Java - lecture notes

with example programs from archive ***java_examples***

Tomasz R. Werner

Warsaw, December 7, 2018 12:57

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

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

Contents

Section 1

General introduction

1.1 Computers and programming languages

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

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

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

1.2 What is Java?

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

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

Some features of Java:

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

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

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

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

Section 2

Compiling and running a Java program

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

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

Listing 1 AAC-HelloWorld/Hello.java

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

Some elements to note:

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

Having a source file(s), we have to compile it. The Java compiler is a program named ***javac*** (***javac.exe*** on Windows). We invoke it passing, as arguments, one or more ***.java***

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

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

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

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

Section 3

Types, variables, literals

3.1 Primitive types

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

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

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

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

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

The value it corresponds to is

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

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

01111111

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

10000000

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

3.2 Object types

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

3.3 Variables and literals

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

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

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

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

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

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

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

0111 1111 1111 1111

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

1000 0000 0000 0000

which is 0b1000000000000000 or 0x8000.

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

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

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

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

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

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

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

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

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

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

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

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

Listing 2 AAG-Literals/Literals.java

```
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");
    }
}
```

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

Listing 3 AAL-Variables/Variables.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);
```

```
15          String john,              // does string john exist?
16                 mary = "Mary";
17          john = "John";
18          System.out.println(john + " and " + mary);
19          john = mary;
20            // reference copied, string "John" lost!
21          System.out.println(john + " and " + mary);
22      }
23  }
```

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

3.4 Conversions

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

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

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

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

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

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

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

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

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

Section 4

Quick introduction to I/O

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

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

Listing 4 AAI-ReadKbd/ReadKbd.java

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

We used here **nextInt** (to read an **int**), **nextDouble** (to read a **double**), and **next**

(to read a **String**): there are analogous functions **nextByte**, **nextShort**, **nextLong**, **nextBigInteger**, **nextFloat**, **nextBigDecimal**, and **nextBoolean**. Note: when all data has been read, the scanner should be closed (by invoking `scan.close()`, as shown in the example).
In order to read data from a graphical widget, or to display a message (string), one can use static functions from class **JOptionPane**, as shown below (the first argument of these functions is **null** for reasons we will learn about later). Note that **showInputDialog** returns a string (strictly speaking the reference to a string); if we know that this string represents a number and we want this number as an **int** or a **double**, we have to parse this string to get numbers using **Integer.parseInt**, or **Double.parseDouble**):

Listing 5 AAJ-ReadGraph/ReadGraph.java

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

Section 5

Instructions and operators

5.1 Operators

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

5.1.1 Assignment operator

Assignment operator

```
b = expr;
```

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

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

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

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

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

the value of a is still 5; when calculating a+1 we got a value (in this case 6) which is subsequently multiplied by 2 and added the current value of b, i.e., 1. We haven't assigned any new value to a, so it remains as it was. However, the value of the whole expression on the right-hand side of the assignment (13) has been assigned to b (erasing its previous value), so b *is* modified here.
There is a special form of assignment, the so called **compound assignment operator**. It has the form

```
a @= b
```

where @ stands for any binary operator, like +, *, %, >>, etc. and is (almost) equivalent to a = a @ b. For example a += 5 would mean *increment* a *by 5*, and a /= 2 — *divide* a *by 2*.
Incrementing and decrementing integral values by 1 is so often used that it has a special syntax. If a is a variable of an integral type, then ++a and a++ cause a to be incremented by 1 (with - instead of + — decremented). However, there is a difference between these two forms. *Pre*incrementation or decrementation (++a or --a) are 'immediate', while *post*incrementation or decrementation takes place *after* the whole expression has been evaluated. For example, after

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

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

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

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

5.1.2 Arithmetic operators

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

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

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

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

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

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

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

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

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

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

prints

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

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

5.1.3 Conditional and relational operators

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

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

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

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

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

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

```
expr1 && expr2
```

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

```
expr1 || expr2
```

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

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

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

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

could result in divide-by-zero error!

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

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

```
expr1 ^ expr2
```

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

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

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

Listing 6 — ABY-RelOps/RelOps.java

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

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

```
cond ? expr1 : expr2
```

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

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

5.1.4 Bit-wise operators

We can operate on variables of integral types (mainly **int**) treating them as "buckets" of single bits. In what follows, remember that operations of shifting, ANDing, ORing

etc., that we will discuss, do not modify their arguments: they return *new* values that we have to handle in some way (display it, assign to a variables, and so on).

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

Shifting

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

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

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

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

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

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

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

ANDing, ORing, etc.

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

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

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

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

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

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

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

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

Listing 8 BAA-Bits/Bits.java

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

5.1.5 Precedence and associativity

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

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

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

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

rather than

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

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

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

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

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

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

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

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

Table 1: Precedence of operators

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

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

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

is correct, because it is equivalent to

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

while

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

would not compile, because we assign uninitialized value of `a` to `c`.

The only ternary operator, the conditional operator, is also right-associative; this instruction

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

is correct, because it's equivalent to

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

while

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

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

Let us look at another example:

Listing 9 AAP-BasicOps/BasicOps.java

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

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

5.2 *Instructions*

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

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

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

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

5.2.1 Conditional statements

Conditional statements may look like this

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

// or

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

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

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

Listing 10 ABX-Ifs/LeapYear.java

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

5.2.2 Switch statement

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

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

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

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

Listing 11 ACC-SimpleSwitch/SimpleSwitch.java

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

```
                System.out.println("Invalid input");
        }
    }
}
```

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

Listing 12 ACD-Switch/Switch.java

```
import java.util.Scanner;

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

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

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

        int num;

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

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

5.2.3 Loops

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

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

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

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

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

Listing 13 AFA-While/Prime.java

```
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 14 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();
    }
}
```

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

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

Therefore,

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

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

Listing 15 AFE-DoWhileDice/Dice.java

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

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

A possible outcome of the program might be

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

***for* loop**

For loops looks like this:

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

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

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

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

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

Listing 16 AFH-ForPyram/Stars.java

```
import java.util.Scanner;

public class Stars {
```

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

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

For example, after inputing 9, the program prints

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

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

Listing 17 AFJ-ForWhileEuler/ForWhileEuler.java

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

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

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

            if (n == 0) break;

            int count = 0;
            for (int p = 1; p <= n; ++p) {
```

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

Section 6

Static functions

Classes usually contain, besides fields, constructors and methods (that we will cover later, when talking about classes) also **static functions** (or static methods). We can think about a function as a piece of code that can be executed (invoked) many times from other functions (for example, from **main**) in such a way that in each invocation some data that the function operates on may be different.
The definition of a static function is of the form

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

and consists of

- The keyword **static** (there are also functions which are not static). Before or after this keyword we could place an access specifier of the function (**public** or **private**) — we will discuss it later.
- The name of the type of a result which this function yields (the so called *return type*); **void** if the function doesn't return any result.
- A name of the function (it should always start with a lower-case letter but is otherwise arbitrary).
- In round parentheses, a list of parameters: these are comma separated pairs Type parName where Type is the name of a type and parName is an (arbitrary) name of this parameter (it should start with a lower-case letter). The list of parameters can be empty, but parentheses are always required.
- The body of the function enclosed in braces.

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

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

What happens is:

- *Copies* of the values of arguments are pushed (placed) on the program's stack (a special region of memory).
- These copies of the arguments, laying on the stack, can be accessed inside the body of the functions by names of the corresponding declared parameters. Their names in the calling function are completely irrelevant, because a function can see only *values* laying on the stack. We can say that it doesn't even know 'who calls' it and where from. It only knows that there must be values lying on the stack that can be associated with its parameters. In particular, we can use literals as arguments.
- All variables defined *inside* a function are *local* to this function — they are also located on the stack (in our example, variable res in **isPrime** is such a local variable). Names of local variables are arbitrary and unrelated to local variables of the same name in other functions.

- Instructions in the body of the function are executed. If the function is **void**, execution stops when the end of the definition is reached or a `return;` statement is encountered. If it is non-**void**, execution ends when statement `return expr;` is encountered, where expr is an expression whose value is of type declared as the return type of the function (or is convertible to this type).
- When the function returns, the stack is "rewound": it is reverted to the state it had before invocation. In particular, all local variables, including those corresponding to parameters, cease to exist.
- If the function returns a value, the invocation expression (something like `fun(a)`) may be considered to be a *temporary*, unmodifiable variable whose type is the return type of the function and value is that of expr appearing in the `return expr;` expression. It is a temporary variable, so normally we have to do something with it: print it, assign its value to a variable, or use it in another expression.

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.

Let us consider an example of two static functions: **isPrime** and **primesBetween**. Of course, as always in Java, they must be placed in a class. The first of these two functions checks if a given number is prime, the second prints prime numbers in a given interval.

Listing 18 BHK-StatFun/StatFun.java

```
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 19 BHL-RecFun/RecFun.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 situation is different when we pass to a function variables of object types (names of such types begin with a capital letter, like **String**). What is then really paseed (put on the stack) are rather (copies of) *addresses* of these variables, and not copies the variables themselves. We will discuss this problem later.

Section 7

Arrays

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

7.1 Creating arrays

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

- When an array is created, its size must be specified and then cannot be modified.
- The type *reference to array of elements of type* Type is denoted by Type[]. Statement `int[] arr;` means that arr is a reference to array of ints — 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 consecutive elements of the array.

For example, this is how we can create an 10-element array of chars 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();
```

Another example which 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];
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);
```

Let us emphasize again that variables declared as arrays are really pointers (in Java called *references*) to anonymous objects representing arrays. Their values are *addresses* of objects representing arrays, not the arrays themselves. 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 itself; consequently, having this address, functions which receive it *can* modify the original array (as they 'know where it is'). Examples can be found in the following program:

Listing 20 CYD-BasicArray/BasicArr.java

```
1 import static java.lang.System.out; // for convenience
2
3 public class BasicArr {
4
```

```
5       public static void main(String[] args) {
6           int[] a1 = {1,2,3};
7           printArr(a1, "Array a1");
8
9           int[] a2;  //  <-- no array here, only reference!
10          a2 = new int[]{4,5,6};
11          printArr(a2, "Array a2");
12
13          a1 = a2;   // <-- whatever was in a1 is lost!
14          printArr(a1, "After a1=a2, a1 is");
15
16            // a1 and a2 now refer to the same array!
17          a1[0] = 44;
18          a2[2] = 66;
19          printArr(a1, "After modifications a1 is");
20          printArr(a2, "After modifications a2 is");
21
22            // ad hoc array
23          printArr(new int[]{7,8,9}, "Ad hoc array");
24
25            // array returned from a function
26          a2 = getArr(5);
27          printArr(a2, "Array returned from function");
28
29            // passing  r e f e r e n c e  to function
30          reverseArr(a2);
31          printArr(a2, "Array a2 reversed");
32      }
33
34      /**
35       *  returns first n triangular numbers
36       */
37      private static int[] getArr(int n) {
38          int[] arr = new int[n];
39          for (int i = 0; i < n; ++i)
40              arr[i] = (i+1)*(i+2)/2;
41          return arr;
42      }
43
44      /**
45       *  prints an array of integer numbers
46       */
47      private static void printArr(int[] a, String message) {
48          out.print('\n' + message + ": [");
49          for (int i : a)
50              out.print(" " + i); // <-- 'i' cannot be changed
51          out.println(" ]; size = " + a.length);
52      }
53      /**
54       *  modifies input array (reversing order of elements)
```

```
55        */
56      private static void reverseArr(int[] a) {
57          for (int i = 0, j = a.length-1; i < j; ++i,--j) {
58              int p = a[i];
59              a[i]  = a[j];
60              a[j]  = p;
61          }
62      }
63  }
```

7.2 Arrays of references to objects

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

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

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

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

7.3 Multi-dimensional arrays

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

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

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

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 21 CYB-SimpleArrays/SimpleArrays.java

```
1  public class SimpleArrays {
2      public static void main(String[] args) {
3
4          // ================================================
5          int[] a = {1,2,3};
6          System.out.println("a.length = " + a.length);
7          for (int i = 0; i < a.length; ++i)
8              a[i] = (i+1)*(i+1);
9          for (int i = 0; i < a.length; ++i)
10             System.out.print(a[i]+" ");
11         System.out.println('\n');
12
13         // ================================================
14         int[][] b = { {1,2,3}, {4,5,6,7,8}, {11,12} };
15         System.out.println("b.length = " + b.length);
16         for (int row = 0; row < b.length; ++row)
17             System.out.println("b["+row+"].length = " +
18                                        b[row].length);
```

```
19          System.out.println();
20
21          for (int row = 0; row < b.length; ++row) {
22              for (int col = 0; col < b[row].length; ++col)
23                  System.out.print(b[row][col]+" ");
24              System.out.println();
25          }
26          System.out.println('\n');
27
28          // ================================================
29          int[] c = new int[]{1,2,3}; // <- size inferred
30          System.out.println("c.length = " + c.length);
31          for (int i = 0; i < c.length; ++i)
32              System.out.print(c[i]+" ");
33          System.out.println('\n');
34
35          // ================================================
36          int[] d = new int[5];       // <- elements are 0
37          System.out.println("d.length = " + d.length);
38          for (int i = 0; i < d.length; ++i)
39              System.out.print(d[i]+" ");
40          System.out.println('\n');
41
42          // ================================================
43          int[][] e = new int[3][2];  // <- a 3x2 matrix
44          System.out.println("e.length = " + e.length);
45          for (int row = 0; row < e.length; ++row)
46              for (int col = 0; col < e[row].length; ++col)
47                  e[row][col] = row+col;
48
49          // ================================================
50          int[][] f = new int[3][];
51          for (int row = 0; row < f.length; ++row)
52              System.out.print(f[row]+" ");
53          System.out.println();
54
55          for (int row = 0; row < f.length; ++row)
56              f[row] = new int[row*row+2];
57
58          for (int row = 0; row < f.length; ++row) {
59              System.out.println("f["+row+"].length = " +
60                                              f[row].length);
61              for (int col = 0; col < f[row].length; ++col)
62                  f[row][col] = row+col;
63          }
64          System.out.println();
65
66          for (int row = 0; row < f.length; ++row) {
67              for (int col = 0; col < f[row].length; ++col)
68                  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 (array of) grades for this student and this course:

Listing 22 CYJ-Arr3D/Arr3D.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) {
```

```
32              double ave = 0;
33              int    num = 0;
34                // over subjects for student s
35              for (int c = 0; c < grades[s].length; ++c) {
36                  num += grades[s][c].length;
37                    // 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  }
```

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. Any class can contain

- fields (non-static);
- static fields;
- constructors;
- methods (non-static);
- static functions (methods);
- static and non-static initializer blocks.

Objects (instances) of classes are always created on the heap — they never have names! There is no way to create an object locally on the stack — only references can be local. 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:

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

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.

Listing 23 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 24 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** seemingly has 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 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 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 (put on the 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. In such situation, the compiler will

automatically add this in front of x 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 of the same class and also 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;
- 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 25 BGI-VerySimple/VerySimple.java

```
1 public class VerySimple {
2     private int    age;
3     private String name;
```

```
4
5       // constructor
6     public VerySimple(int age, String n) {
7         this.age = age;
8         name = n;
9     }
10      //getters
11    public int getAge()  {
12        return age;
13    }
14    void setAge(int a) { // package private
15        age = a;
16    }
17      // getter (with no corresponding setter)
18    public String getName() {
19        return name;
20    }
21 }
```

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 26 BGI-VerySimple/Main.java

```
1  public class Main {
2      public static void main(String[] args) {
3          VerySimple alice = new VerySimple(23,"Alice");
4          VerySimple bob   = new VerySimple(21,"Bob");
5          alice.setAge(18);
6
7          System.out.println(
8                  alice.getName()  + " " + alice.getAge());
9          System.out.println(
10                 bob.getName()    + " " + bob.getAge());
11     }
12 }
```

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 be alway invoked on an object (in the example, it was the object pointed to by reference alice). 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 27 BGP-Point/Point.java

```
1  public class Point {
2      private int x, y;
3
4      public Point(int x, int yy) {
5          System.out.println("Point(int,int) with " +
6                             x + " and " + y);
7          this.x =  x;
8          y      = yy;
9      }
10
11     public Point(int x) {
12         this(x,0);
13         System.out.println("Point(int) with " + x);
14     }
15
16     public Point() {
```

```
17          this(0);
18          System.out.println("Point()");
19      }
20
21      public Point translate(int dx,int dy) {
22          x += dx;
23          y += dy;
24          return this;
25      }
26
27      public Point scale(int sx,int sy) {
28          x *= sx;
29          y *= sy;
30          return this;
31      }
32
33      public int getX() { return x; }
34      public int getY() { return y; }
35
36      /**/
37      @Override
38      public String toString() {
39          return "[" + x + "," + y + "]";
40      }
41      /**/
42
43      public static void main(String[] args) {
44          System.out.println("\n*** Creating point p1 (1,2)");
45          Point p1 = new Point(1,2);
46
47          System.out.println("\n*** Creating point p2 (1)");
48          Point p2 = new Point(1);
49
50          System.out.println("\n*** Creating point p3 ()");
51          Point p3 = new Point();
52
53          p3.translate(4,4).scale(2,3).translate(-1,-5);
54
55          System.out.println("\np1: [" + p1.getX() + "," +
56                                        p1.getY() + "]");
57
58          System.out.println("\np1: " + p1 + "  p2: " +
59                                  p2 + "  p3: " + p3);
60      }
61  }
```

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 illustretes 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 inherit it from class **Object**. The version from class **Object** doesn't return a very useful string, but, what is important, it exists and returns some string.
Another example

Listing 28 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 + ")";
    }
    /**/

```

```
25      public boolean isOlderThan(Person other) {
26          return birth_year < other.birth_year;
27      }
28  }
```

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

Listing 29 BGR-BasicClass/Main.java

```
1   import javax.swing.JOptionPane;
2
3   public class Main {
4       public static void main(String[] args) {
5
6           Person john = new Person("John", 1980);
7           Person mary = new Person("Mary", 1985);
8
9           System.out.println(
10              "Two Persons created: " +john + " and " + mary);
11
12          Person older = mary.isOlderThan(john) ? mary : john;
13          System.out.println("Older: " + older.getName() +
14                  " born in " + older.getYear());
15
16          String s = older + " is older";
17          JOptionPane.showMessageDialog(null,s);
18          System.exit(0);
19      }
20  }
```

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 has 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

```
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 30 BHE-StatEx/StatExample.java

```
1  public class StatExample {
2      private static double rate =  1;
3      private static char     ID = 'A';
4
5      private double amount;
6      private char       id;
7
8      public static   void setRate(double r) { rate = r; }
9      public static double getRate() { return rate; }
10
11     StatExample(double a) {
12         id = ID++;
13         amount = a;
14     }
15
16     @Override
17     public String toString() {
18         return "I'm " + id + ", I have $" + amount +
19             " = " + rate*amount + " PLN";
20     }
21
22     public static void main (String[] args) {
23        StatExample.setRate(4.1);
24        StatExample sa = new StatExample(10);
25        StatExample sb = new StatExample(16);
26        StatExample sc = new StatExample(20);
27        System.out.println(sc + "\n" + sb + "\n" + sa);
28     }
29 }
```

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 31 AFL-FunRecur/SimpleRec.java

```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] + " ");

        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));
    }
```

```
38
39        // should be called with from=0
40      static void selSortRec(int[] arr, int from) {
41          if (from == arr.length-1) return;
42          int indmin = from;
43          for (int i = from+1; i < arr.length; ++i)
44              if (arr[i] < arr[indmin]) indmin = i;
45          int temp = arr[from];
46          arr[from] = arr[indmin];
47          arr[indmin] = temp;
48          selSortRec(arr,from+1);
49      }
50
51      static int gcd(int a, int b) {
52          return  b == 0 ? a : gcd(b, a%b);
53      }
54
55      static int fact(int n) {
56          return  n <= 1 ? 1 : n*fact(n-1);
57      }
58
59        // must be called with k=2
60      static boolean isPrime(int n, int k) {
61          if (k*k > n)  return true;
62          if (n%k == 0) return false;
63          return isPrime(n,k+1);
64      }
65
66      static long counter = 0;
67        // stupid way of calculating Fibonacci numbers
68      static long fibo(int n) {
69          ++counter;
70          return n <= 1 ? (long)n : fibo(n-2) + fibo(n-1);
71      }
72
73      public static void main(String[] args) {
74            // Arrays
75          int[] a = {13,3,55,7,9,11};
76          printArrRec(a,0);
77          revArrayRec(a,0,a.length);
78          printArrRec(a,0);
79          selSortRec(a,0);
80          printArrRec(a,0);
81          printArrRecReverse(a,0);
82          System.out.println("Max. in a: "+maxElemRec(a,0));
83
84            // GCD
85          System.out.println("Greatest common divisor of" +
86                  " 5593 and 11067 is " + gcd(5593,11067));
87
```

```
88            // Factorials
89          System.out.println("10! = " + fact(10));
90          System.out.println("12! = " + fact(12));
91            // NO ERROR BUT WRONG!!!
92          System.out.println("13! = " + fact(13) + " WRONG!");
93
94            // Primes
95          System.out.println("Primes up to 100");
96          for (int n = 2; n <=100; ++n)
97              if (isPrime(n,2)) System.out.print(n+" ");
98          System.out.println();
99
100           // Fibonacci numbers
101         for (int n = 40; n <= 46; n += 2) {
102             counter = 0;
103             long r = fibo(n);
104             System.out.println("Fibo(" + n + ") = " + r +
105                                "; counter = " + counter);
106         }
107     }
108 }
```

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 ca be seen from the output of the following program:

Listing 32 BHG-StatOrd/Stats.java

```
1  public class Stats {
2      private static int sino;
3      private static int siyes = 2;
4      private static final int fin;
5
6      {          // 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 33 BHG-StatOrd/Main.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 34 BHF-StatBlocks/StatBlocks.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;

```

```
10 import static java.nio.charset.StandardCharsets.UTF_8;
11
12 public class StatBlocks {
13     static private double rateUSD = 1;
14     static private double rateUAH = 1;
15
16       // static initializer block
17     static {
18         try {
19             URL nbp = new URL(
20                 "http://www.nbp.pl/kursy/xml/LastA.xml");
21             BufferedReader bw =
22                 new BufferedReader(
23                     new InputStreamReader(
24                         nbp.openStream(),UTF_8));
25             String txt =
26                 bw.lines().collect(Collectors.joining(" "));
27             bw.close();
28             Matcher m =
29                 Pattern.compile(".*USD.*?(\\d+,\\d+)" +
30                                 ".*UAH.*?(\\d+,\\d+).*")
31                        .matcher(txt);
32             m.matches();
33             rateUSD = Double.parseDouble(
34                          m.group(1).replace(",","."));
35             rateUAH = Double.parseDouble(
36                          m.group(2).replace(",","."));
37         } catch (Exception e) {
38             if (JOptionPane.showConfirmDialog(
39                     null, "Fetching data failed; continue" +
40                     " anyway with default rates = 1?",
41                     "WARNING! FETCHING DATA FAILED!",
42                     JOptionPane.YES_NO_OPTION
43                     ) != JOptionPane.YES_OPTION
44                ) System.exit(1);
45         }
46     }
47
48     private static int ID = 1;
49
50     private final int id;
51
52       // non-static initializer block
53     {
54         if (ID % 100 == 13) ++ID;
55         id = ID++;
56     }
57
58     private double amount;
59
```

```
60      public StatBlocks(double a) { amount = a; }
61      public StatBlocks()         { this(20);   }
62
63      public static double getRateUSD() { return rateUSD; }
64      public static double getRateUAH() { return rateUAH; }
65
66      @Override
67      public String toString() {
68          return String.format(Locale.US,
69                  "My id = %d. I have %5.2f " +
70                  "PLN = $%5.2f = %6.2f UAH",id,
71                  amount, amount/rateUSD,amount/rateUAH);
72      }
73
74      public static void main (String[] args)
75                              throws Exception {
76          double rUSD = StatBlocks.getRateUSD();
77          System.out.println("Current rates are:\n" +
78              "    USD/PLN = " + rateUSD + '\n' +
79              "    UAH/PLN = " + rateUAH);
80          StatBlocks sa = new StatBlocks();
81          StatBlocks sb = new StatBlocks(40);
82          StatBlocks sc = new StatBlocks(80);
83          System.out.println(sc + "\n" + sb + "\n" + sa);
84      }
85  }
```

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):

```
    USD/PLN = 4.0663
    UAH/PLN = 0.1506
My id = 3. I have 80.00 PLN = $19.67 = 531.21 UAH
My id = 2. I have 40.00 PLN = $ 9.84 = 265.60 UAH
My id = 1. I have 20.00 PLN = $ 4.92 = 132.80 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 35 BGX-Singlet/Connect.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 36 BGX-Singlet/Config.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 37 BGX-Singlet/Main.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");
```

```
9         }
10    }
```

Section 9

Strings and *StringBuilders*

9.1 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 **String**s 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);
- `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)` → '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")` → 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")` → 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 example below:

Listing 38 BHH-Strings/Strings.java

```
1  public class Strings {
2      private static void pr(String m, String a, String b) {
3          System.out.println(m + ": \"" + a +
4                             "\" -> \"" + b + "\"");
5      }
6      public static void main (String[] args) {
7            // length()
8          String a = " Shakespeare   ";
9          String b = a.trim();
10         System.out.println("a.length() -> " + a.length());
11         System.out.println("b.length() -> " + b.length());
12         pr("        trim()", a, b);
13
14           // substring
15         a = "abcdefgh";
16         b = a.substring(3,6);
```

```
17          pr("substring(3,6)", a, b);
18          b = a.substring(3);
19          pr("  substring(3)", a, b);
20
21            // toUpperCase, toLowerCase
22          b = a.toUpperCase();
23          pr(" toUpperCase()", a, b);
24
25            // split
26          String[] arr = "ONE:TWO:THREE".split(":");
27          for (String d : arr)
28              System.out.print(d.toLowerCase() + " ");
29          System.out.println();
30
31          arr = "one   two       three".split("\\s+");
32          for (String d : arr)
33              System.out.print(d.toUpperCase() + " ");
34          System.out.println();
35
36            // charAt
37          String ny = "New York";
38          for (int i = 0; i < ny.length(); ++i)
39              System.out.print(ny.charAt(i) + " ");
40          System.out.println();
41
42            // toCharArray
43          char[] ca = ny.toCharArray();
44          for (int i = ca.length-1; i >= 0; --i)
45              System.out.print(ca[i]);
46          System.out.println();
47
48      }
49  }
```

9.2 Class StringBuilder

The fact that **String**s 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 **StringBilder** 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 39 BHI-StrBuilder/StrBuilder.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: %d = %.3f sec%n",
                elapsedS, (double)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: %d = %.3f sec%n",
                elapsedB, (double)elapsedB*1e-9);

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

The program prints

```
sb = one,two,three,four
String: 10853917886 = 10,854 sec
Builder: 4034038 = 0,004 sec
elapsedS/elapsedB = 2690,58
```

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

Section 10

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 40 DJV-Inherit/Point.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 41 DJV-Inherit/Pixel.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 42 DJV-Inherit/Main.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 43 DJW-InherAnimal/Animal.java

```
1  public class Animal {
2      protected String name;
3      protected double weight;
4        // no default constructor!
5      public Animal(String n, double w) {
6          name = n;
7          weight = w;
8      }
9      public String getVoice() {
10         return "?";
11     }
12     public static void voices(Animal[] animals) {
13         for (Animal a : animals)
14             System.out.println(a + " " + a.getVoice());
15     }
16     @Override
17     public String toString() {
18         return name + "(" + weight + ")";
19     }
20 }
```

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 44 DJW-InherAnimal/Dog.java

```
1  public class Dog extends Animal {
2      public Dog(String name, double weight) {
3          super(name, weight);
4      }
5      @Override
6      public String getVoice() {
7          return "Bow-Wow";
8      }
9      @Override
```

```
10      public String toString() {
11          return "Dog " + super.toString();
12      }
13  }
```

and class **Cat**

Listing 45 DJW-InherAnimal/Cat.java

```
1   public class Cat extends Animal {
2       public Cat(String name, double weight) {
3           super(name, weight);
4       }
5       @Override
6       public String getVoice() {
7           return "Miaou-Miaou";
8       }
9       @Override
10      public String toString() {
11          return "Cat " + super.toString();
12      }
13  }
```

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 46 DJW-InherAnimal/Main.java

```
1   public class Main {
2       public static void main (String[] args) {
3           Animal[] animals = {
4                   new Dog("Max", 15), new Cat("Batty", 3.5),
5                   new Dog("Ajax", 5), new Cat("Minnie", 4)
6           };
7           Animal.voices(animals);
8       }
9   }
```

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.

Section 11

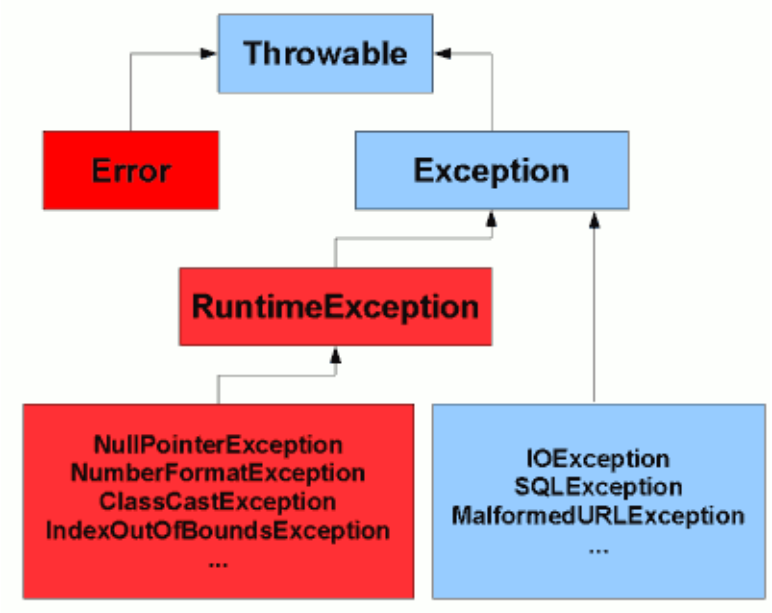
Exceptions

Exceptions are situations when program does not know what do 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 may, 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

11.1 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.

11.2 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.

11.3 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).

11.4 Throwing exceptions

In some situations, we can throw exceptions by ourself. 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 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 constructor, fields and methods if we find it appropriate for our purposes. This is seldom useful, because the main message passed by an exception is just its type.

11.5 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 47 AXW-ScanExcept/ScanExcept.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();
    }
}
```

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 48 AXX-TryCatch/TryCatch.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 49 AYC-Excpts/Excpts.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 50 AYN-CheckedExc/MyUncheckedException.java

```
public class MyUncheckedException
                    extends IllegalArgumentException {
```

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

and the we use it

Listing 51 AYN-CheckedExc/CheckedExc.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)
```

```
38                                  throws InterruptedException {
39          if (time < 0)
40              throw new MyUncheckedException("Negative time");
41
42          System.out.println(
43                  "Going to sleep for " + time +"ms");
44          Thread.sleep(time);
45          System.out.println("Waking up");
46      }
47  }
```

11.6 Assertions

Another way of dealing with possible errors in our programs is by using *assertions* (line 33 of the listing 47 on page 79). After the keyword **assert**, we specify a condition (something with **boolean** value) and, after a colon, a message. If the condition is **false**, the program will print the message and terminates (by throwing an exception). Note however, that assertions are active only when a program is run with **-ea** (*enable assertions*) option. Because assertions are sometimes *on* and sometimes *off*, the program should always behave in the same way, whether assertions are enabled or not (no side effects). Normally, assertions check conditions that we are almost sure should always hold. If, however, the condition is *not* met, an *unchecked* exception of type **AssertionError** will be thrown: we should never even try to handle it, because it indicates a serious flaw in the program that must be corrected.

Listings

List of listings

Index