CYB-SimpleArrays/SimpleArrays.java . . . 47
24 CYJ-Arr3D/Arr3D.java . . . 49
25 CYS-PassArr/PassArr.java . . . 53
26 BHK-StatFun/StatFun.java . . . 54
27 BHL-RecFun/RecFun.java . . . 55
28 AFR-SelSort/SelSort.java . . . 56
29 AFM-Overload/Overload.java . . . 57
30 AFN-Resol/Resol.java . . . 58
31 BGO-TrivPoint/TrivPoint.java . . . 60
32 BGO-TrivPoint/Main.java . . . 61
33 BGI-VerySimple/VerySimple.java . . . 63
34 BGI-VerySimple/Main.java . . . 63
35 BGP-Point/Point.java . . . 64
36 BGR-BasicClass/Person.java . . . 66
37 BGR-BasicClass/Main.java . . . 67
38 BHE-StatEx/StatExample.java . . . 68
39 AFL-FunRecur/SimpleRec.java . . . 69
40 BHG-StatOrd/Stats.java . . . 72
41 BHG-StatOrd/Main.java . . . 73
42 BHF-StatBlocks/StatBlocks.java . . . 73
43 BGX-Singlet/Connect.java . . . 75
44 BGX-Singlet/Config.java . . . 76
45 BGX-Singlet/Main.java . . . 76
46 BGU-SingList/Node.java . . . 78
47 BGU-SingList/MyList.java . . . 78

48 BGU-SingList/Main.java 79
49 BGW-Stack/Node.java 80
50 BGW-Stack/MyStack.java 80
51 BGW-Stack/Main.java 81
52 BGZ-ArrStack/ArrStack.java 81
53 BGZ-ArrStack/Main.java 82
54 BGY-SimpQueue/MyQueue.java 83
55 BGY-SimpQueue/Main.java 83
56 BHH-Strings/Strings.java 86
57 BHI-StrBuilder/StrBuilder.java 88
58 DJV-Inherit/Point.java 91
59 DJV-Inherit/Pixel.java 92
60 DJV-Inherit/Main.java 92
61 DJW-InherAnimal/Animal.java 94
62 DJW-InherAnimal/Dog.java 94
63 DJW-InherAnimal/Cat.java 95
64 DJW-InherAnimal/Main.java 95
65 AAR-EquToString/EquToString.java 96
66 HUM-HashEquals/Person.java 97
67 HUM-HashEquals/AHash.java 98
68 AXW-ScanExcept/ScanExcept.java 104
69 AXX-TryCatch/TryCatch.java 105
70 AYC-Excpts/Excpts.java 106
71 AYN-CheckedExc/MyUncheckedException.java 107
72 AYN-CheckedExc/CheckedExc.java 108
73 KEP-WriteBin/WriteBin.java 111
74 BHJ-BytesChars/BytesChars.java 112
75 KFE-GrepNew/GrepNew.java 114
76 KFD-Grep/Grep.java 116
77 KFI-File2Str/File2Str.java 117
78 HUS-Tokens/Tokenizer.java 118
79 GXP-Split/Splitting.java 124
80 GXX-RegGroups/RegGroups.java 128
81 GXR-RegExNames/Name.java 132
82 GXR-RegExNames/RegEx.java 133
83 GXU-Regexes/Regexes.java 134
84 GXW-REExplorer/REExplorer.java 135
85 GXY-RegFind/RegFind.java 138

Index

<< operator, 24
= operator, 18
>> operator, 24
>>> operator, 24
& operator, 24
&& operator, 21
| operator, 25
|| operator, 21
! operator, 21
^ operator, 22, 25

activation record, 51
algorithm, 2
AND, 7
AND operator, 21
antidiagonal, 47
argument, 18, 51
arithmetic operators, 19
array, 43
- column of, 47
- creating, 43
- index, 43
- jagged, 46
- length, 43
- multi-dimensional, 46
- rectangular, 46
- reference, 43
- row of, 47
- size, 43
- square, 47

assert, 105
assertion, 109
assertion statement, 105
AssertionError, 109
assignment operator, 18
associativity, 27
attribute, 59

backreference, 129
base class, 90
binary operator, 18
bit, 1
Bit-wise operators, 24
block, 29
Boolean type, 6
BufferedReader, 112
BufferedReader (class), 110
BufferedWriter (class), 110
byte, 1
byte code, 5
ByteArrayInputStream (class), 110
ByteArrayOutputStream (class), 110

capturing group, 127
CASE_INSENSITIVE, 131
casting, 14, 26, 94
catch, 99, 100
CharArrayReader (class), 110
CharArrayWriter (class), 110
checked exception, 99
class, 4, 10, 59
- AssertionError, 109
- base, 90
- BufferedReader, 110
- BufferedWriter, 110
- ByteArrayInputStream, 110
- ByteArrayOutputStream, 110
- CharArrayReader, 110
- CharArrayWriter, 110
- Class, 90
- derived, 90
- Error, 99
- Exception, 99
- File, 117
- FileInputStream, 110
- FileNotFoundException, 105
- FileOutputStream, 110
- FileReader, 117
- inner, 59, 62
- InputStream, 110
- InputStreamReader, 110, 112
- IOException, 100, 101, 103, 111
- Matcher, 126
- name of, 4
- nested, 83
- NumberFormatException, 105
- Object, 90
- ObjectInputStream, 110
- ObjectOutputStream, 110
- OutputStream, 110
- OutputStreamWriter, 110
- Pattern, 126
- PrintStream, 110
- PrintWriter, 110

Reader, 110
RuntimeException, 99
Scanner, 16, 103
StreamTokenizer, 118
String, 85
StringBufferInputStream, 110
StringBuilder, 88
StringReader, 110
StringWriter, 110
Throwable, 99
Writer, 110
Class (class), 90
class file, 5
Closeable (interface), 115
column of array, 47
comment, 4
compiler, 2
compound assignment operator, 18
compound instruction, 29
concurrency, 3
conditional operator, 23
conditional statement, 29
constructor, 59, 63
default, 60, 64
delegating, 64
overloading, 64
conversion, 14, 26

decorator, 114
default constructor, 60
derived class, 90
diagonal, 47
do-while loop, 38
DOTALL, 131

eager evaluation, 76
encapsulation, 62
equals (method), 96
Error (class), 99
Euclid, 2
evaluation
lazy, 75, 76
exception, 99
checked, 99
FileNotFoundException, 105
IOException, 100, 101, 103
NumberFormatException, 105
propagating, 102
rethrowing, 101
throw, 99
throwing, 102
unchecked, 99
Exception (class), 99
executable, 1
expression, 29, 40

field, 59
static, 59
FIFO, 82
File (class), 117
FileInputStream (class), 110
FileNotFoundException, 105
FileOutputStream (class), 110
FileReader (class), 117
final method, 90
finally, 101
first in, first out, 82
floating point type, 6
for loop, 40
for-each loop, 43
free memory, 59
function
argument, 51
main, 4
overloading, 57
parameter, 51
recursive, 55
static, 51, 59

garbage collector, 10
Gosling J., 2
graphical user interface, 3
greedy quantifier, 123
group, 127
GUI, 3

hashCode (method), 96
heap, 15, 59
hermetization, 62
hexadecimal notation, 1

implicit conversion, 26
index, 43
inheritance, 90
initialization blocks, 72
inner class, 59
InputStream (class), 110
InputStreamReader (class), 110, 112
instance, 59
instanceof, 90
instruction, 1, 29
compound, 29
instruction set, 1

integral type, 6
interface
  Closeable, 115
interpreter, 2
IOException, 100, 101, 103
IOException (class), 111

jagged array, 46
Java Virtual Machine, 3, 5
JIT, 5
just-in-time compilation, 5
JVM, 3, 5

labeled loop, 37
last in, first out, 80
late binding, 90
lazy evaluation, 75
left shift, 8
LIFO, 80
list, 77
local variable, 51
logical type, 6
loop
  do-while, 38
  for, 40
  for-each, 43
  labeled, 37
  named, 37
  while, 35

machine code, 1
main, 4
main diagonal, 47
Matcher (class), 126
member, 59
  non-static, 59
method, 59
  equals, 96
  final, 90
  hashCode, 96
  overriding, 96
  static, 51
  virtual, 90
modulus operator, 20
multi-dimensional array, 46
MULTILINE, 130
multithreading, 3

named loop, 37
nested class, 83
network programming, 3
non-static member, 59
NOT, 8
NOT operator, 21
NumberFormatException, 105

object, 59
Object (class), 90
object type, 10
ObjectInputStream (class), 110
ObjectOutputStream (class), 110
operand, 18
operating system, 1
operator, 18
  <<, 24
  =, 18
  >>, 24
  >>>, 24
  &, 24
  &&, 21
  |, 25
  ||, 21
  !, 21
  ^, 22, 25
  AND, 21
  arithmetic, 19
  assignment, 18
  associativity, 27
  binary, 18
  bit-wise, 24
  compound assignment, 18
  conditional, 23
  instanceof, 90
  modulus, 20
  NOT, 21
  OR, 21
  precedence, 26
  remainder, 20
  shift, 24
  short-circuited, 21
  ternary, 18
  unary, 18
  XOR, 22
option
  CASE_INSENSITIVE, 131
  DOTALL, 131
  MULTILINE, 130
  UNICODE_CASE, 131
  UNICODE_CHARACTER_CLASS, 132
option flags, 130
OR, 7

OR operator, 21
OutputStream (class), 110
OutputStreamWriter (class), 110
overloading, 57, 64
overriding, 90
overriding a method, 96

package private, 62
parameter, 51
Pattern (class), 126
pointer, 6, 43, 52
polymorphism, 90
possessive quantifier, 123
precedence of operators, 26
primitive type, 6
PrintStream (class), 110
PrintWriter (class), 110
private, 62
processor, 1
program, 1
propagating exception, 102
protected, 62
public, 62

quantifier, 123
- greedy, 123
- possessive, 123
- reluctant, 123

queue, 82

random numbers, 39
Reader (class), 110
rectangular array, 46
recursion, 55
reference, 43, 59, 61
reference type, 6
regex, 86, 122
register, 1
regular expression, 122
reluctant quantifier, 123
remainder operator, 20
rethrowing exception, 101
return type, 51
right shift, 8
row of array, 47
RuntimeException (class), 99

Scanner, 103
Scanner (class), 16
scientific notation, 11
selection sort, 56
shift
- left, 8
- right, 8

shift operator, 24
short-circuited operator, 21
singleton, 75
singly linked list, 77
square array, 47
stack, 15, 51, 80
stack frame, 51
state, 59
statement, 29
- conditional, 29
- switch, 30

static, 59, 68
static field, 59
static function, 51
static method, 51
stream (IO), 110
StreamTokenizer (class), 118
String, 85
String (class), 85
StringBufferInputStream (class), 110
StringBuilder (class), 88
StringReader (class), 110
StringWriter (class), 110
subnormal number, 9
superclass, 90
switch expression, 34
switch statement, 30
System.err, 110
System.in, 110
System.out, 110

ternary operator, 18
this, 62
throw exception, 99
Throwable (class), 99
throwing exception, 102
toString, 66
try, 99, 100
try-with-resources, 111, 115
type, 6
- boolean, 6
- casting, 14
- conversion, 14
- floating point, 6
- integral, 6
- logical, 6
- object, 10
- primitive, 6
- reference, 6

unary operator, 18
unchecked exception, 99
UNICODE_CASE, 131
UNICODE_CHARACTER_CLASS, 132

variable, 6, 10
    local, 51
virtual method, 90
void, 51

while loop, 35
Writer (class), 110

XOR, 7
XOR operator, 22