Interreg
EUROPEAN UNION
Austria-Czech Republic
European Regional Development Fund

INFORMATICS

Introduction to Java programming

UNIVERSITY OF APPLIED SCIENCES UPPER AUSTRIA

VŠTE

EUROPEAN UNION

Contents

1. INTRODUCTION TO PROGRAMMING IN JAVA

Creating programs for Java Wageform is done in two phases. In the first step we write the so-called source code. The source code is a program notation. In the second step, the source code is translated into byte code. Translation of the source code into the byte code is provided by the compiler. Bytecode are Java Virtual Machine (JVM) instructions. Each instruction consists of an opcode and no or several operands. JVM allows you to run the byte code. It functions as an interpreter, ie it reads instructions and executes them immediately. Modern JVMs often include a Just-In-Time compiler (JIT compiler) that translates JVM instructions into the native processor code. Native code consists of instructions that are specific to the processor type. E.g. Intel and SPARC processors have different sets of instructions. Because a program in native code runs faster than byte code, translating it into native code can significantly speed up the program. However, the time needed for the translation of the byte code into the native code must be included in the total program runtime. This time is not negligible, therefore, only parts that are performed repeatedly are usually translated.

1.1. Classes

The Java programme consists of one or more classes. We declare the class using the keyword class. A class can contain methods. Each class and method has a name (except for anonymous classes, which will not be discussed here). Both the class and method contents are enclosed in braces {and}. We write commands into the methods. Each program must contain the main method, which always has the same header:

```
public static void main( String[] args )
```

The meaning of each word in this declaration will be discussed later. To begin with, we will write commands only to the main method. This method serves as an entry point to the application. To start the program, JVM calls this method. The commands are executed one by one in the order in which they are written. Use the System.out.println () method to exit the screen. A program that prints a simple greeting looks like this:

```
public class Prvni {
   public static void main( String[] args ) {
      System.out.println( ”ahoj” );
   }
}
```

This source declares the First class that contains the main method

1.2. Java virtual machine

Java Virtual Machine (JVM). JVM allows you to run Java programs. JVM's task is to provide programs with the same environment. The same environment ensures that the Java program runs on a SPARC-based computer running Solaris as well as an Intel-based computer running MS Windows. Starting the Java program is done in two steps: in the first step we start the JVM and in the second step the JVM loads the byte code into memory and runs it (it calls the main method). Program memory is divided into three areas: stack, heap, and method area. The method area contains a program (byte code). The program consists of a sequence of JVM instructions. Each instruction consists of a single-byte character and possibly operands. E.g. the goto instruction has an opcode 167 and performs an unconditional jump to the address specified by the operand.

The stack is used for allocating local variables, for storing parameters and results of JVM instructions, and for passing parameters and results when calling methods. Stack memory is allocated in stack frames. Each method call allocates a new frame. The frame contains memory for local variables and the so-called operand stack. The JVM maintains a pointer to the so-called stack top, which is the last value entered. The operand stack is used for JVM instruction parameters and results. Running program memory Ex. the iadd instruction fetches two values from the stack top, sums those values and places the result on the stack top.

Each operand stack item has a length of at least 4 bytes. Thus, all JVM arithmetic instructions work with operands of at least 32 bits in length. In the variable area, memory is also allocated after 4 bytes. That is, each variable occupies at least 32 bits in the JVM.A heap is an area of memory where objects can be created at runtime. The heap is created using the new keyword. Unlike C ++, for example, there is no need to delete objects. The so-called garbage

collector takes care of the removal of objects in Java. The garbage collector itself detects unused objects and deactivates those objects. After the object is deactivated, the memory occupied by the object is ready for further use.

The description of the programming language is divided into two parts: syntax and semantics. Syntax describes the rules for correct writing. E.g. says where you need to write parentheses. If the program is syntactically correct, it can be compiled and run. Semantics defines the meaning of language constructs. E.g. says the = operator performs the assignment. For a program to do what we want it to be, it must be semantically correct. In this text we will discuss both syntax and semantics. We will discuss most of the syntactic constructs of Java and we will say its semantics. When writing the program we use so-called keywords. These are words that have a special meaning in the language (two of them, const and goto, but remain unused). Java is case sensitive. E.g. Class is something other than class. Keywords are only lowercase.

1.3. BlueJ

BlueJ is a free, multi-platform development environment developed specifically for object-oriented programming in Java. It allows students to design a class diagram of a developed application in a simplified version of UML. The main advantage of BlueJ is its interactivity - it allows you to instantiate individual classes, send them messages and call their methods. (Wikipedia)

1.4. Installation

You must install Java first. Download the Java Development Kit (JDK) from oracle.com. BlueJ can be downloaded from BlueJ.org. Install both. Instructions for using BlueJ can be found, for example, at: https://www.bluej.org/tutorial/tutorial-english.pdf.

1.5. Creation of variables

Values are stored in variables. A variable is a named location in memory. In the framework of the creation of the variable - its declaration, in the Java programming language, it must first be stated with each variable what type of data to be included in Example 1 in our case int, boolean, char. Their meaning is explained in the next paragraph.

The name of the variable follows. The names of variables are written without diacritics and spaces. The spaces between the words are simply omitted and every other word starts with a capital letter. Close the term with a semicolon.

Example 1

```
int a;
boolean IsTrue;
char CarConsumption;
```

In Example 1, We created variables, but we did not assign any values. In Example 2, we created variables to which we assigned specific values. The first assignment to the variable is called initialization.

Example 2

```
int a=2;
boolean IsTrue =true;
char CarConsumption ='A';
```

If we want to declare more variables at a time, we must separate them with a comma:

```
int a, b;
```

We will distinguish between expression and command. The expression always has some value. This value is obtained by evaluating the expression. E.g. 42 and x + 1 are expressions. The command is the code that does something. E.g. X = 1; Is a command that assigns a value of 1 to the x variable. If we have an expression that evaluates something, we can usually command it by writing a semicolon behind it. In this case, we say that the evaluation of this expression has a side effect.

2. DATA TYPES

2.1. Integers

Int - We can only enter integers ranging from -2,147,483,648 to 2,147,483,647, including zero. An integer can represent, for example, the number of computer operations per second.

Other integer data types are listed in the table:

Type	**Bytes**	**Range**	
byte	1	-128	127
short	2	-32 768	32 767
int	4	-2147 483 648	2147 483 647
long	8	-9,22 x 1018	9,22 x 1018

If the result of the operation does not lie within the permissible interval of the given data type, an overflow occurs. In this case, the result is opposite to the result of the mathematical operation. It does not cause an error in Java, but it should be counted. E.g. If we have a value of 127 in a byte variable and add 1 to it, it will be in the -128 variable.

2.2. Real numbers

To work with real numbers, Java has two primitive data types: float and double. Both use the same representation method: the real number is stored as a triple sign, mantissa, and exponent. So we only store numbers approximately. Float type uses 32 bits: 1 bit occupies a sign, 8 bits of exponent, and 23 bits of mantissa. The double type uses 64 bits: 1 bit for the sign, 11 for the exponent, and 52 for the mantissa.

2.3. Conversion

Conversion of a value to another data type. E.g. Converting double to int. Some conversions are done automatically, eg int type to double type, others need to request, eg type long to int type. Do not need to write auto-conversion:

```
int a = 100;
 long a = b;  // Automatic conversion from int to long
```

If the conversion is required, we specify the target type in brackets before the converted value. E.g. Convert from double to int as follows:

```
double d1 = 5.85;
int i1 = (int) d1;
```

The result of this conversion will be the value that is in the entry of the original number before the decimal point (for nonnegative values it is the whole part of the number). I.e. In variable i2 the value will be 5.

```
double d2 = -4.99;
int i2 = (int) d2;
```

In variable i2, the value is -4. Recall that this is not the whole part of the number because the whole part of the number -4.99 is -5.

2.4. Logical

Boolean - Contains only true and false expressions. It is also called a logical variable. If we compare two numeric variables in a <b. Let us call them a and b. If and is actually smaller than b then the expression a <b is true and true is set to variable c. If true is not in c, false is stored

```
int a = 5;
int b = 6;
boolean c = a<b;
System.out.println(c);
```

In this example, the system writes "true".

The false is internally represented as 0, the value true as 1.

2.5. Character

Char - We can store one character in it. Before and after this character, there must be apostrophes (simple quotation marks). We can print characters from the Unicode table. The char value can be understood as the character index in the Unicode table.

```
char c ='H';
System.out.println(c);
```

2.6. Text strings

We use String to work with strings. Chain constants are written into quotation marks.

```
String s = ” Hi how are you?”;
```

Chains can be joined using the operator +.

```
String s1 = ”jdk”, s2 = ”7.0”;
String s3 = s1 + s2;  // Creates a string ”jdk7.0”
```

Another value can be added to the string. In this case, the value is first converted to a string and then the two strings are merged.

```
int x = 42;
String s = ”answer is ” + x;
```

The example creates the string "answer is 42"

2.7. Comments

There are three kinds of comments in Java:

- one line - starts with // and continues to the end of the line
- Multiline - starts with / * and ends the characters * /
- Documentation - starts with characters / ** and ends with characters * /

Documentation comments are intended for processing by Javadoc, which generates documentation in HTML format.

```
/**
    * Main program class.
    */

   public class Main {
      public static void main( String[] args ) {
         /* Prints the answer */
         System.out.println( 42 ); // answer is 42
      }
   }
```

By the comments, we add additional information to the source text. The comment transporter skips, so they have no effect on the program run. We should comment, for example, on the importance of variables, unusual procedures and non-traditional algorithms. Thus, sites whose meaning does not have to be obvious to a reader at first glance.

2.8. Terminal input and output

In this chapter, we'll show you how to write to the screen and read from the keyboard. The output of the screen is the so-called Output Current (System.out). We use three of his methods: print (), println (), and printf (). The call below lists: Hi Baby

```
System.out.println( ” Hi Baby!” );
```

The print () and println () methods print out the value we specify in brackets after the method name (this is the value we call the parameter). It differs by adding the transition to a new line to println (), so the next call to print () or println () will print from the beginning of the line. You can use strings to associate with the + operator when printing.

```
int v = 200;
System.out.println( ”Car moved ” + v + ” km/h” );
```

A more elegant output offers the printf () method (so-called formatted output), which is similar to the C language.

```
int m = 6;
System.out.printf( ” African elephant weighs %d tons”, m );
```

The printf () parameters are a formatting string and a list of values. The format string can contain so-called output conversions that determine the shape in which the appropriate value is printed. Each conversion starts with a% sign. E.g. % D is the decimal integer output. When executing the command, the appropriate value from the value list is placed instead of the conversion. If the value is missing or does not match the output conversion, an error occurs.

A special case is conversion% n, which does not match any value in the value list. This conversion is reflected by moving to a new line. In MS Windows, a pair of characters \ r (carriage return) and \ n (line feed) is used to move to the new line. Unix systems use \ n. Conversion% n ensures that the correct transition to the new line is used, ie \ r \ n on MS Windows and \ n on Unix.

For real numbers, we have% f conversion. We can specify the number of digits after the decimal point (the default value is 6). E.g. % .2f prints two digits after a decimal point.

```
System.out.printf( ” Euler’s constant is about %.2f%n”, Math.E );
```

Characters printed using the conversion %c.

```
char c = ’@’;
int i = c;
System.out.printf( ”Character ’%c’ Is in the Unicode table
in position %d%n”, c, i );
```

We use a conversion for strings %s.

```
String java = ”Java”;
System.out.printf( ” Our favorite programming language is %s%n”,
java );
```

For a read from the keyboard, we have the so-called "Input Stream" (System.in) in Java. Mostly we do not use it directly but through the Scanner class. This class offers methods for reading primitive types and strings. If we use the Scanner class, our program usually starts with the import command, which tells the translator where to look for the Scanner class. Before we first read, we create an instance of the Scanner class using the new keyword. To retrieve an int value, call the nextInt () method at this instance. import java.util.Scanner;

```
public class Read {
   public static void main( String[] args ) {
      Scanner sc = new Scanner( System.in );
      int x = sc.nextInt();
      System.out.printf( ” Readed value is: %d%n”, x );
   }
}
```

The nextDouble () method is used to retrieve the double value. Read the string by next ().

```
Scanner sc = new Scanner( System.in );
double d = sc.nextDouble();
System.out.printf( ”Readed number is: %f%n”, d );
String s = sc.next();
System.out.printf( ”Readed string is: %s%n”, s );
```

3. OPERATORS

Numbers can perform common arithmetic operations in Java: addition, subtraction, multiplication, division, modulo (the remainder after integer division). Operators are used to writing operations. E.g. The addition is written using the + and modulo operator with the% operator. Thus, the x + 2 entry is a write of the addition operation. Each operator works with one or more values or variables called operands. Depending on the number of operands we can divide the operators into unar (one operand), binary (with two operands) and ternary (with three operands). The federal operator is eg ++ (increment) and the binary operator is eg = (assignment). The only ternary operator is?: (Conditional operator). The result of the operator execution is the value (we say the operator returns the value). E.g. Binary operator + (addition) returns the sum of its operands. The type of result depends on the operator and sometimes on the operands. For operators returning an integer value, the result is either int or long. E.g. If we add two int values, the result will be int, if we add two long values, the result will be long, and if we add two-byte values, the result will be int. Operator / (partitioned) returns integer shares for integer operands. E.g. 15/4 is 3 and -15 / 4 is -3.

If the value of the second operand is 0, an error occurs. If at least one operand is real (ie, float or double), the second operand is converted to real, and the operator divided will return the ratio of both operands. E.g. 4.5 / 3 is 1.5. If the second operand is 0, the real divide will result in some of these values: plus infinity if the first operand is positive, minus infinity if the second operand is negative or NaN (Not a Number) if the first operand is positive Infinity, negative infinity, NaN or 0. Operator% (modulo) returns the remainder after integer division. E.g. 17% 4 is 1. This operator is also defined for negative values, but it is different from mathematics. The sign of the result is always the same as the first operand sign: -17% 4 is -1, 17% -4 is 1, and -17% -4 is -1.

Operators can be chained, ie, multiple operators can be written. E.g. X + 2 * y is an expression that contains addition and multiplication operators. The ranking of operators evaluates operator priority (English precedence). E.g. In the expression x + 2 * y, multiplication and then addition is performed first because the multiplication operator has a higher priority than the addition operator. The other ranking order can be given by brackets: (x + 2) * y. If multiple operators with the same priority are used in the expression, the associativity of the operator (associativity) is determined by the evaluation order. E.g. In x - y - z we first make x - y and then subtract z from the result. We say the subtractor operator associates from left to right. Thus, the expression x - y - z has the same value as the expression (x - y) - z.

Some operators have the opposite, ie from right to left. An assignment operator is an example. The return value of this operator is the value of the left operand after the assignment. In the expression x = y = 1, first assignment is y = 1 and then the operator return value (in this case 1) is assigned to x. Therefore, the expression x = y = 1 is evaluated as x = (y = 1).

3.1. Operators of incrementation and decrementation

The Increment Operator (++) causes the value of the variable to be incremented by one. It can be written in two ways: prefixed and postfixed. In the prefix notation, the operator precedes his operand, followed by a postfix. In both cases, the variable is incremented, but the difference is in the return value of the operator. The prefix operator returns the value of the variable after magnification, the postfix operator value before magnification.

```
int x = 1;
int y = ++x;
```

In the expression y = ++ x, the increment operator first performs because it has a higher priority than the assignment operator. This will increase the x value by one. The return value of this operator is x after magnification, ie 2. This is used as an assignment operator. Y will be 2.

```
int x = 1;
int y = x++;
```

In the expression y = x ++, the incremental operator first performs. Its return value is the value of the variable x before magnification, ie 1. The value y is stored in the variable y.

The decrementation operator (-) causes the value of the variable to decrease by one. It is used similarly to the increment operator.

```
int x = 1;
System.out.println( --x );
```

3.2. Logical operators

Logical expressions can be joined together by logical operators. The logical operator has boolean operands and returns a boolean value. There are 2 logical operators.

The operator and && returns true when and only if both operands are true.

```
if( x == 0 && y == 0 ) {
     System.out.println( ”x and y are equal to zero” );
  }
```

The operator or || returns true when minimaly one from operands are true.

```
if( x == 0 || y == 0 ) {
    System.out.println( ” At least one of the numbers x, y
 is equal 0” );
  }
```

Both operators use the so-called abbreviated evaluation, ie the second operand is evaluated only if the value of the whole expression is not known after the first operand is evaluated. E.g. If the first operand has a false value in the logical product, the second operand is not evaluated and the value of the whole expression is false. Because these operators are often used in terms, they are conditional.

In addition to conditional operators, Java also has operators that always evaluate both operands. A & (logical product) is written and | (Logical sum). They can be used in the same place as conditional operators.

```
boolean b1 = x > 0 & y == 1;
boolean b2 = x <= 0 | y <= 0;
```

If we combine and and or, it is necessary to keep in mind that and has a higher priority than or:

```
if( x == 0 || y > 0 && z > 0 ) {
   System.out.println( "x is zero or y and z are positive " );
}
```

3.3. Assigning operators

We have already recognised one of the assigning operators, the operator =. Other assignment operators allow us to perform some arithmetic operation with the variable. E.g. The operator + = adds a second operand to the variable.

```
x += 5;
```

Other assignment operators are - =, * =, / =,% =. Each of these is a record of the corresponding operation with a variable on the left. E.g. Operator% = performs the modulo operation:

```
x %= 6;  // stejné jako x = x % 6
```

3.4. Priority of operations

All operators return a value that is the result of the operation. They have the same priority and associate from right to left. In the statement,

```
int y = x + = 1;
```

the operator first performs the = = and only then =. The + operator = returns the value x after adding 1. This value is used as the value of the second operator of the operator =.

We can sort the probed operators by priority (from highest to lowest):

- Increment (++), decrement (-)
- casting
- multiplication (*), division (/), modulo (%)
- addition (+), subtraction (-)
- assignment operators (=, + =, - =, * =, / =,% =)

Priority tells us how strongly operators are weighing on operands. E.g. Casting has a higher priority than multiplication, therefore, in the expression (int) d * 2, the casting is done first and then multiplied.

```
double d = 5.8;
int i = (int) d * 2;  // same like ((int) d) * 2
System.out.println( i );
```

The result will be 10.

Assign operators associate from right to left and arithmetic operators (addition, subtraction, multiplication, division, modulo) from left to right.

All binary operators evaluate the operands in the same order: first left and then right. This order is significant if the operand evaluation has a side effect.

```
int x = 0;
int y = x + x++;
```

On the second line, the + operator is first evaluated. Its left operand is 0, the right operand is also 0, so the operator returns 0 and assigns it to y. When evaluating the right operand, the variable x 1 is increased (the evaluation has a side effect). If the order of the operands is changed, the value of 1 is assigned to y.

```
int x = 0;
int y = x++ + x;
```

3.5. Relational operators and equality operators

We use so-called relational operators and equality operators to write conditions. Relational operators are:

- less than (<)
- greater than (>)
- less than or equal to (<=)
- greater or equal (> =)

Equality operators are:

- equals (==)
- is not equal (! =)

4. BASIC PROGRAMMING STRUCTURES

4.1. If

The if and switch commands are used to branch out the program. The if statement lets you break a program by some condition. It starts with the keyword if followed by boolean in brackets. Boolean is an expression whose value is true or false. Then follows the command. When executed, the expression in brackets is evaluated and if true (the condition is met), the command is executed. In the example below, it is tested whether the variable x is zero. If it is equal to zero, the variable x is assigned a value of 2.

```
If (x==0) x=2;
```

We use relational operators and equality operators to write the condition.

The if statement can contain the else branch that is executed if the condition for if is not met. If the condition is not met and the other branch is missing, nothing will be done (will continue after the if statement).

```
if( x == 0 )
      System.out.println( ” Can not be divided by zero!” );
  else
      z=5/x;
```

If you want to write multiple commands, we will use a block. The block begins with the opening bracket {and ends with the closing bracket}. The block is a Java command, so we can use it everywhere there's a command.

```
if( x == y ) {
      x++;
      y--;
  }
```

The block limits the validity of the variable declaration. Each declaration is valid only until the end of the block in which it is listed. We say it is local in this block.

```
if( x == y ) {
    int z = y;
 }  // This parenthesis terminates the declaration of the
//variable z
// Here you can not use z
```

We can use a boolean variable in parentheses.

```
// In the month variable we have the serial number of the
 month
  boolean isMay = (month == 5);
  if( isMay ) {
     System.out.println( ”time for love” );
  }
```

To write the opposite condition, we use the logic negation operator (!).

```
boolean isHere = true;
 if( ! isHere ) {
    System.out.println( ”Is not here” );
 }
```

If we need to branch off program eg. By the value of integer variables, we can use a concatenation of if statements.

```
if( x == 1 ) {
      System.out.println( ”one” );
  } else if( x == 2 ) {
      System.out.println( ”two” );
  } else if( x == 3 ) {
      System.out.println( ”three” );
  }
```

4.2. Switch

Thus, if (as in the previous example) is chained together, we can sometimes replace it with the switch command. His entry begins with the key word switch. Behind it is an int or String type (or a type that can be converted to int) in brackets, and a block with any number of case labels. For every case, there is a constant (or a constant expression, an expression whose value is known for translation), a colon and a sequence of commands.

```
switch ( x ) {
    case 1:
       System.out.println( ”one” );
       break;
    case 2:
       System.out.println( ”two” );
       break;
    case 3:
       System.out.println( ”three” );
 }
```

In execution, the expression is considered to be the keyword switch and its value is compared with the values given in case (in the order in which they are listed). Once commencing, orders commence. Execution ends with the break command. If the break command is missing, all commands are executed until the end of the switch command. The switch command can contain a default branch that is executed if no case label matches.

5.CYCLES

Cycles are used to re-execute commands.

5.1. While

The while cycle begins with the keyword while followed by the condition and the cycle body in brackets. The body of a cycle is either a command or a block. Within the while cycle, the condition is evaluated and, if fulfilled, the body of the cycle is executed. Then the condition is reevaluated and, if satisfied, the cycle body has performed again, etc. If the condition is not met, the commands per cycle are continued. If the condition is not met at the start, the cycle body will not be executed once. The cycle while is, therefore, a cycle with a repeat count of 0 or more.

Example of syntax:

```
While(condition) {
// body
}
```

A concrete example

```
int x = 5;
while( x > 0 ) {  // Do until x is greater than zero
     System.out.println( x );
     x --;
  }
```

This example writes numbers from 5 to 1. It will, therefore, run 5 times.

5.2. Do while

The cycle into while begins with the keyword in which the body of the cycle is followed, followed by the keyword while and the condition, see the syntax example. This is done as follows: first, the cycle body is performed, the condition is evaluated and, if satisfied, the cycle body is re-performed, the condition is evaluated, etc. If the condition is not met, it continues after the cycle. The cycle body is always at least once do.

Example of syntax:

```
do {
// body
} while (condition);
```

A concrete example

```
do  {
System.out.println( x );
        x --;
} while (x > 0);
```

5.3. For

The cycle for has the form: for (*cycle variable, condition, command*) It is done as follows: initialization of the control variable cycle first, then the condition is evaluated and, if fulfilled, the body of the cycle is executed. Then the command is executed, the condition is reassessed, and if satisfied, the body of the cycle is repeated again, etc. Initialization of the control variable of the cycle is performed only once at the beginning. If the condition is not met at the start, the cycle body will not be executed once.

Example of syntax:

```
for (cycle variable; condition; command){
// body
}
```

A concrete example

```
int a;
  for( a = 1; a < 10; a++ ) {
     System.out.println( a );
  }
```

The cycle variable can be newly declared within the cycle. This declaration is valid only in the given cycle (ie in the header and body of the cycle). We say that the variable is local in this cycle.

```
for( int a = 1; a <= 10; a++ ) {
     System.out.println( a * a );
  }
```

Any of the parts that control the cycle variable, condition, or command may be missing. If the condition is missing, it is an endless cycle (the condition is still met). If no control cycle or command variable is given, no command is executed.

5.4. Break a continue

In the body of the cycle, we can use the break and continue commands. The break command immediately terminates the execution of the cycle.

```
int s = 100;
  while( s > 0 ) {
     int n = sc.nextInt();
     if( n == 0 ) {
        break;
     }
     s -= n;
     System.out.println( s );
  }
  // Here will continue after the break executed
```

The continue command terminates the execution of the cycle body and goes to the cycle condition.

```
int s = 0;
  do {
     int n = sc.nextInt();
     if( n == 0 ) {
        continue;
     }
     s += n;
     System.out.println( s );
     // here goes continue
  } while( s < 100 );
```

6.STATIC METHODS

So far, we have written the entire program into the main method. If the same sequence of commands had to be performed in multiple places, we had to repeat these commands. This can be avoided. The methods allow us to divide the code into logical units and reuse them. In this chapter, we will understand the method as a static method. We already know one static method. It's the main method. In addition to the main method, we can declare other methods in the class, such as the method for printing program information.

```
class MainClass {
   static void printInfo() {
      System.out.println( ”Version: 1.0” );
      System.out.println( ” Autor: 007” );
   }
   public static void main( String[] args ) {
      printInfo ();
   }
}
```

The declaration of the static method begins with the keyword static. (See example above) followed by the return type and method name. The return type can be any Java type. If the method returns no value, we will specify the return type as void. The method name usually begins with a lowercase letter. If the name consists of more words, we divide the words by writing the first letters of the other words. E.g. SpoctiPolomerCruzniceOpsane. It is not customary to use the underscore method in the name. Behind the name of the method is the parameter list in brackets. The parameter list is made up of variable declarations. The separator of declarations in the parameter list is a comma.

6.1. Calling methods

In the place where we want to perform the method, we will write the method call. The method call consists of the name of the method and the parameter list. The return value of the method will be the value of the expression that is given behind the return statement.

The order in which we declare the methods does not matter. You can also call a method that is declared later.

```
class MainClass {
    public static void main( String[] args ) {
        Scanner sc = new Scanner( System.in );
        int x = sc.nextInt();
        long factorial = countFactorial( x );
        System.out.printf( "%d! = %d%n", x, faktorial );
    }
    static long countFactorial ( int n ) {
        long fact = 1;
        for( ; n > 1; n-- ) {
            fact *= n;
        }
        return fact;
    }
}
```

In the void method, you can use the return statement without parameters. This is used for premature termination of the method.

```
// print rectangle a x b from @
static void printRectangle ( int a, int b ) {
   // The minimum rectangle side size is 2
   if( a < 2 || b < 2 ) {
      return;
   }
   for( ; a > 0; a-- ) {
      for( int i = 0; i < b; i++ ) {
         System.out.print( '@' );
      }
      System.out.pritln();
   }
}
```

The return statement may occur multiple times in the method. However, it will always be done once as the last command of the method. Its execution causes the method to terminate immediately.

```
static boolean isPrimeNumber ( int n ) {
    if( n == 2 ) {  // 2 prime number
       return true;
    }
    if( n % 2 == 0 ) {
       // Even number is not a prime number (except 2)
       return false;
    }
    int sqrt = (int) Math.sqrt( n );
    for( int i = 3; i <= sqrt; i += 2 ) {
       if( n % i == 0 ) {
          // We found a divisor, so n is not a prime number
          return false;
       }
    }
    return true;
 }
```

7.INSTANCE VARIABLES

We refer to instance variables as instance attributes or shorter attributes (instances of instances or arrays). Static methods are already known. They are declared using the static keyword.

Instantial methods are declared similar to static. Unlike static methods, however, their headers do not contain the static keyword. Instantial methods often work with instance attributes.

7.1. Array

The array allows you to work with multiple values of the same type. For example, to store twenty int values, we can either enter twenty variables or create twenty element arrays. In many cases, the array works more easily. Type the array type in Java using square brackets. The declaration of the variable type array of the int items looks like this:

```
int[] p;
```

Array type variable is a so-called reference. It will contain a reference (reference) to the array. The array declaration itself does not. You can create an array using the new keyword:

```
p = new int[6];
```

In this case, we created an array of six int elements. If we need a number of array elements, we will use the NameArray.length. In our case

```
p.length
```

The value of this variable is set when an array is created and can not be changed (read only).

```
System.out.printf( "array p has %d elemets%n", p.length );
```

We access the individual elements of the array using indices, which are written in square brackets:

```
p[1] = 5;
```

The indices begin always from zero, the first number will have index 0, the second number will have index 1, the third number will have index 2 etc. The validity of the index is checked at runtime. Using an invalid index will cause a runtime error. Once created, array elements are initialized to values whose internal representation is 0. For numeric types, this is 0, for boolean it is false and for character char, it is a character at position 0 in the Unicode table. We use the for loop to read values in the field:

```
int[] p = new int[10];
  for( int i = 0; i < a.length; i++ ) {
     p[i] = i;
  }
```

We will do most of the list of field elements again with the for loop:

```
for( int i = 0; i < a.length; i++ ) {
     System.out.println( a[i] );
  }
```

The length of the field must be non-negative. An attempt to create a negative length field causes an error. Field creation can be combined with initialization. In this case, new is not used:

```
int[]numbers = { 3, 5, 6, 7};
```

The field size is given by the number of values in compound brackets. The array may be a parameter of the method and may also be a return type.

7.2. Multidimensional arrays

So far we have used one index to determine the element in the field. We call this one-dimensional field. Java allows you to declare and create multidimensional arrays. E.g. The two-dimensional field of int items looks like this:

```
int[][] p;
```

A multidimensional field in Java is a field of fields. The variable p is a reference to an array whose elements are one-dimensional arrays of integers. Create a field using the new keyword:

```
p = new int[2][3];
```

We access the field elements using indexes:

```
p[0][1] = 1;
```

The number of array elements is in the variable length of the field.

```
System.out.printf( "pole p má %d řádků%n", p.length );
System.out.printf( "první řádek má %d sloupců%n", p[0].length );
```

We use nested cycles when working with multidimensional arrays:

```
for( int i = 0; i < p.length; i++ ) {
     for( int j = 0; j < p[i].length; j++ ) {
        p[i][j] = 1;
     }
  }
```

Multidimensional arrays can be created sequentially. The subfields can then have a different number of elements. Thus, the two-dimensional array is not necessarily rectangular.

8. CLASSES

A class is a form that describes a uniquely bounded data set (s) and operations above them. We use the class to create instances, individual objects that contain the data itself (over which the corresponding operations are called).

For example, let's have a car class, this class describes that the car has a tag, type, age and mileage, and contains the information operation, the object (), which lists the type, age and non-tag type. Using this teme agee - class - then we create individual instances, in real life, we would describe them as specific vehicles.

Each newly created instance (vehicle) in the program assigns data (brand, type, age and mileage). When we call the operator information later on (), this object (instance) will write us the message specific to the vehicle.

8.1. Class declaration

For the class declaration, we use the class keyword that is preceded by the access specifier followed by the class name. The class body itself is enclosed in a block (brackets). If the name is composed of multiple words, the first letter of each word is usually written in large letters (eg ListingInfo). We do not use underscore. In the class, we can declare variables and methods. Variables and methods can either be static or instantiated.

```
class Cat {
       int weight;  // instance atribut
       int age;
      void showInfo() {  // Instance method
         System.out.println( info );
      }
  }
```

We can instantiate from the class. The class is a temwagee that tells how objects will look, what attributes and methods they will have. In our case, each Cat instance will have two int variables. By declaring the Cat variable, we introduce a variable in which we can store a reference (reference) on a Cat class instance.

```
Cat  v;
```

Because class-type variables refer to objects, they are called reference variables. Each reference variable occupies the same space: 32 bits in 32-bit JVM and usually 64 bits in 64-bit JVM. On the other hand, objects of different types usually have different sizes. The size of the object is given by its attributes. We create objects using the new keyword:

```
V = new Cat();
```

After executing this command, the variable v will contain a reference to an instance of the Cube class. Attributes and methods are accessed using a dot. We call the method by using the method name and the parentheses:

```
v.info = 1;
v. showInfo ();
```

We can create any number of instances from one class. These instances are independent of each other.

```
Cat v2 = new Cat();
v2. showInfo = 2;
v2. showInfo ();
```

Instantial methods are used to perform operations over objects of the given type. E.g. In My class we can declare a method that will increase the value of the attribute x by 1:

```
class My {
   int x;
   void IncreaseA () {
      a++;
   }
}
```

Instantial methods, as well as class methods, can have parameters and return value. Parameters and return value are specified in the method declaration.

```
class My {
      int x;
      // Adds dx to x and returns a new x value
      int moveX( int dx ) {
         x += dx;
         return x;
      }
  }
```

9. ACCESS SPECIFIERS

We use access specifiers to specify access rights to individual classes, their operations, and variables. Their importance is above all in concealing implementation details that the user can not (not) see and possibly use.

Public ..... From any class.

Private ..... Only within the given class, no access from the outside.

Protected ..... From any class of the same package, or from the descendant of the class anywhere.

none (package-friendly) From any class of the same package.

As an example, we have an employee who has his / her age and salary. It also contains the IntroduceYourself method

```
class Employee {
   public Employee (int age, int wage) {
       this.age = age;
       this.wage = wage;
    }
      private int age = 1;
   public int getAge () { return age; }
   public void setAge(int age) { this. age = age; }
   private int wage = 1;
      public int getWage() { return wage; }
   public void setWage(int wage) { this.wage = wage; }
   public void introduceYourself(){
       System.out.println("My age a wage are "  +
      age + "years "+ wage + "Euros");
    }
 public static void main(String[] args) {
      Employee employee = new employee (30,100);
   }
}
```

9.1. Heredity

Using heredity, we can create a class hierarchy in which we can tell any node that it is a special case of any of its ancestors. In the normal world, we would say that a chair is a type of furniture, an airplane is a type of means of transport. But we can not inherit the chair from the animals because they also have 4 legs!

In Java, to create a subtype in the class header, immediately after the name, we include the extends keyword and the expanded class name. This class will inherit all non-private (including package friendly, if the extension class in the same package) methods and class variable ancestors that can be declared and overlapped again.

On the contrary, the class does not inherit the private and static methods of its predecessor, since they relate only to the particular class of the ancestor. End (final) methods marked as a descendant, the class inherits but can not overlap them. Each child object can be cast on an ancestor.

9.2. Super

When calling subtype methods, we often find that we do not want to overlap the whole method, we just want to add more functionality to it. At this point, we can call super.method (), calling the ancestor functionality. We can also call the ancestor constructor by calling super () - this must be the first call in the descendant constructor.

As an example, we created the class director, which is derived from the Employee Class. However, this does not mean that this class should have access to all private variables and methods of the original class automatically. See. Access specifiers.

Calling the parent class constructor is done using the super keyword and must be the first command in the constructor's body. If we do not call the constructor of the parent class, it automatically joins the program - in this case, the so-called implicit constructor, ie a constructor without parameters.

```
class Director extends Employee {
   public Director(int age, int wage)
   {
      super(age,wage);
         }
      public static void main(String[] args) {
      Employee k = new Director(30, 50);
        }
}
```

10. POLYMORPHISM

We can overlay any method in a child's own implementation. When we call this method over a given object, the overlapping code will always be executed. Regardless of whether we approach the method by reference to an ancestor object or a descendant object (whose class contains that overlap).

This property is achieved by late binding, when the type of object on which the method is called will be decided at runtime, not during compilation.

This, however, is only a wager for overlapping methods, not for overloading them. If we have two methods on a given object that will differ only by the parameter (one for the ancestor, the other for the descendant), then the method that has the same parameter in the parameter as the current reference to the object is called. Calling overloaded methods is decided at the time of translation when it still does not have to be clear whether the reference will point to an ancestor or offspring object.