Computer Science for Engineers
Exercise 3

Introduction to Java

Prof. Dr. Dr.-Ing. J. Ovtcharova
Dipl. Wi.-Ing. Dan Gutu
28th of November 2008
Outline of the First Exercise

Exercise 1

1. Object Creation
2. Statements and Declarations
3. Flow Control Elements
4. The Eclipse Development Environment
• Declaration of an object = declaration of a variable of the data type of the object’s class

• Syntax:

```
classname objectname;
```

• example: class AccountWithBankXYZ
  "desired objects": myAccount, anotherAccount, yetAnotherAccount

declaration of the objects:
AccountWithBankXYZ myAccount;
AccountWithBankXYZ anotherAccount, yetAnotherAccount;
At object declaration no memory space is reserved. This happens not until **object creation**.

Object creation must be explicitly expressed as an instruction, because it must be defined how much memory space is needed for an object of a certain class.

At object creation all instance variables of the object are initialized.

Precisely: object creation is performed by calling a special method – the **constructor** – of the according class.
Constructors (1)

- Constructors are special methods without explicitly declared return values (Return values are always objects of the respective class!).
- Constructors have the same names like their respective classes (consider upper/lower case!)
- Rule for the lab course: in the constructor all instance variables have to be initialized.
- Syntax of the call:

  \[
  \text{objectname} = \text{new classname ( Parameter(s) ) ;}
  \]

- Example: class AccountWithBankXYZ
  
  constructor name: AccountWithBankXYZ()  (here: no parameters)
  
  object declaration: AccountWithBankXYZ myAccount;
  
  object creation: myAccount = new AccountWithBankXYZ();
- **Method overloading**: methods of the same class have the same name but different parameter lists.

- Can be used for class methods, instance methods, and constructors.

- **Most common use**: multiple constructors for one class.

- Examples:

  ```java
  public AccountWithBankXYZ() { // ... }
  public AccountWithBankXYZ(int acNumber, String holderName,
                            double acBalance) { // ... }
  ```

  ```java
  public void Rectangle(double p1x, double p1y,
                        double p2x, double p2y) { // ...}
  public void Rectangle(double side1, double side2) {
    // ...}
  ```
Constructors (3)

- Declaration of a constructor:
  
  ```
  public Class name
  ( Parameterlist )
  ```

- End of an instance method:

- Parameter list

  ```
  data type variable name , ...
  ```

- A constructor must also be declared public!
Constructors (4)

Declaration is done by the

**constructor**.

A constructor always uses the class name.

Each class has **at least** one constructor.

If no constructor is explicitly declared, the **default constructor** is used.

The default constructor has **no parameter list** and initializes all the variables to null.

If another constructor is defined, the default constructor is not available any more.

If required, it must be implemented **explicitly**!
Constructors (5)

The simplest type of constructors has no parameters.

This constructor replaces the default constructor.

Each instance of the class Circle is initialized with the same center and radius.

```java
public class Circle {
    private double x, y;
    private double r;

    public Circle() {
        x = 1.5;
        y = 2.0;
        r = 2.0;
    }

    // Other methods
    // ....
}
```
A constructor can contain supplementary parameters. Example:
• x-coordinates of the center
• y-coordinates of the center
• Radius

public class Circle {
    private double x, y;
    private double r;
    // ...
    // new constructor
    public Circle(double x1, double y1, double r1) {
        x = x1;
        y = y1;
        r = r1;
    }
    // other methods
    // ...
}
public class Circle {
    public double x, y, r;

    // Constructors
    public Circle() {
        x = 2.5;
        y = 3.0;
        r = 1.5;
    }
    public Circle( double x, double y, double r ) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
    public Circle(Circle c) {
        x = c.x;
        y = c.y;
        r = c.r;
    }

    public double circumference() { return 2 * 3.14159 * r; }
    public double area() { return 3.14159 * r * r; }
}

Circle a = new Circle();
Circle b = new Circle( 1.0, 1.0, 2.0 );
Circle c = new Circle(b);
Example: Object Creation by Instantiation of a Class (1)

Creation of a circle object in Java:

```java
Circle c = new Circle();
// initialize the circle so that it has
// center (2.5/3) and radius 1.5
c.x = 2.5;
c.y = 3.0;
c.r = 1.5;
```
Example: Object Creation by Instantiation of a Class (2)

Without parameters

Circle c1 = new Circle ()

```java
public Circle() {
    x = 1.5;
    y = 2.0;
    r = 2.0;
}
```

With parameters

Circle c1 = new Circle (2.0, 3.0, 2.8)

```java
public circle(double x1, double y1, double r1) {
    x = x1;
    y = y1;
    r = r1;
}
```
Outline of the First Exercise

Exercise 1

1. Object Creation
2. Statements and Declarations
3. Flow Control Elements
4. The Eclipse Development Environment
• **Programming**
  
  = Produce a **sequence of statements** to tell a computer what to do.

  - Statements must be **understandable by humans**, but also possible for a computer to blindly follow.

• **Statements in Java:**
  
  - Similar to statements in C und C++
  
  - **Executed sequentially**
    
    - exception: flow control or exception handling statements
      
      ▪ Executed according to their purpose.
      
      ▪ By themselves have no value.
• The **building blocks** of Java programs are shown in the following hierarchy:

  Classes

  *Global variables*

  Methods

  *Local variables*

  Statements

• A **class** includes **objects with the same characteristics**

• A **method** describes **processing of data** and contains a number of statements

• Statements end with a **semi-colon**; there are different types of statements
Types of Statements in Java

- Declarations
- Null statements
- Block statements
- Label statements
- Expression statements
- Synchronisation statements
- Unreachable statements

- Flow control statements
- Decision statements
- Iterative statements
- Branching statements
- Security statements
• **Declarations** define variables
  
  - Variables can be of type:
    - Classes
    - Interfaces
    - Arrays
    - Objects
    - Primitive data types
  
  - The correct format of a declaration depends on the declaration type.

• Java defines the following **declaration types**:
  
  - Class declaration
  
  - Interface declaration
  
  - Data array declaration: `int b[] = new int[10];`
  
  - Object declaration: `ClassName c = new ClassName();`
  
  - Declaration of primitive data types: `int a = 0;`
Null statements

- Null statements occupy code space, but don‘t have any function
- Used where an instruction is needed according to syntax, but nothing needs to occur
- Syntax:
  - ;
  - (Single semi-colon)
- Avoid!
  - Often there is a better, more elegant implementation
Block statements (1)

class

method

{ 
statement 1
statement 2

{ 
statement 3
statement 4

} 

statement 5

}

Block statement

Nested statement block

Statement block
In Java methods and static initialisations are defined by block statements and code blocks

Block statements:
- Several rows of statements, in between brackets ({});
- Statements can be arranged into further sub-statements.
- Sub-statements are blocks within a statement.
- A statement block has its own space for included statements.
- Within this block, local variables can be declared.
  - Local variables are not valid outside of the block.
  - After execution of the block, local variables no longer exist.
- Blocks can often be nested.

Common sources of errors:
- Errors arise from incorrectly closed blocks
Label statements (1)

• Every statement in Java may have a label.

• Names of keywords or any other locally defined labels must not be used.

• The name must always be followed by a colon.

• Used by the statements `break` and `continue`.

• Avoid!
  
  - There are often better, and more elegant implementations
Label statements (2)

Arbitrary labels
(no keywords)

statement 1

XXX: statement 2

statement 3

YYY: statement 4

ZZZ: statement 5
Operators and Expressions

• **Logical, arithmetic and comparison operators:**
  - &, |, ! (log. Negation)
  - ==, !=, >, >=, <, <=,
  - +, -, *, /, % (Modulo), - (Negation)
  - Increment ++, Decrement --, Addition/Subtraction assignments += , -=

• **Expressions: Application of operators:**
  - (2+7) is an expression

• Expressions have a **result** and sometimes **side effects**
Expressions and Side Effects

- Expressions have results and sometimes side effects

- \(3 + 2\)
  - Result: 5
  - Side effect: none

- \(a += 3\)
  - Result: \(a+3\)
  - Side effect: \(a = a+3\)
  - Equivalent to \(a = a+3\)

- \(++a\)
  - Result: \(a+1\)
  - Side effect: \(a = a+1\)

- \(--a\)
  - Result: \(a\)
  - Side effect: \(a = a-1\)

- \(a++\)
  - Result: \(a\)
  - Side effect: \(a = a+1\)
• In Java there are **different types of expression statements**.

• Expression statements must be ended by placing a **semi-colon** at the end:
  
  - \((2+2)\);
  
  - Expression will be evaluated.
  
  - Expressions have results and sometimes side effects.

• An expression must be **completely executed** before the next statement can be carried out.
  
  - It is therefore important to pay attention when closing statements in complex nestings.

• On the **left hand side** of an assignment there should always be a **variable**.
## Types of Expression Statements

<table>
<thead>
<tr>
<th>Expression statement</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| Assignment               | `a=42;`               | Result: 42  
Side effect: Assigns a value to a variable             |
| Post-Increment           | `a++`                 | Result: `a`  
Side effect: `a` contains `a+1` or `a-1`               |
| Post-Decrement           | `a--`                 |                                                       |
| Pre-Increment            | `++a`                 | Result: `a+1` or `a-1`  
Side effect: `a` contains `a+1` or `a-1`               |
| Pre-Decrement            | `--a`                 |                                                       |
| Method call              | `System.out.println("Hello");` | Invokes a method                                       |
| Expression assignment    | `Testclass t = new Testclass();` | Assigns a new object of type Testclass to variable t |
(1) // Increment example - or decrement operator ++
(2) int n = 7;

(3) n++; // n now has the value 8
(4) ++n; // n now has the value 9
(5) n--; // n now has the value 8
(6) --n; // n now has the value 7

(7) int a = n++;
(8) // a now has the value 7, n the value 8

(9) int b = ++a;
(10) // b now has the value 8, as does a.
(1) // Increment example - or decrement operator ++
(2) int n = 7;

(3) n=n+1; // n contains the value 8
(4) n=n+1; // n contains the value 9
(5) n=n-1; // n contains the value 8
(6) n=n-1; // n contains the value 7

(7a) int a = n;
(7b) n=n+1;
(8) // a contains the value 7, n the value 8

(9a) a=a+1;
(9b) int b = a;
(10) // b contains the value 8, as does a.
Outline of the First Exercise

Exercise 1

1. Object Creation
2. Statements and Declarations
3. Flow Control Elements
4. The Eclipse Development Environment
Definition:
Logical values are elements of the set \{T, F\}, with

- \( T = \text{"true"} \)
- \( F = \text{"false"} \)

• Basic operations on logical values:
  - **Unary operations**
    - Negation, "not"
  - **Binary operations**
    - Conjunction, "and"
    - Disjunction, "or"
### Basic operations on Logical values

#### Unary operations

<table>
<thead>
<tr>
<th>Negation</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

#### Binary operations

**Conjunction (Logical AND)**

<table>
<thead>
<tr>
<th>Conjunction</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

**Disjunction (Logical OR)**

<table>
<thead>
<tr>
<th>Disjunction</th>
<th>T</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
### Comparative operators in Java

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
<th>examples</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal to</td>
<td>7 == 7</td>
<td>returns true</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
<td>6.21 != 6.2</td>
<td>returns true</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>6 &lt; 6</td>
<td>returns false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.2 &lt; 6.21</td>
<td>returns true</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>6.2 &gt; 6.21</td>
<td>returns false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.21 &gt; 6.2</td>
<td>returns true</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>6 &lt;= 6</td>
<td>returns true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.2 &lt;= 6.21</td>
<td>returns true</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>6 &gt;= 6</td>
<td>returns true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.21 &gt;= 6.2</td>
<td>returns true</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>true &amp;&amp; true</td>
<td>returns true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>true &amp;&amp; false</td>
<td>returns false</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logical OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical NOT</td>
<td>! true</td>
<td>returns false</td>
</tr>
<tr>
<td></td>
<td></td>
<td>! false</td>
<td>returns true</td>
</tr>
</tbody>
</table>
Example: Boolean Logical Operators

```java
Class BoolLogic {
    public static void main(String arg[]) {
        boolean a = true;
        boolean b = false;
        boolean c = a || b;  // (Logical OR)
        boolean d = a && b;  // (Logical AND)
        boolean e = a ^ b;   // (exclusive OR)
        boolean f = (!a && b) | (a && !b);
        boolean g = !a;

        System.out.println("a = " + a);
        ...
    }
}
```

The output of this program is:

- `a = true`
- `b = false`
- `a||b = true`
- `a&&b = false`
- `a^b = true`
- `!a&&b|a!b = true`
- `!a = false`
<table>
<thead>
<tr>
<th>Weight</th>
<th>operations</th>
<th>operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>concatenation</td>
<td>,</td>
</tr>
<tr>
<td>2</td>
<td>assignment</td>
<td>= += -= &lt;&lt;= &gt;&gt;= &amp;= ^=</td>
</tr>
<tr>
<td>3</td>
<td>if-then-else</td>
<td>? :</td>
</tr>
<tr>
<td>4</td>
<td>logical OR</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>logical AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>6</td>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>XOR</td>
<td>^</td>
</tr>
<tr>
<td>8</td>
<td>AND</td>
<td>&amp;</td>
</tr>
<tr>
<td>9</td>
<td>equality, inequality</td>
<td>== !=</td>
</tr>
<tr>
<td>10</td>
<td>relation</td>
<td>&lt; &lt;= &gt; &gt;=</td>
</tr>
<tr>
<td>11</td>
<td>dislocation</td>
<td>&lt;&lt;= &gt;&gt;= &gt;&gt;&gt;</td>
</tr>
<tr>
<td>12</td>
<td>addition, subtraction</td>
<td>+ -</td>
</tr>
<tr>
<td>13</td>
<td>multiplication, division, modulo</td>
<td>* / %</td>
</tr>
<tr>
<td>14</td>
<td>NOT, complement, decrement, increment</td>
<td>! - ++ --</td>
</tr>
<tr>
<td>15</td>
<td>parenthesis, vectors</td>
<td>( ) [ ]</td>
</tr>
</tbody>
</table>
Program Example Showing the Order of Operations

(1) int aNumber = 10, anotherNumber = 16;
(2) int anExpression1 = aNumber + 2*(anotherNumber-10)%4;
(3) System.out.println("anExpression1 = " + anExpression1);
(4) int anExpression2 = aNumber + 2*((anotherNumber-10)%4);
(5) System.out.println("anExpression2 = " + anExpression2);
(6) boolean testVariable1 = true, testVariable2 = false;
(7) boolean anExpression3 = testVariable1 & !testVariable2;
(8) System.out.println("anExpression3 = " + anExpression3);

Output on the screen:
anExpression1 = 10
anExpression2 = 14
anExpression3 = true
Program Example for the Operators |, &, ||, &&

(1) int m=4, n=4;
(2) boolean wahr = true;
(3) boolean b1 = wahr | (m++ > 9);
(4) System.out.println("b1= " + b1 + " m= " + m);
(5) boolean b2 = !wahr & (m++ > 9);
(6) System.out.println("b2= " + b2 + " m= " + m);
(7) boolean b3 = !wahr | (m++ > 9);
(8) System.out.println("b3= " + b3 + " m= " + m);
(9) boolean b4 = wahr & (m++ > 9);
(10) System.out.println("b4= " + b4 + " m= " + m);
(11) boolean b5 = wahr || (n++ > 9);
(12) System.out.println("b5= " + b5 + " n= " + n);
(13) boolean b6 = !wahr && (n++ > 9);
(14) System.out.println("b6= " + b6 + " n= " + n);
(15) boolean b7 = !wahr || (n++ > 9);
(16) System.out.println("b7= " + b7 + " n= " + n);
(17) boolean b8 = wahr && (n++ > 9);
(18) System.out.println("b8= " + b8 + " n= " + n);

Output on the screen:

b1 = true m = 5
b2 = false m = 6
b3 = false m = 7
b4 = false m = 8
b5 = true n = 4
b6 = false n = 4
b7 = false n = 5
b8 = false n = 6
Control Flow

• So far:
  - Each statement is executed only once and strictly in a top-down way.

• Now:
  - Selection Structure: Possibility to choose which statement is to be executed next.

• Means:
  - Selection statements (e.g. if, if-else, switch)
  - Iteration statements (e.g. while, do, for)
  - Branching statements (e.g. break, continue)
  - Protection statements (try, catch, etc.)
an if statement tests a boolean variable or expression

composition:

IF condition is fulfilled
THEN something happens

IF condition is not fulfilled
THEN nothing happens

structure:

IF L THEN A

containing:
L: boolean expression
A: statement
• Inside a method algorithms can be implemented by conditions, iterations and assignments

• Verify conditions:

```java
int aNumber;
if (aNumber == 0)
    System.out.println("the number is 0");
```

• The statement block is executed only if the Boolean Expression is true

• Example:
Program Example: if - Statement

(1) boolean testVariable = false;

(2) if( testVariable )

(3) System.out.println("default");

(4) testVariable = true;

(5) if( testVariable )

(7) System.out.println("Variable is 'true'");

Output on the screen:
Variable is ‘true’
Note: Always use brackets, also for a single if-statement!

(1) boolean testVariable= false;
(2) if( testVariable )
(3) {
(4) // System.out.println("default");
(5) }
(6) testVariable= true;
(7) if( testVariable )
(8) {
(9) // System.out.println("Variable is 'true'");
(10) }
if-else-statement is an extension of if-statement

- **extension**
  - **else-part**
    - **active if** boolean result of if-statement is false
    - belongs to last if-statement without else (same in Pascal, C or C++)

- **note**
  - keywords in Java are written in lower case!!!

- **each group of one or more statements must be used in curly brackets** {}
  - missing brackets at the end of a statement cause a compiler error message
If-else Statements

```
if (Logic expression) 
  Statement or block of statements
else
  Statement or block of statements
```
Program Example: if-else - Statement

(1) boolean testVariable = false;

(2) if( testVariable ) {

(3) System.out.println("If-Branch");

(4) } else {

(5) System.out.println("Else-Branch");

(6) }

Output on the screen:

Else-Branch
**switch - Statement without break**

- **switch - Statements**
  - Allow that the control flow is moved on to another statement within the same block together with substatements.
    - To which statement the control flow is moved depends on the value of the selector expression in the statement.
  - Represent rather a controlled „jump“ than a multiple branching in a structogram.
    - A structure chart („structogram“) is possible if each case ends with a break.
  - Are not exited automatically

- **switch – Statements and constants**
  - Have to be of type `byte`, `short`, `char` or `int`
  - After the expression has been evaluated the program continues at the label marked with „konst“.

```
Expression = ?
konst1 konst2 konst3 konst4 konst5 konst6 default
```
Example `switch`-Statement without break

```java
(1) int testVar = 2;
(2) switch (testVar) {
(3)     case 0:
(4)         System.out.println("zero");
(5)     case 1:
(6)         System.out.println("one");
(7)     case 2:
(8)         System.out.println("two");
(9)     case 3:
(10)        System.out.println("three");
(11)     case 4:
(12)        System.out.println("four");
(13)     case 5:
(14)        System.out.println("five");
(15)     case 6:
(16)        System.out.println("six");
(17)     case 7:
(18)        System.out.println("seven");
(19)     default:
(20)         System.out.println("no match");
(21) }
```

Output on the screen:
```
two
three
four
five
six
seven
no match
```
• The Control flow of a `switch` – statement does not automatically exit the structure.
  - Without `break` the statement is executed from the first match until the end.
  - Note: This is different from Pascal and often a source of error!

• Corrective:
  - Put a `break`-statement at the end of each code section. Thus the execution of more than one block is avoided.
  - `breaks` can also be used with meaning.
  - Not to put a `break`-statement is a mean that allows the execution of several blocks.
• Syntax of a `switch`-statement:

```
(1) switch (integer-selector) {
(2)   case int-value1:
(3)      <statement>;
(4)      break;
(5)   case int-value2:
(6)      <statement>;
(7)      break;
(8) ...
(9)   default:
(10)      <statement>;
(11) }
```

• `case`-statement
  - Within one block, each case must have a different value.
  - `case` must be followed by a value that can be calculated during the interpretation of the program.
Example `switch`-Statement with `break`

```java
(1)  int testVar = 2;
(2)  switch (testVar) {
(3)    case 0:
(4)      System.out.println("zero"); break;
(5)    case 1:
(6)      System.out.println("one"); break;
(7)    case 2:
(8)      System.out.println("two"); break;
(9)    case 3:
(10)     System.out.println("three"); break;
(11)   case 4:
(12)     System.out.println("four"); break;
(13)   case 5:
(14)     System.out.println("five"); break;
(15)   case 6:
(16)     System.out.println("six"); break;
(17)   case 7:
(18)     System.out.println("seven"); break;
(19)   default:
(20)     System.out.println("no match");
(21)   }
```

Output on the screen: `two`
Iteration Statements

• Iteration statements are also called loops or sometimes repeating statements.

• Java knows three types of iteration statements:
  - while
  - do
  - for

• They are identical to the analogue statements in C and C++

• They have the optional parameters:
  - continue and
  - break
while – Statement

• while-statement
  - Boolean-expression is evaluated once at the beginning of the loop and again before each further iteration of the statement
    ▪ value = true
      • the corresponding substatement or block is executed
    ▪ value = false
      • no more repeating
      • jumps to next statement after the while-statement

• Source of error:
  - The loop is not executed at all: This is called a “rejecting loop”.

• syntax of a while-statement

  (1) while (expression) {
  (2)   <statement>
  (3)  }

execute as long as value = true

Statement
Example **while**- Statement

(1) int testVar= 5; // numeric variable

(2) while( testVar > 0 ) {

(3)  if ( testVar > 1 )

(4)    System.out.println( testVar + " seconds");

(5)  else

(6)    System.out.println( testVar + " second");

(7)  testVar--;

(8) }

(9) System.out.println("-+=* BOOM *=+-");

Output on the screen:

5 seconds
4 seconds
3 seconds
2 seconds
1 second
-+=* BOOM *=+-
**do - Statement**

- **do** statements test the value of a boolean variable or expression
  - **Value = true**
    - The substatement or block is executed
  - **Value = false**
    - The repetition stops
    - The loop is exited

- **the difference between while and do:**
  - The statement of the do-loop always executes at least once, even if the expression evaluates to false the first time
  - The evaluation is performed at the end of the loop

**Syntax of a do-statement**

```plaintext
(1)   do {
(2)   <statement>
(3)   } while (expression);
```

execute as long as value = true
Example do-Statement

```
(1) boolean testVariable = false;

(2) do {

(3)   System.out.println("Variable is " + testVariable);

(4) } while ( testVariable );

(6) System.out.println("... finish");
```

Output:

```
Variable is false
... finish
```
**for - Statement**

- Syntax of a for-Statement:
  - (1) for (Initialization; test; iteration)
  - (2) <statement or block>

- Initialization part contains the declarations:
  - They are only valid within the for – statement and its substataements.

- Condition part of a for-statement
  - Contains one or more declaration or assignment statements, separated by a comma.
  - Contains a boolean expression that is evaluated once in each looping
  - If the value is false the loop is exited.

- Iteration part of a for-statement
  - one or more expressions separated by comma
  - Several expressions only if already declared in the initialization part
  - Is evaluated once per looping
  - Allows to increase or decrease an index that is tested in the condition part
Program Example: `for`-statement

```java
/* Program that shows unicode character set in reverse order */
for ( int i=122; i>114 ; i-- )
System.out.println(i + " = " + (char)i);
```

Output on screen:

122 = z
121 = y
120 = x
119 = w
118 = v
117 = u
116 = t
115 = s
Iterative Statements: An Overview

• **Do-statement**
  – Loop is passed through at least once
  – Abort-test at the end

• **While-statement**
  – Loop is not compulsory passed through at least once
  – Abort-test at the beginning

• **For-statement**
  – “a complex while-loop“
  – Loop is not compulsory passed through at least once
  – Abort-test at the beginning
**break-Statement**

- **break-statements are used for:**
  - Exiting loops
  - Terminates a statement sequence in a `switch`-statement
- **break-statements with no label**
  - attempt to transfer control to the next enclosing switch: while, do, or for statement
- **break-statements with label `Identifier`**
  - transfers control to the enclosing labelled statement that has the same `Identifier` as its label
  - must refer to a `label` within the immediately enclosed method or initializer block
  - a `label` is an identifier followed by a colon right before an iteration statement
  - If an invalid or unlabelled statement with `Identifier` as its `label` encloses the `break` statement, a compile-time error occurs
• A block can be left with `break`:

```java
Class Break {
    public static void main(String arg[]) {
        System.out.println("Hello");
        {
            System.out.println("Before break");
            break;
            break;
            System.out.println("This is not executed");
        }
        System.out.println("World");
    }
}
```

**Output on the screen:**
Hello
Before break
World
Program Example: `break` – statement without label identifier

(1) int i;
(2) for( i= 0; ; i++ ) {
(3)    if ( (i * 1.0 / 234) > 12.0 )
(4)        break;
(5) }
(6) System.out.println("i= " + i);

Output on the screen:

i= 2809
Example break - Statement with label Identifier

```java
(1) int y,x;
(2) boolean found= false;
(3) search:
(4) for( y=0; y < Matrix.length; y++ ) {
(5)     for (x=0; x < Matrix[y].length; x++ ) {
(6)         if ( Matrix[x][y] == 4711 ) {
(7)             found= true;
(8)             break search;
(9)         }
(10)     }
(11) }
(12) if ( found )
(13)    System.out.println("found at("+x","+y")");
```

Matrix:

```
1   -35    67  999
-26   88   95   -51
247  199  4711   -4
  2   -19  -66    0
   23   74   15   -7
  99   55   44   22
```

Output on screen:

```
Found at (2,2)
```
• **break- /continue- statement** is very distinctive:
  
  - **break-** statement quits the loop
  
  - **continue-** statement stops the execution of the current iteration, forcing an early iteration of the loop and thereby halting the processing of the remaining code, “go to” just after the code. This causes control to be directly transferred to the conditional expression.

• **continue- statement**

  - with **no label**
    
    • immediately ends the current iteration and begins a new one
    
    • attempts to transfer control to the next enclosing `while`, `do`, or `for` statement of the immediately enclosing method or initializer block

  - with **label Identifier**
    
    • immediately ends the current iteration and begins a new one
    
    • attempts to transfer control to the enclosing labeled statement that has the same `Identifier` as its label

  - continue target must be a `while`, `do`, or `for` statement or a compile-time error occurs
Example *continue*-Statement

```c
(1) /* extract from a program to prevent
(2) division by zero*/
(2) float number, result;
(3) int divisor;
(4) // ...
(5) for(divisor = -42; divisor < 42; divisor ++ ) {
(6) if (divisor == 0 ) continue;
(7) result= number / divisor;
(8) }
(9) // ...
```
Example `continue`-Statement with label identifier

```java
(1) int y, x;
(2) boolean found = false;
(3) search:
(4) for (y = 0; y < Matrix.length; y++) {
(5)   for (x = 0; x < Matrix[y].length; x++) {
(6)     if (Matrix[x][y] == 4711) {
(7)       found = true;
(8)       continue search;
(9)     }
(10)   }
(11) }
(12) if (found)
(13)   System.out.println("found at("+x", "+y")");
```

Matrix:
```
1   -35    67  999
-26   88    95   -51
247  199  4711   -4
   2   -19   -66   0
  23    74    15   -7
 99   55    44   22
```

Output on screen:
```
Found at (3,5)
```
• Moves control back to the initializer of
  - the method
  - the constructor or
  - the static initializer
  
  that contains the return-statement

• return-statements other than void can
  - contain a parameter (the return value) of the same type as the method
  - These return values can then be processed by the calling program.
public class circle {
    private double x, y; // center of circle
    private double r; // radius
    // method to calculate circumference and circular
    // area
    public double circumference() {
        return 2*3.14159*r;
    }
    public double area() {return 3.14159 * r * r; }
    // ...
}
throw-statement

- throw-statement creates runtime exceptions during the program flow
  - The normal program will be interrupted by an exception
    - Exception must be handled by the program
    - After the handling, the program flow continues
  - The runtime exception uses an object as argument
    - The object is positioned after the throw-instruction
    - Designates a reference expression, that is usually derived from the Exception class

Exceptions will be handled in detail later during this lecture!
Exkurs: Output on the screen

- Syntax:

\[
\text{System.out.println ( Value ) ;}
\]

- Value can be:
  1) “… Text … “
  2) Variables of primitive data type or String (String)
  3) Combinations of 1) and 2), united by “+”

- Example:

  System.out.println("Output on screen");
  System.out.println(DM_PER_EURO);
  System.out.println("DM/Euro: “ + DM_PER_EURO);
  System.out.println(DM_PER_EURO + " = DM/EURO");
Comments

• Comments are used mainly for adding explanatory notes to the program

• Syntax of a comment:

// Comment

• “//” comments out the rest of the line, after the “//“

• Syntax multi-line comments:

/* Comment */
Outline of the First Exercise

Exercise 1

1. Object Creation
2. Statements and Declarations
3. Flow Control Elements
4. The Eclipse Development Environment
Eclipse
Basic Elements of the Development Environment

- A development environment is a special tool that can simplify the programming process and make it more efficient.

- In the computer lab the Eclipse development environment will be introduced.

- The source code, entered in Eclipse, will be compiled into Java-bytecode using the Java-compiler. The result (as long as no syntactical or orthographic errors are found) will be bytecode saved with the ending *.class

- The Eclipse debugger runs the program step by step allowing visualization of variable contents.
• Basically, an editor and a compiler are required
• IDEs offer comfort for the development process
  - Syntax highlighting,
  - Just in Time Compiler (during programming, errors are revealed)
  - Automatique Refactoring (improve the structure of the code)
  - ….

• Several free and commercial IDEs are available. For Java:
  - Eclipse (free)
  - Netbeans (free)
  - JDeveloper (com.)
  - Jbuilder (com.)
Eclipse Installation Guide

• Prerequisites
  - Java Development Kit (JDK)
  - Java Runtime Environment (JRE)
  - http://java.sun.com/javase/downloads/?intcmp=1281

• Eclipse IDE
  - For our purpose, „Eclipse IDE for Java Developers“ is sufficient, download at eclipse.org

• TODO:
  - Install JDK and JRE (for the respective platform)
  - Eclipse must not be installed (for Windows)
    - Download, Unpack, Start. (eclipse.exe)
Eclipse After First Start

- Eclipse has different perspectives (Java, Debug), each with special functions
- To reach the Java-Perspective go to:

  in Menu Window->Open Perspective->Java
• Creating a new project:
  - A project is a directory, where all the information for corresponding classes will be saved; for each new program a new project must be created:
    - *Menu File-* > *New-* > *Project*, then the „*New Project Wizard“ will appear
    - Select *Java-* > *Java Project* and click *Next*
    - Give the project a name and click *Finish*
First Steps

- **Create a new class**
  - Select the project in the package explorer
  - Right click on the package name and select *New->Class*
  - Leave the „Package“ field empty, the class will then be set up using the default package
  - Enter a class name (following convention must begin with a capital letter)
  - Select „**public static void main(String[] args)**“ to produce a start class
  - Click *Finish*
Start the Program

• Click on the class and enter some source code (i.e. `System.out.println("Hello world!");`)
• Eclipse automatically compiles when saving
• Select „Menu Run->Run As->Java Application“ to run the program for the first time
• To run the program again you only need to click the „run“ button
• The result is visible in the field below
Importing Classes in to an Existing Project

- Copy other Java source code into the project directory (i.e. with Explorer)
- Select the project name in the package explorer
- Right click and select refresh
- Double clicking the name of the class will show the source code in the editor
Debugging

• All syntactic and orthographic errors are recognised by the compiler when you try to save
• Mistakes in the code are marked with a red dot
• Pressing the „Debug“ button automatically switches in to the „Debug“ environment
Debug Environment
Debugger Menu

Continue (stop at the next break point)

Pause

Stop the program (cancel Debugging)

Go one step in to the next method

Go one line further; here you can observe step by step how the variable contents change
Some general rules (1)

• Code must be documented
  - Undocumented code is worthless, because its understanding is very time consuming. Existing code can be utilized erroneously.
  - Not actualized documentation can lead to misinterpretation
  - Linkage between code and documentation. For example, synchronize code and documentation with Javadoc
  - Create easy understandable documentation:
    ▪ // loop 2
    ▪ // this function gets some Features
      public void getFeatures()
    ▪ // this function can be used to load data
      public Object loadData
Some general rules (2)

- Code blocks shall not contain more than 100 lines of code
  - “Divide and conquer” must not be considered, code becomes unreadable
  - Code reutilization is difficult
- Always take into consideration that you do not know all the requirements
  - Create a program structure that always can be enlarged
- Do not optimize unfinished code
  - „Premature optimization is the root of all evil“
- Test the code carefully
  - Automatic tests through Unit Tests
  - Take the side effects into consideration (a small modification may create huge problems somewhere else)