Technische Hochschule Rosenheim
Technical University of Applied Sciences

Programming Basics – WiSe21/22

Control structures

Prof. Dr. Silke Lechner-Greite

Table of contents - overall

1. Introduction
2. Fundamental language concepts
3. Control structures
4. Methods
5. Arrays
6. Object orientation
7. Classes
8. Packages
9. Characters and Strings
10. Unit Testing
11. Exceptions
12. I/O

Concept of control structures

- Control structures = all linguistic resources which control the execution order of statements

- Control structures can be used to formulate complex algorithms that solve challenging problems.

- Description forms for algorithms:
  - Natural language
  - Source code
  - Neutral, abstract form
    (e.g. structure chart or flow diagram)
    -> presentation of the solution idea, visualisation key structures

Chapter 3: Control structures

3.1 Statement series (sequence)

3.2 Selection structures (selections)

3.3 Repeating structures (loops or iterations)

3.4 Effects on variables

Statement series (sequence) (1)

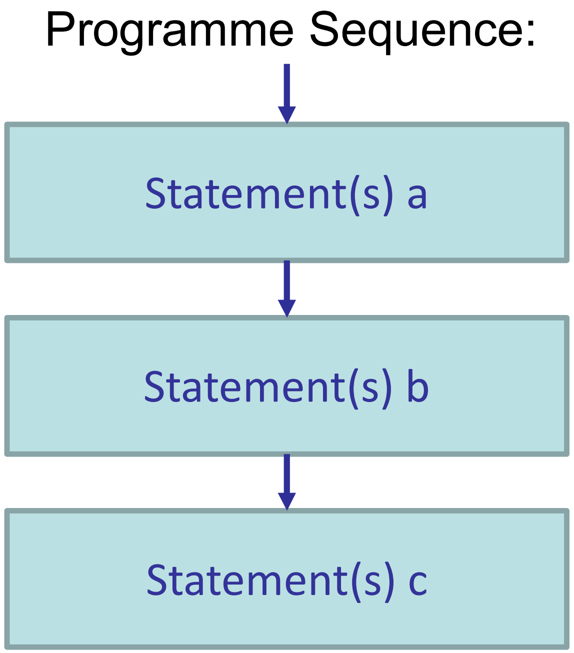
- So far: simple statements (instructions)
  - Variable definitions, value assignments and output statements
  - Cannot be broken down into smaller statements

- Control structures = compound statements
  - Complete, subordinate statements as building blocks

- Simplest control structure: sequence
  = statement series = succession of statements (instructions)

Statement series (sequence) (2)

- Individual statements that are processed sequentially from top to bottom



Based on: Habelitz (2012): Programmieren lernen mit Java

Statement series (sequence) (3)

- Recommended implementation in the programme:
  - Each individual statement should be on a new line

```
Statement a:  System.out.println("The first line!");
Statement b:  System.out.println("The second line!");
Statement c:  System.out.println("The third line!");
```

Chapter 3: Control structures

3.1 Statement series (sequence)

3.2 Selection structures (selections)

3.3 Repeating structures (loops or iterations)

3.4 Effects on variables

Selection structures

- A branch allows conditional execution of statements (selection)
  - Execution of individual statements can be made dependent on whether a certain condition is fulfilled

- Two different types:
  - `if` statements
  - `switch case` statements

`if` statement (1)

- `if` statement (= "conditional statement", "branch") consists of
  1. a condition and
  2. a subordinate statement

- Subordinate statement is only executed if the condition is met, otherwise it is skipped
- Syntax:

```
if (condition)
    statement;
```

```
if (condition)
{
    statementA;
    statementB;
}
```

`if` statement (2)

- Example: output the text "Warning!" if the variable `temperature` has a negative value; otherwise do not output anything

```
if (temperature < 0)
  System.out.println(“Warning!");
```

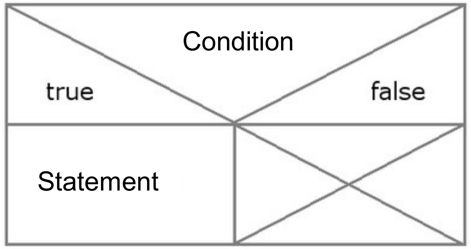
- First check the condition `temperature < 0` ….
  - *Evaluates the condition to `true`?*

    Execute the output statement
  - *Evaluates the condition to `false`?*

    Skip the output statement

`if` statement (3)

- Structure chart for `if` statements:



- Condition = expression with true/false result
  - Expression is a `boolean` expression
  - Expression can either be "true" or "false“, no third possibility

`if` statement (4)

- Condition: comparison of two (numerical) expressions
- Comparison using relational operator (= comparison operator)

Operator	Meaning	Precedence
<	less than	5
<=	less than or equal to	5
>	greater than	5
>=	greater than or equal to	5
==	equal to	6
!=	not equal to	6

Examples:

```
if (a == 0)
if (b > 10)
if (number <= 100)
```

Exercise – Simple `if` statement

- Live exercise
  - Complete Task 1 on the live exercises sheet “Control structures”
  - You have 5 minutes.

Two-way `if` statements (1)

- Two-way `if` statement: extension of the simple `if` statement
- Semantics:
  - *Is the condition met?*

    Execute the first statement
  - *Otherwise (if the condition is not met)*

    Execute the second statement
- Syntax:

```
if (condition) {
    statement a
}
else {
    statement b;
}
```

Multiple statement series are combined into one unit
= block

Blocks as statement (instruction) groups

- Sequence is grouped into a block with curly brackets
- Syntax:

```
{
   statement;
   statement;
   statement;
     . . .
}
```

- Empty block: Block without statements (instructions)

```
{   }
```

- Empty statement: equivalent to an empty block

```
;
```

Two-way `if` statements (2)

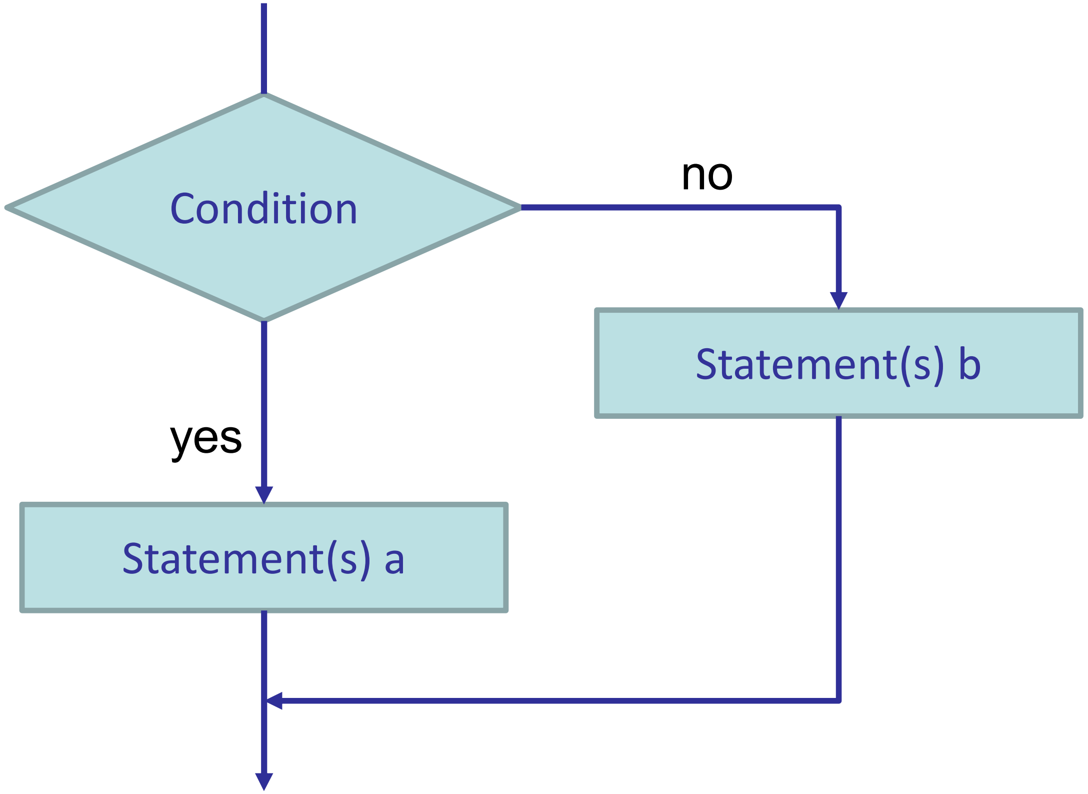
- Example:

```
double x = ...;
if (x >= 0)
  System.out.println(Math.sqrt(x));
else
  System.out.println("undefined");
```

- Important note: there is always
  - precisely one of the two statements executed, but
  - never both, and
  - never none.

Two-way `if` statements (3)

- Programme Sequence:



Based on: Habelitz (2012): Programmieren lernen mit Java

Exercise – Two-way `if` statement

- Live exercise
  - Complete Task 2 on the live exercises sheet “Control structures”
  - You have 5 minutes.

Excursus:
Comparison of floating-point values

- Floating-point arithmetic is affected by rounding errors!
- Example:

```
double a = 1.0/3.0;
double b = 10 + a – 10;
if (a == b)
  System.out.println("the same");
else
  System.out.println("different");
```

Unexpected results!

- Compare floating-point values to ranges, not individual values

```
if (Math.abs(a-b) < 1E-10))
   ...
```

Appropriate tolerance limit, depending on the purpose

Nested `if` statements

- `if` statement
  - controls other statements
  - is itself a statement
  - can be subordinated to another one: nested `if` statements
- Example: does x lie between 0 and 100?

```
if (x < 0) {
    System.out.println("below 0");
} else {
       if (x > 100) {
              System.out.println("over 100");
       } else {
             System.out.println("between 0 and 100");
       }
}
```

Dangling else (1)

- Assignment problem with two `if` and one `else`
- Interpretation options:

1. `else` assigned to the first `if`

```
if (condition)
  if (condition)
        statement 1;
else
    statement 2;
```

2. `else` assigned to the second `if`

```
if (condition)
  if (condition)
        statement 1;
  else
        statement 2;
```

Dangling else (2)

- However, compiler ignores indentation, orients itself based on the programme text: in both cases the same!
- Ambiguity not allowed => rule:
  - `else` belongs to the last free `if` in the text of the same block ("free" `if` = not yet assigned to an `else`)
- Explicit brackets create clarity and avoid errors

```
if (condition) {
  if (condition)
        statement 1;
}
else
    statement 2;
```

```
if (condition) {
  if (condition)
        statement 1;
  else
        statement 2;
}
```

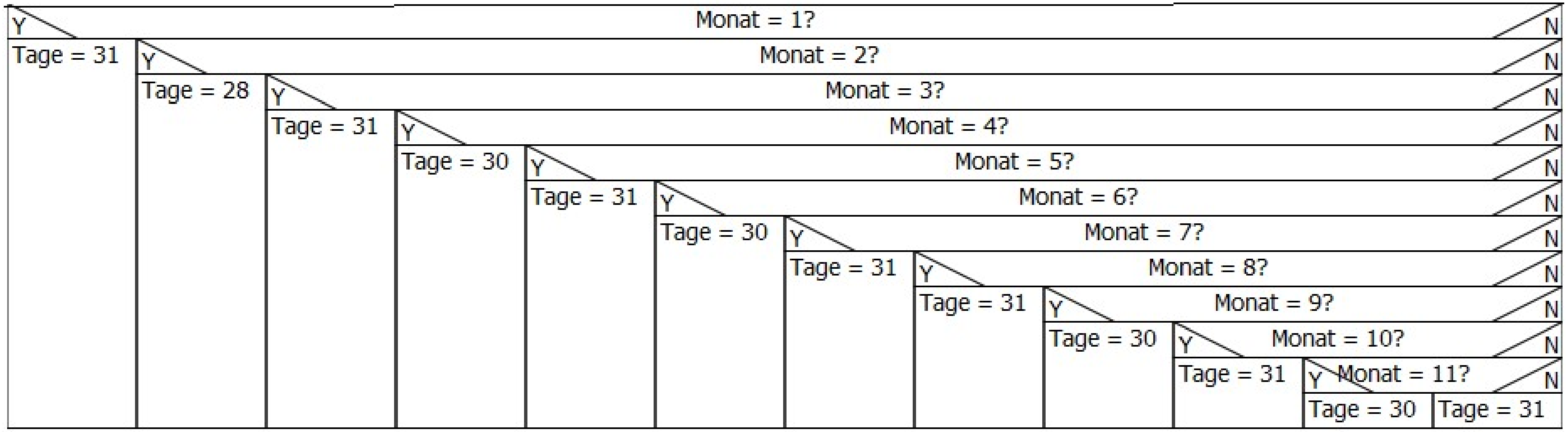
Exercise – `if` statement

- Live exercise
  - Complete Task 3 on the live exercises sheet “Control structures”
  - You have 5 minutes.

`if` cascade (1)

- ➢ Deep nesting of `if` statements = `if` cascade
- ➢ Typically: comparison of a value with a list of possibilities
- ➢ Example: convert month number to number of days

Legend:
Monat = month
Tage = days
Y = yes
N = no



Monat = 1? Y N
Tage = 31
Monat = 2? Y N
Tage = 28
Monat = 3? Y N
Tage = 31
Monat = 4? Y N
Tage = 30
Monat = 5? Y N
Tage = 31
Monat = 6? Y N
Tage = 30
Monat = 7? Y N
Tage = 31
Monat = 8? Y N
Tage = 31
Monat = 9? Y N
Tage = 30
Monat = 10? Y N
Tage = 31
Monat = 11? Y N
Tage = 30
Tage = 31

`if` cascade (2)

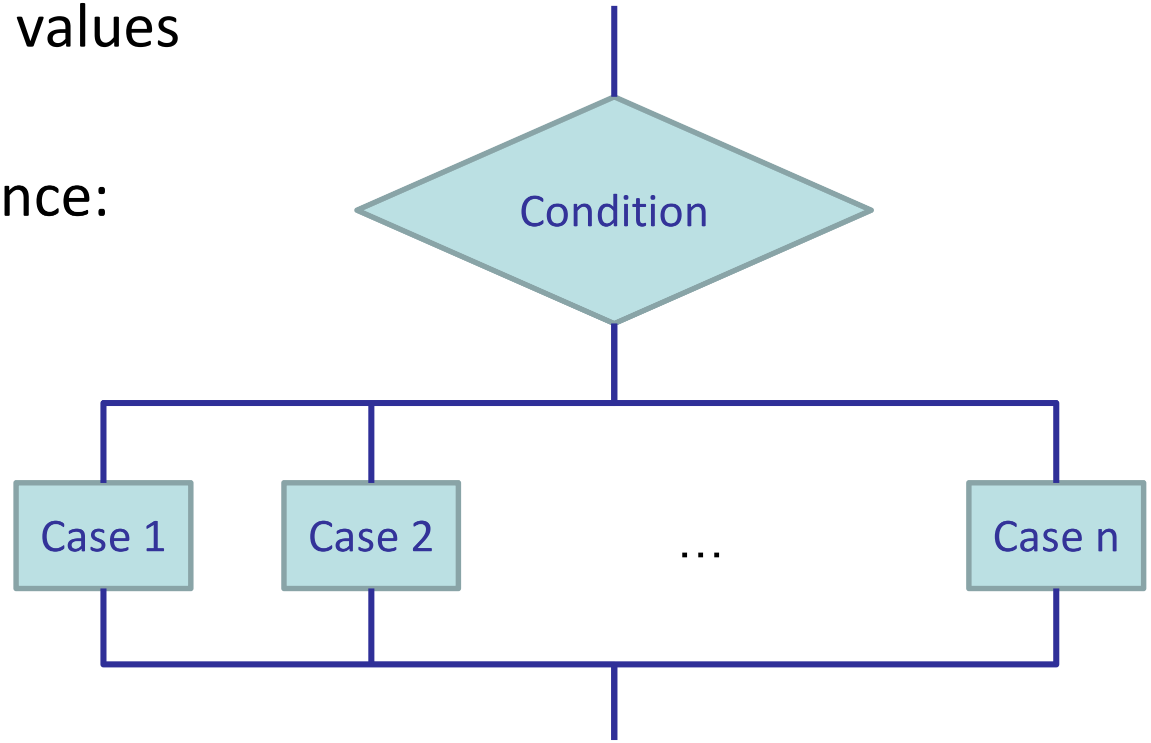
- Layout for `if` cascade: consistent indentation is abandoned to limit the depth of indentation

```
if (month == 1)
  days = 31;
else if (month == 2)
  days = 28;
else if (month == 3)
  days = 31;
else if (month == 4)
  days = 30;
. . .
else if (month == 12)
  days = 31;
```

`switch case` statements (1)

- A `switch case` statement is basically a simple type of `if` cascade in which the result of a particular expression is compared to a list of fixed values

- Program Sequence:



Based on: Habelitz (2012): Programmieren lernen mit Java

`switch case` statements (2)

- Syntax:

```
switch (expression) {
 case label:
      statement(s);
 case label:
     statement(s);
   . . .
}
```

- Processing:
  1. The expression is calculated (only once)
  2. The result is searched for under the `case` labels
  3. Statements in the appropriate branch are executed until a `break` statement is reached
- If no `case` label matches: no effect

switch case statements (3)

- Example: converting months into days

```
switch (month) {
 case 1:
  days = 31;
     break;
 case 2:
  days = 28;
    break;
 case 3:
    . . .
 case 12:
  days = 31;
     break;
}
```

`switch case` statements (4)

- case labels
  - Must be unique
  - Must all be calculable by the compiler
  - Only constant expressions allowed, which consist of `final` variables and literals
- Multiple `case` labels allowed per branch:

```
switch (month) {
 case 1:
 case 3:
 case 5:
 case 7:
 case 8:
 case 10:
 case 12:
   days = 31;
      break;
 ...
}
```

switch case statements (5)

- Default branch
  - The label default will be executed if no other label is applicable
  - May only occur once
  - Must be the last label
  - Corresponds to a final else in the if cascade
  - Always useful!

```
switch (month) {
 case 2:
   days = 28;
      break;

 case 4:
 case 6:
 case 9:
 case 11:
   days = 30;
      break;

 default:  //all remaining months
   days = 31;
      break;
}
```

Exercise – `switch case` statements

- Live exercise
  - Complete Task 4 on the live exercises sheet “Control structures”
  - You have 8 minutes.

switch case statements (6)

- Fall through
  - Behaviour in case of a missing break statement at the end of a branch
  - Rarely useful

```
switch (n) {
 case 1:
   System.out.println("one");
      //no break - fall through
 case 2:
   System.out.println("two");
   //no break - fall through
 case 3:
   System.out.println("three");
   //no break - fall through
}
```

Chapter 3: Control structures

3.1 Statement series (sequence)

3.2 Selection structures (selections)

3.3 Repeating structures (loops or iterations)

3.4 Effects on variables

Repeating structures

- Instruments for programme control
  - Certain statement blocks are not only executed once, but executed several times
  - Decision about repetition depends on a condition
    - Pre-test (top-controlled) loops
      (statements may possibly be skipped and not executed at all)
    - Post-test (bottom-controlled) loops
      (statements are executed at least once)

`while` loop (1)

- Java syntax with example:

```
while (condition) {
    statement(series);
}
```

Top of loop

Bottom of loop

- `boolean` expression condition controls the flow:

1. Evaluate condition
2. If `true`:
   a) Execute statement (or statement series)
   b) Back to 1

   If `false`: loop ends

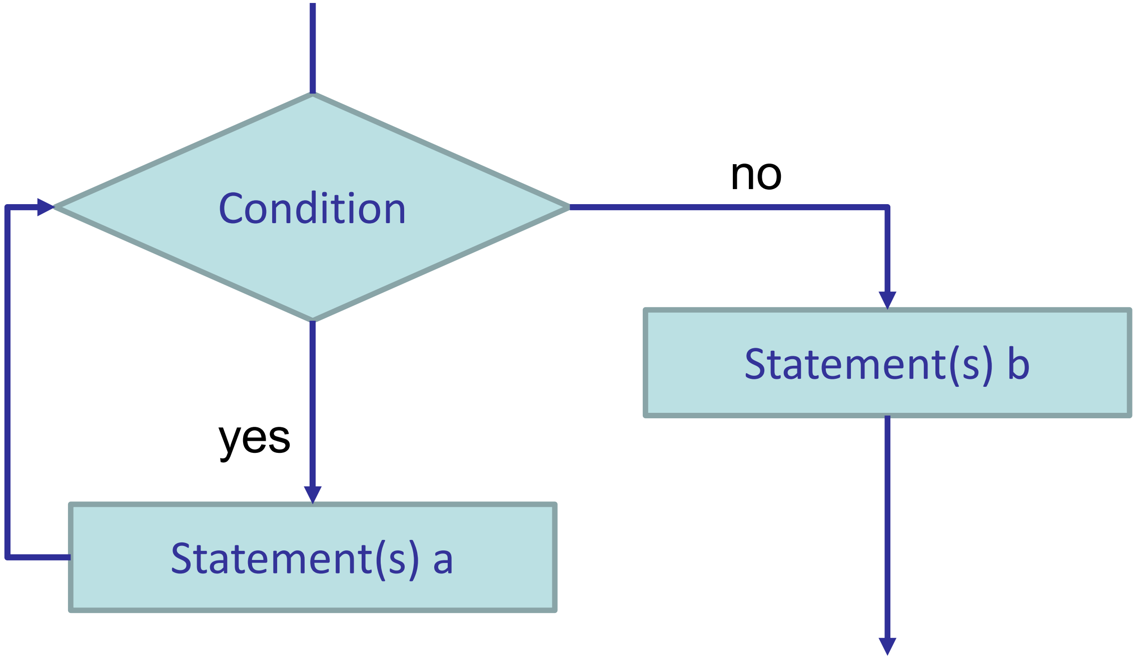
`while` loop (2)

- Example:

```
int counter = 0;

while (counter < 10) {
  System.out.println(counter);
    counter = counter + 1;
}
```

- Programme Sequence:



Based on: Habelitz (2012): Programmieren lernen mit Java

while loop (3)

- Alternatives to formulating count-controlled (i.e. for) loops:
  - Start with 0 or 1
  - Test the end value with < or <=
- Common practice: start with 0, test with <
  - End value = number of loop iterations

- Frequently used: short form for value assignments (operator assignment)

```
x = x + 2;      is equivalent to      x += 2;
```

```
x = x * 7;      is equivalent to     x *= 7;
```

  - …. further binary operators

Exercise – `while` loop

- Live exercise
  - Complete Task 5 on the live exercises sheet “Control structures”
  - You have 8 minutes.

Increment and decrement operators (1)

- Loop variables are often counted up or down in increments of one
  - Initial situation:

```
variable = variable + 1;
variable = variable -1;
```

  - With operator assignments:

```
variable += 1;
variable -= 1;
```

  - With increment operator ++ (decrement operator --)

```
variable++;
variable--;
```

Increment and decrement operators (2)

- Java distinguishes between two types:

1. Prefix operators

```
++variable;
--variable;
```

✓ Statement changes variable

✓ Expression: new value of the variable

```
int a = 1;
System.out.println(++a);   // a = 2, then output 2
System.out.println(++a);   // a = 3, then output 3
System.out.println(a);    //output 3
```

2. Postfix operators

```
variable++;
variable--;
```

✓ Statement changes variable

✓ Expression: old value of the variable

```
int a = 1;
System.out.println(a++);  // output 1, then a = 2
System.out.println(a++);  // output 2, then a = 3
System.out.println(a);    // output 3
```

Nested loops

- Nested loops: `while` loop is itself a statement => can be in the body of another loop

- Schematic representation:

```
while (condition) {           //outer loop
  while (condition) {       //inner loop
        statement(series);
    }
}
```

Exercise – Nested `while` loop

- Live exercise
  - Complete Task 6 on the live exercises sheet “Control structures”
  - You have 8 minutes.

`do` loop (1)

- Syntax:

```
do {
     statement(series);
} while (condition);
```

- Method of operation:
  1. Execute statement
     (or statement series)
  2. Evaluate condition
  3. If `true`: back to 1

     If `false`: loop ended

ONLY FOR COMPARISON:

```
while (condition) {
     statement(series);
}
```

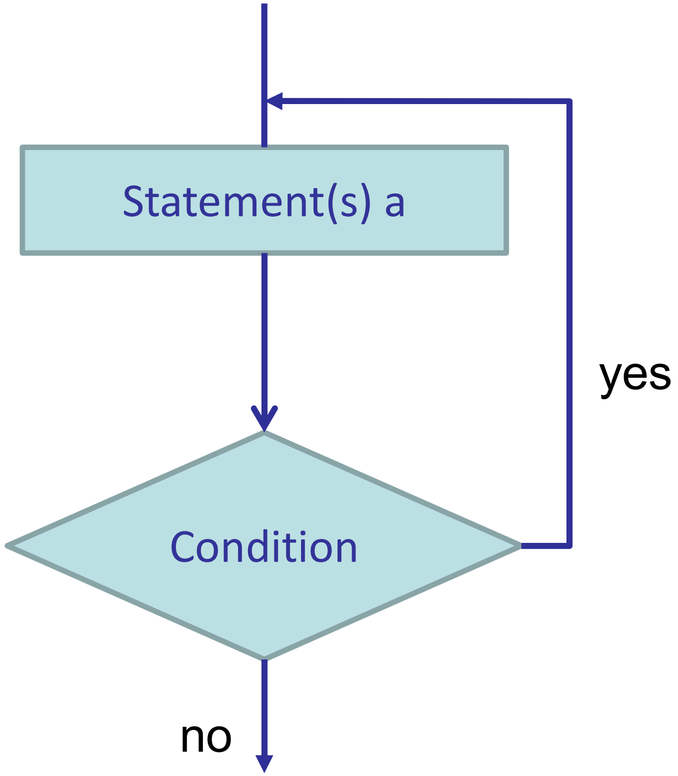
do loop (2)

Example:

```
int number = 0;

do {
   System.out.println(number);
      number++;
    } while (number < 100);
```

Programme Sequence:



Based on: Habelitz (2012): Programmieren lernen mit Java

`for` loop (1)

- `for` loops = linguistic devices for counting loops
- Syntax:

```
for (start; condition; update) {
    statement(series);
}
```

- Method of operation:
  1. Execute “Start” statement
  2. As long as the condition is met
     a) Execute statement (or statement series)
     b) Execute “update” statement
- Programme sequence same as `while` loop, no distinction

`for` loop (2)

- Example:

```
for (int i = 0; i < 10; i++) {
      System.out.println(i);
}
```

is equivalent to

```
int i = 0;

while (i < 10) {
  System.out.println(i);
    i++;
}
```

- `for` loop is more compact than `while` loop

Exercise – `for` loop

- Live exercise
  - Complete Task 7 on the live exercises sheet "Control structures"
  - You have 8 minutes.

`for` loop vs. `while` loop

- `for` and `while` loops can replace each other

```
for (start; condition; update) {
    statement(series);
}
```

```
{
  start;
   while (condition) {
         statement(series);
         update;
    }
}
```

- Validity range of the counter variables of `for` loops: header and body => consecutive `for` loops with the same control variables allowed

```
for (int i= 0; i < 10; i++) {
    System.out.println(i);
}
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

Selection of loop variant

- When `for`, when `while`?
- Criteria:
  - Number of loop iterations known beforehand => `for`
  - Single loop variable, simple counting up or down (increment, decrement, etc.) => `for`
  - Initialisation, test, count up or down fit in one line => `for`
  - Otherwise : `while`

jump statements (1)

- Should not be used in well-structured programmes
- However, there are situations in which the source code can be simplified and/or structured more clearly

- Loop termination with `break`

```
while (true) {
  int n;
    // determine n …
  if (n == 0)
         break;
    //process n
}
```

Leaving an **endless loop** with `break`

jump statements (2)

- Skipping statement(s) with `continue`

```
for (int i = 1; i <= 10; i++) {
   if (i == 5) {
     continue;
       }
   System.out.println(i);
}
```

What do the following do?

- `break` statement
  - Terminates a `switch`, `while`, `do` or `for` statement, that immediately surrounds the `break` statement

- `continue` statement
  - Interrupts the current loop iteration of a `while` , `do` or `for` loop and jumps to the repeat condition of the immediately surrounding loop

Chapter 3: Control structures

3.1 Statement series (sequence)

3.2 Selection structures (selections)

3.3 Repeating structures (loops or iterations)

3.4 Effects on variables

Validity ranges - scope (1)

- Validity range (i.e. scope) of a variable
  - starts with the definition (declaration)
  - ends with the block containing the definition
- Outside the scope: variable does not exist anymore
- Compiler checks validity ranges

Validity ranges - scope (2)

- Example:

```
public class Summation {
  public static void main(String[] args) {
        int n = 100;
        int sum = 0;
        int z = 0;
        while (z < n) {
          int square = z * z;
            z++;
            sum = sum + square;
        }
        System.out.println(sum);
    }
}
```

Scope of the variable `square`

Scope of the variable `z`

Naming conflicts

```
public class Summation {
  public static void main(String[] args) {
        int n = 100;
        int sum = 0;
        int z = 0;
        while (z < n) {
            int n = z * z;
            z++;
            sum = sum + n;
        }
        System.out.println(sum);
    }
}
```

Duplicate local variable n;
Compiler Message:
Variable ‘n’ already defined in this scope

- ➢ Within the scope of a variable, there must be no other variable with the same name in the source code!

Lifetime

- Time interval in which the variable exists during the runtime of the programme
  - At variable definition (declaration) = creation of the variable
  - At the end of the block: destruction of the variable

```
public class Summation {
  public static void main(String[] args) {
        int n = 100;
        int sum = 0;
        int z = 0;
        while (z < n) {
          int square = z * z;
            z++;
            sum = sum + square;
        }
        System.out.println(sum);
    }
}
```

The variable `square` is repeatedly generated and also destroyed again during programme execution