(3) Java Introduction: Imperative Programming – Part 2

Nico Ludwig (@ersatzteilchen)

TOC

- Cited Literature:
  - Just Java, Peter van der Linden
  - Thinking in Java, Bruce Eckel

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Output to Console

- Neither user-input nor user-output is part of Java's language-core, instead this functionality must be used via dedicated variables.
- Writing messages to the console is a very important means to communicate!
  - We can output message to the console via the variable *System.out*. *System.out* is a *PrintStream*.
  - Like *String* variables, *System.out* provides a set of methods, but those are methods provided by the type *PrintStream*, not *String*!
    - In this case the methods we can call, allow to output data. The most important methods are *print()*/*println()*.

Good to know
When we talk about console output, the term stdout will be used quite often.

- For the time being we'll output *String*s to the console and format the output by concatenating value like so:

```
System.out.print("Hello there, You know about " + 42 + "?");
```

Terminal
```
NicosMBP:src nico$ java Program
Hello there, You know about 42?
```

  - The method *println()* works like *print()*, but it also puts a newline after the text was output to console:

```
System.out.println("Hello there, You know about " + 42 + "?");
```

Terminal
```
NicosMBP:src nico$ java Program
Hello there, You know about 42?
```

- *System.out* is usually buffered, but on most OS' *System.out* is configured to autoflush the buffer.
  - In other words: everything works as expected! (We do usually not need to *flush() System.out* explicitly.)

4

- *System.out* is of type *PrintStream*.
- If *System.out* (STDOUT) is buffered or not depends on the console on which it is attached (read: platform dependent). On most OS' the console is buffered.
- There also exists *System.err*, which also writes to the console (STDERR), but this one is always unbuffered. Writing to *System.err* is like writing to another "channel": if the standard output of a program is e.g. redirected to a file, important messages are still visible, if they are output to *System.stderr*.
- If a *PrintStream* is configured to autoflush, the buffer will be flushed, when *println()* is called, a byte[] is written or a newline is output as literal escape sequence ('\n') or as format specifier ("%n").

Input from Console – Part 1

- The way Java provides reading input from the console with its standard libraries is poor!
  - Basically, Java accepts console input via the variable *System.in*, but we have to do somewhat more to work with it comfortably.
- To make our lives easier with console input, we'll introduce and use the complex type *Scanner*.
  - The type *Scanner* provides a set of methods, which allow relatively comfortable reading from the console.
  - *Scanner* is defined in the package *java.util*, which we have to import in order to use *Scanner*.
- What is a package? And what does it mean to "import a package"?
  - Java organizes its complete type system in a set of packages and sub packages.
    - For the time being let's assume, that packages are organized like boxes within boxes, in which other types are organized.
    - The hierarchical idea behind packages appears on their path-like/segmented names, e.g. *java.util*, *java.util.concurrent* or *java.util.concurrent.locks*
  - We have already used a package, but didn't notice it: all complex types we have used up to now reside in the package *java.lang*.
    - *java.lang* is automatically imported into any java-file, we do not have to import it explicitly.
  - Packages need to be imported before the definition of the *Program* class!
    - The syntax "import *java.util.**" means, that all types of the package *java.util* will be imported.
    - Alternatively, we could have written "import *java.util.Scanner*" in order to import only *Scanner* from *java.util*.
  - Occasionally we'll leave the import statement for *java.util.Scanner* away.

Good to know
In opposite to C/C++' #includes, imports are real Java statements. – Mind that imports are terminated with a ';'.

```
// <Program.java>
import java.util.*; // Import the package java.util.

public class Program {
      // … use the types defined in java.util, e.g. Scanner
}
```

5

- Another way to read from the console is the type *java.io.Console*. An object of *java.io.Console* can be retrieved by calling *System.console()*. The problem with that *Console* object is that the Java application needs to be attached to a real console of the OS, and this is usually not the case, when starting Java applications from an IDE (then *System.console()* returns null).
  - There is an interesting feature: *java.io.Console.readPassword()* can read passwords from the STDIN. It means, that the input text is not echoed to the console. The read password is stored in a char[].

Input from Console – Part 2

- All right, now its time to use *Scanner* for input from the console.
  - Alas we have to mention some Java topics, before we have officially introduced them, esp. instantiation with the keyword new.

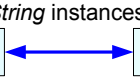
```
import java.util.*; // Import the package java.util to make the type Scanner available!

public class Program {
	public static void main(String[] args) {
		Scanner inputScanner = new Scanner(System.in); // Wrap System.in into a new Scanner.
		String theName = inputScanner.nextLine();		// Read a line of text that was entered by the user.
		int theAge = inputScanner.nextInt();			// Read an int that was entered by the user.
		inputScanner.close(); // Close the Scanner, because we're done with it!
	}
}
```

- In opposite to primitive types, values of complex types such as *Scanner*, have to be created with the keyword new.
  - The syntax creating *Scanner*, "new *Scanner(System.in)*", allows passing an *InputStream* value like *System.in*.
  - Creating values with new is sometimes called "instantiation", and *inputScanner* can also be called an instance of *Scanner*.
  - The wording isn't strict, we can also say *inputScanner* is a variable of type *Scanner*, on the other hand, *theAge* is an instance of int.
  - When *System.in* is passed to a new *Scanner*, this *Scanner* uses *System.in*, but provides a set of other methods to the programmer.
  - We're going to use a *Scanner*, which internally uses *System.in*. → Programmers say, that the *Scanner* instance wraps *System.in*.

- *String* is also a complex type, but we can create *String* instances with literals instead of using new.

```
// Creating a String value with a literal:
String aString = "Arthur";
```



```
// Creating a String value with new:
String aString = new String("Arthur");
```

Input from Console – Part 3

- OK, let's continue with *Scanner*. After we have instantiated the *Scanner* we can can start reading input from user.
  - Hence, we'll write import statements on the top of snippets in this course, if required.

```
import java.util.*; // Import the package java.util to make the type Scanner available!

Scanner inputScanner = new Scanner(System.in); // Wrap System.in into a new Scanner.
```

  - The first action with the *inputScanner* is to call the method *nextLine()*:

```
String theName = inputScanner.nextLine();    // Read a line of text that was entered by the user.
```

  - *nextLine()* will pause the program execution, until the user enters something on the console and hits enter.
  - *nextLine()* will then convert the user input to a *String* and returns this *String* as result of its call.

- The next action is to call the method *nextInt()*:

```
int theAge = inputScanner.nextInt();      // Read an int that was entered by the user.
```

```
// The "<" notation will be used in
// comments to highlight (user) input:
int theAge = inputScanner.nextInt();
// <37
```

Terminal

```
NicosMBP:src nico$ java Program
37
```

  - *nextInt()* works like *nextLine()*, but it converts the input data to an int and returns this int as result.
  - If the user didn't input an int, but e.g. a name, while *nextInt()* is awaiting an int, the call will fail and the program will be terminated!
    - We'll discuss a remedy in a minute! A name is a *String* and can not be converted into an int of course.
  - When we are done with the *Scanner* instance, we have to close it! – We have to call *inputScanner*'s method *close()*:

```
inputScanner.close();                    // Close the Scanner, because we're done with it!
```

- So far the basic introduction of the type *Scanner*. We'll learn how to use it as we go on with the examples.

- *If Scanner.nextInt()* fails, a *java.util.InputMismatchException* will be thrown. – If unhandled, the program will be terminated.

Checking User Input from Console – Part 1

- Let's inspect following snippet, when we run it in a program it looks like so on the console:

```
int yourAge = inputScanner.nextInt();
// ...
```

Terminal

```
NicosMBP:src nico$ java Program
```

  - Hm … the program pauses execution until input has ended and enter has been hit.
  - The point is: how should the user know, that she should enter an int? – Maybe the program did even stop working?

- When the user is meant to input data for processing, the program should prompt!
  - Therefor we just add an output, which explains, what the user should input:

Good to know
Prompt from latin *promptare*: "to publish something".

```
System.out.println("Please enter your age:"); // prompt
int yourAge = inputScanner.nextInt();
// ...
```

Terminal

```
NicosMBP:src nico$ java Program
NicosMBP:src nico$ Please enter your age:
```

- So far so good. There is another problem with input validity. E.g. in this case, the entered age must be greater or equal 0.
  - With conditional code we can handle this situation easily:

```
System.out.println("Please enter your age:");
int yourAge = inputScanner.nextInt();
if (yourAge < 0) { // Check the input value to meet our expectations.
      System.out.println("Your age mustn't be less than 0!"); // another prompt
}
```

Terminal

```
NicosMBP:src nico$ java Program
NicosMBP:src nico$ Please enter your age:
-38
Your age mustn't be less than 0!
NicosMBP:src nico$
```

Good to know
When a program prints to the output, what a user has entered, this is called echo. (Here, the console echos the entered -38 automatically.)

  - Checking user input is very important! Usually we cannot trust user input! It's the most probable source of bugs in programming!
  - The example is not perfect, the program should be able to repeat the prompt, until the user's input is valid. We'll handle this soon!

- Formally, we can tell the application prompt from the command prompt. The command prompt is the prompt of the system console or terminal. On most OSes, the command prompt has a special syntax, e.g. the name or path of the current directory and some special character ($, # or >), which precedes the cursor ready for input.

Checking User Input from Console – Part 2

- However, there is still a potential source of problems: What if the user enters a text, although a number (int) is awaited?

```
// Remember our last example:
System.out.println("Please enter your age:");
int yourAge = inputScanner.nextInt();
if (yourAge < 0) {
        System.out.println("Your age mustn't be less than 0!");
}
```

Terminal

```
NicosMBP:src nico$ java Program
Please enter your age:
Nico
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:864)
    at java.base/java.util.Scanner.next(Scanner.java:1485)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2117)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2076)
    at Program.main(Program.java:18)
NicosMBP:src nico$
```

  - In this case a so-called *Exception* is thrown by the Java interpreter. *Exception*s indicate run time error conditions in Java.

- *Scanner* provides integrated checks, for entered data to be of unexpected type, e.g. text instead of a number for age.
  - Because we didn't discuss *Exception* handling up to now, we'll apply *Scanner*'s methods to pre-check user input before retrieval:

```
System.out.println("Please enter your age:");
// Pre-check input: did the user input an int?
if (!inputScanner.hasNextInt()) {
        System.out.println("Your age must be a valid number!");
        inputScanner.next(); // Ignore the invalid user input.
}
```

Terminal

```
NicosMBP:src nico$ java Program
Please enter your age:
Nico
Your age must be a valid number!
NicosMBP:src nico$
```

  - *Scanner* provides a score of methods to check the type of user data in the input buffer (*hasNextInt()*, *hasNextDouble()* etc.).
  - It is required to clear the *Scanner*'s input buffer (e.g. with *next()*) to accept other, hopefully correct input from the user.

This is an important Lesson: Usually we cannot trust user input – Part 1

- When a user has to enter data, there is a certain probability that the input data is wrong! Therefor we should:
  - (1) write programs, which prompt an output, before it requires input from the user.
    - The text of the prompt should explain to the user in simple words, which input and in which format it is required.
    - (Then the following input operation blocks program execution and forces user interaction.)
  - (2) then check the user's input for correctness.
    - Forgetting to check user input is a major source of bugs!

- Usually, programs should behave highly interactive and give the user a chance to:
  - (1) Enter data, until it is valid,
  - (2) or, allow the user to quit data input.
  - ((3) And maybe echo the entered data and request further confirmation from the user.)
  - But to implement programs like this, we have to discuss more means of program control flow.

- So: good quality programs have to prompt for user input and check user input.
  - And may optionally echo user input.

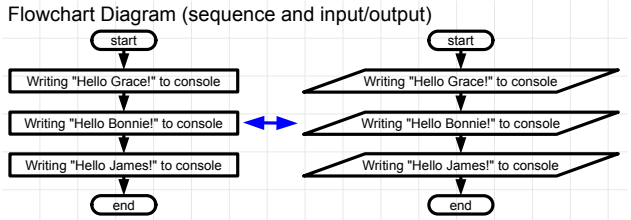
- The discipline to check input before continuing in the program is called defensive programming.
  - Defensive programming is a cornerstone of high quality programs!

This is an important Lesson: Usually we cannot trust user input – Part 2

- Usually checking input is done on multiple levels, usually we tell apart basic checks from checking business rules.
  - A basic check would be "Did the user enter a number?".
  - A business rule would be "The user should enter the age, which is a positive number. – Did the user enter a positive number?"
    - Business rules could also apply blocklists and allowlists to check inputs.
  - If input passes basic checks and business rules checks, the input is said to be plausible.

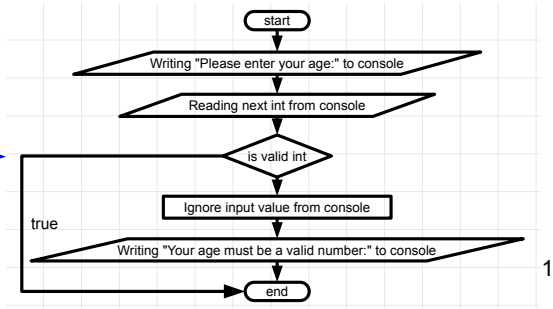
Input and Output in a Flowchart Diagram

- We've special parallelogram-shaped flowchart symbols for sequence items, which deal with input and output.



- So we can redesign our last example with this new Flowchart element:

```
System.out.println("Please enter your age:");
// Pre-check input: did the user input an int?
if (!inputScanner.hasNextInt()) {
        System.out.println("Your age must be a valid number:");
        inputScanner.next(); // Ignore the invalid user input.
}
```



Operators – Part 1 – Mathematical Operators

- The concept of arithmetic operators is well known from mathematics: "+", "-" and "*" (multiplication), "/" (division).

```
int difference = 5 - 3;
```

Good to know
The operator symbol '-' is sometimes called dash or hyphen, although those characters are different.

- Some words on integral division:
  - If in a division operation only ints are involved, the division will yield an int result and no fractional number!

```
double result = 2 / 3;
```

result (0.0)

  - Actually, *result* evaluates to 0.0!
  - To correct the expression, so that the division yields a fractional result, at least one of the operands needs to be of floaty type.
    - (1) If the division uses int literals one of the literals can be just be written as, e.g., double literal:

```
double result = 2.0 / 3;
```

result (0.6666666666666666)

    - (2) One of the contributed operands (an int literal, int variable or other int expression) can be cast to a double:

```
double result = (double)2 / 3;
```

result (0.6666666666666666)

  - (1) and (2) will evaluate to *result* having the value $0.\overline{6}$.
  - Notice: the result of an expression right from assignment is of the type of the largest and floatiest operands' type.

- We already discussed, that the result of the division by 0 will fail at run time with an *ArithmeticException*.

- What does that mean "integral division has an integer result"?
  - I.e. their results are no floating point values!
- The division by 0 is not "not allowed" in maths, instead it is "just" undefined.

Operators – Part 2 – Mathematical Operators

- "-" and "+" can be used as so called "unary" operators, to mark their operand as negative or positive value:

```
double amount = 4;
double negativeAmount = -amount; // makes negativeAmount the negative value of amount
```

negativeAmount (-4.0)

- Java provides support for division with remainder with the operator "%", which is called modulus operator.
  - "%" calculates the symmetric modulus (in difference to the mathematical modulus) of its operands.

```
int result1 = 5 % 2;
```

result1 (1)

```
int result2 = -5 % 2;
```

result2 (-1)

  - Remainder calculations are very handy, e.g. to program loops doing something at every x^{th} loop.
  - Modulus also works with floaty values in Java!

- Special functions, like exponentiation, log and trigonometric functions are not represented by operators in Java.
  - Basically all important functions can be found as methods in the class *Math*. (We already mentioned *Math*'s *Math.PI*.)

```
double result1 = Math.pow(10, 2);
```

result1 ($10^2 = 100$)

```
double result2 = Math.log10(100);
```

result2 ($\log_{10} 100 = 2$)

  - The arguments of trigonometric functions need to be passed in the unit radiant (rad) of degrees (°) of an arc!
    - 360° = 2π rad => 180° = π rad => 1 rad = 180°/π => 90° = π/2 rad

```
double result3 = Math.sin(Math.PI / 2);
```

result3 (sin(90°) = sin(π/2) = 1)

  - *Math* is a kind of library of math-related methods.

Operators – Part 3 – Operator Notation – Arity

- Binary operators (+, -, *, /, %, ==, <, >, &, |, &&, || etc.) have two operands:

```
// Addition as binary operator:
int sum = 2 + 3;
```

sum (5)

  - Usually, binary operators are written with whitespaces around operands and operator.
  - The operands of binary operators are colloquially called left hand side (lhs), i.e. 2, and right hand side (rhs), i.e. 3.

- Unary operators (-, +, ++, --, ! etc.) have one operand:

```
// Minus as unary operator:
int i = 4;
int j = -i; // makes j the negative value of i
```

j (-4)

  - Usually, unary operators are written without whitespaces between operand and operator.

- The ternary operator has three operands:

```
// The conditional operator is the only ternary operator:
int i = 2;
int j = 3;
String answer = (i < j) ? "i less than j" : "i not less than j";
```

answer ("i less than j")

  - There are no specific conventions on the notation.
    - It is recommendable (not mandatory) to write the condition in parentheses.

Good to know
The ternary operator ?: is sometimes called Elvis-operator, because is resembles Elvis Presley's iconic pompadour haircut.
(In a narrower sense, ?: is called Elvis-operator, if it is used as binary operator, a notation, which is not supported in Java.)

Operators – Part 4 – Operator Notation – Placement

- Prefix operators are written in front of its only argument:

```
// Negation as prefix operator:
int negatedValue = -value;
```

  - This notation is also called (forward) polish notation, or PN.

- Postfix operators are written after its only argument:

```
// Increment as postfix operator:
int result = item++;
```

  - This notation is also called reversed polish notation, or RPN.

- Infix operators are written in between its arguments:

```
// Addition as infix operator:
int sum = 2 + 3;
```

Good to know
The polish notation was developed by the Polish mathematician Jan Łukasiewicz because, he could formulate his mathematical theories in a better manner.
Originally, the notation was called Łukasiewicz notation, but nobody could pronounce or write his name correctly, so it was called polish notation.
(Actually Łukasiewicz is pronounced as [wukaˈɛvit͡ʂ].)
Generally, polish notation is the name of all notations different from infix notation.
There exist programming languages, which only apply PN, e.g., Lisp.
Some pocket calculators (e.g. the legendary Hewlett-Packard HP-12C) use RPN, instead of algebraic notation. The advantage of RPN is, that neither operator priorities nor parentheses required.

- The []-operator (brackets) awaits its only argument in between the brackets.
  - This operator is used to access array elements. We'll discuss arrays in a future lecture.

- Additionally, parentheses (parens), i.e. (), also await their only argument in between the parentheses.
  - Parens are used to structure expressions and force prioritized evaluation.

Operators – Part 5 – Operator Notation – Placement (other Languages)

- Esp. mathematical operations are usually expressed as a set of operators and operands.
- The interesting point is, that programing languages in the wild use all thinkable operator placements.
- Postfix operator placement/RPN means, that a list of operands is written in front of an operator.
  - Assume the addition of two integers in the programming language PostScript:

```
% PostScript - Addition:
/sum 2 3 add def
```

  - *sum* => variable name, 2 and 3 => operands, add => operator, def => keyword (define variable)
  - An iconic postfix operator in Java is the postfix increment:

```
// Java - Postfix increment:
i++;
```

- Prefix operator placement/PN means, that an operator is written in front of a list of operators.
  - PN is used for mathematical functions like f(5) or g(8, 2.59) or predefined mathematical operations like cos(90), sin(18) and tan(35).
  - Assume the addition of two integers in the programming language Lisp:

```
; Lisp - Addition and init of variable sum:
(let ((sum (+ 2 3))))
```

  - This is Java's notation for method calls:

```
// Java - Method call:
double result = Math.pow(2, 3);
```

Operators – Part 6 – Operator Notation – Placement (other Languages)

- The advantages of RPN/PN:
  - Flexibility: In many languages and situations multiple or lists of operands can be specified.
  - Neither operator priorities nor parentheses required.
  - E.g. Java allows defining methods, which accept a list of operands (also called method arguments).

- Infix operator placement means, that an operator is written in between two operators.
  - This notation is used for all the elementary mathematical operations like addition or multiplication.
  - But, it is the most limited notation: Infix notation is only available written binary, i.e. with exactly two operands.

```
// Java - Addition:
int result = 2 + 3;
```

- In reality, we'll mainly find combinations of all notation variants to write meaningful code:

```
// Java - Addition and prefix method call:
double result = Math.pow(7, 2 + 3);
```

- There exist three notations, prefix, infix and postfix and parentheses (parentheses, brackets and angle brackets).

Operators – Part 7 – Logical Operators

- When we discussed conditional code, we encountered snippets similar to this:

```
int age = 90;
int countOfGreetingsForBonnie = 0;
// Using cascaded if statements:
if (age > 80) {
        if (countOfGreetingsForBonnie <= 0) {
                System.out.println("Hello Bonnie!");
        }
}
```

  - The condition, under which "Hello Bonnie" is written to the console can be summarized to:
    - "Hello Bonnie" is written to the console, if *age* is greater than 80 and *countOfGreetingsForBonnie* is less than or equal to 0.
    - These two cascaded if statements do reflect an logical and-combination of two conditions, i.e. boolean expressions.

- Logical operators can be applied in boolean expressions for control structures (conditional code and loops).
  - In Java we can use following logical operators: && (logical "and"), || (logical "or"), ! (logical "not").
  - Using && we can reformulate and condense the code with the cascaded if statements shown above to only one if statement:

```
int age = 90;
int countOfGreetingsForBonnie = 0;
// Using logical and (&&):
if (age > 80 && countOfGreetingsForBonnie <= 0) {
        System.out.println("Hello Bonnie!");
}
```

  - Java's logical operators evaluate to boolean results, which are no integral values in Java.

Operators – Part 8 – Logical Operators – Comparison

- Besides logical operators, we have already used Java's comparison operators to compare values:
  - Comparison operators: ==, !=, <, >, <=, >=

- "=" and "==" are different operators!
  - "=" for assignment and "==" for equality comparison: i.e. "==" evaluates to a boolean value!

```
if (a == 0) { // Is a's value zero?
        // pass
}
```

  - Using "=" instead of "==" for equality comparison will end in a compile time error:

```
// A typical beginner's error: the "optical illusion":
if (a = 0) { // (1) Invalid! Oops! a == 0 was meant!
        // pass
}
```

Good to know
In maths, the expression $a = 0$ can mean "0 is assigned to a", or "it is asserted, that a is 0". But maths also provides dedicated notations to distinguish both (very different) meanings:
$a := 0$ "a is defined and set to the value 0"
-> Java's assignment
$a = 0$ "it is asserted, that a is 0"
-> Java's equality comparison

- Remember: we cannot use these operators to compare *Strings*! *String* comparison must be done with *equals()*:

```
// Correct way to compare two Strings:
String herName = "Gwen";
if (herName.equals("Gwen")) {
        // pass
}
```

  - The crux: using "==" to compare *String*s won't end in a compile time error, but potentially in wrong results at run time!

20

- JavaScript additionally defines the comparison operator ===. It performs a strict comparison of values: it evaluates to true, if value and type of lhs and rhs are equal, else it evaluates to false. (The comparison operator == is less strict, the types of the operands need not to match.) E.g. 1 === 1 will evaluate to true, whereas "1" === 1 will evaluated to false.

Operators – Part 9 – Logical Operators – Negation

- In Java there exist two negation operators "!" and "!=", which are used to express inequality in boolean expressions:

```
// Using "!=" for inequality comparison:
if (count != 0) { // If count is unequals 0:
        // pass
}
```

Good to know
The ! (exclamation mark or exclamation point) is sometimes also called "bang" or "shriek" among developers.

  - We can also formulate this comparison with the unary negation operator.

```
// Using "!" for inequality comparison:
If (!(count == 0)) { // If count is unequals 0:
        // pass
}
```

  - This is an example for the need for parens: we've to put the comparison expression in parens to make ! effective on it as a whole.
  - Using ! in this case states "if count == 0 is not true" ...

- Because we cannot use == to compare *String* for equality we have to use "!" and *equals()* to compare them for inequality:

```
// Correct way to compare two Strings for inequality:
String herName = "Peter";
if (!herName.equals("Gwen")) {
        // pass
}
```

Operators – Part 10 – Logical Operators – Short-Circuit Evaluation

- The binary logical operators "&&" and "||" support short-circuit evaluation to improve performance:
  - Let's think about following snippet:

```
int age = 70;
int countOfGreetingsForBonnie = 0;
if (age > 80 && countOfGreetingsForBonnie <= 0) {
        System.out.println("Hello Bonnie!");
}
```

  - Short-circuiting: if the lhs of "&&" evaluates to false, the rhs need not to be evaluated at all, because the overall result is false!
  - *age* > 80 evaluates to false, *countOfGreetingsForBonnie* <= 0 needs not to be evaluated, because the result of "&&" must be false.
  - "&&": If the lhs evaluates to false, the rhs need not to be evaluated, because the result of the whole expression must be false!
  - "&&": If the lhs evaluates to true, the evaluated result of rhs is the result of the whole expression!

- For the operator "||" there exists a similar short-circuit evaluation rule:
  - "||": If the lhs evaluates to true, the rhs need not to be evaluated, because the result of the whole expression must be true!
  - "||": If the lhs evaluates to false, the evaluated result of rhs is the result of the whole expression!

- The idea is to place a more expensive operation into a short-circuit expression, so that it might not necessarily be evaluated!
  - => The more expensive operation could be the rhs for "&&" and "||". It depends on the probability of the lhs leading to short-circuit.

Operators – Part 11 – Non-Shortcut logical Operations

- So the operators && and || to express logical operations, which provide short-circuit evaluation.
- Sometimes, we need logical operations, which always evaluate both sides, e.g. the exclusive or (xor) operation.
  - Xor accepts two boolean arguments. It only evaluates to true, if both arguments have different values.
  - We can write down following truth table for xor:

	true	true	false	false
⊻	true	false	false	true
	false	true	false	true

- In Java we express xor with the ^ symbol:

```
boolean x = true;
boolean y = false;
boolean result = x ^ y;
System.out.println(result);
// >true
```

Good to know
The symbol '^' is called caret or hat.

- Besides xor, Java also provides non-short-circuit variants of the operators && and ||, namely & and |:

```
boolean a() {
        System.out.println("in a());
        return true;
}

boolean b() {
        System.out.println("in b());
        return false;
}
```

```
boolean result1 = a() & b();
System.out.println(result);
boolean result2 = a() | b();
System.out.println(result);
```

Terminal

```
NicosMBP:src nico$ java Program
in a()
in b()
false
in a()
in b()
true
NicosMBP:src nico$
```

  - When using & or |, both arguments are always evaluated, i.e. their side effects appear, but the result is the same as if we used && or ||.

- There exist different notations of the xor operator in mathematics.

Operators – Part 12 – Summary of Logical Operators

- Used to compare values and combine boolean results.
  - Comparison: ==, !=, <, >, <=, >= and instanceof
  - Combination: &&, || and !
  - Logical operators return results of type boolean.
- && (logical and) and || (logical or) support short circuit evaluation.

```
// The mathematic boolean expression a = b and c = b:

if (a == b && c == b) // Ok
{ // pass
}
if (a && b == c) // Wrong!
{ // pass
}
```


- On many C/C++ compilers the shown "optical illusion" leads to a warning message. Some compilers (e.g. gcc) accept "marking" the assignment to be meant as condition by writing them into an extra pair of parentheses: if ((a = 0)) { /* pass */ }.
  - In Groovy this way of marking is required.
  - In Swift assignment expressions don't return a value at all. With this simple rule a Swift expression like if a = 0 { /* pass */ } is a syntax error!

Operators – Part 13 – Bit-Wise Operators

- Java's bit-wise operators have no direct representation in mathematics.
  - Those operators work on integral values and operate on their bit representation (the bit-pattern in computer memory) directly.
  - Basically, bit-wise operators have nothing to do with logical operators.

Good to know
The symbol '&' is usually called ampersand. Ampersand is a corrupted form of the phrase "and per se and", which was used in past to name the '&' itself as specific symbol, e.g. in an enumeration.

- E.g. let's quickly talk about the bit-and operator, which is expressed with the symbol "&":
  - (1) Bit-and combines each bit-value of each bit of the two operands with a logical and operation.
  - (2) Then the resulting bit-pattern is interpreted as integral value.
  - => Logical operators (e.g. && and ||) have a boolean result, bit-wise operators (& and |) have integral bit-pattern results

```
int x = 65535;
int y = x & 255;
System.out.println(y);
// >255
```

	1111 1111	1111 1111	*x* (65535)
&	0000 0000	1111 1111	mask (255)
	0000 0000	1111 1111	*y* (255)

- We'll not discuss bitwise operators in great depth in this Java course, but it is an important topic in the C++ course.

- Besides bit-and, there exist a lot of other bitwise operators: ^, |, ~, <<, >> and >>>.

Good to know
Usually developers call the operator | ("vertical bar") "pipe". Some fonts represent the pipe with the symbol ¦ ("broken bar" or "parted rule").
Often developers call the operators <, >, <<, >> and >>> "chevrons", angle brackets or pointy brackets.

- Why do we need bit-wise operators?
- Bit shifting operations (x << n and x >> n):
  - Shifts all bits to left or right by n positions, the "new" bits will be filled with 0s.
  - A single left shift is a multiplication by 2; a single right shift is a division by 2.
  - Whereas << and >> preserve the sign, the special bit shift operator >>> also shifts the sign (its called "signed right shift") and fills with 0s.

Operators – Part 14 – Precedence

- Let's discuss this snippet:

```
int result = 3 + 4 * 6;
// result = 27
```

  - However, how the result is effectively calculated, if many operators are involved is not obvious!
    - Esp. because we could have mixed expressions with prefix-, postfix- and infix-notated operators, the language must define some rules.
  - In this case the multiplication 4 * 6 is evaluated to 24 before the addition of 3 takes place.
    - We imply operator priority as we know it from maths (PEDMAS), i.e. the evaluation order is independent from their written order in the expression.
  - Because Java has more operators than we know from basic maths, there are more rules if many operators are involved.

- The priority, in which operators are evaluated is called precedence.
  - To simplify matters, several operators are summarized in 15 precedence groups respectively, in which operators have the same precedence.
  - The operators' precedences are mostly as we know from basic maths.

- Precedence is controllable with parentheses as in maths:

```
int result = (3 + 4) * 6;
// result = 42
```

  - If we want to cascade multiple parentheses, we must only use parentheses (round brackets) and no other form of bracket as we sometimes see in maths.

Precedence Group	Operators	Type
15	() []	Parentheses Array subscript ...
14	++ --	Unary post-increment Unary post-decrement
13	++ -- + - !	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation...
12	* / %	Multiplication Division Modulus
11 ...	+ - ...	Addition Subtraction ...

- The abbreviation **PEDMAS** helps remembering the mathematical rules of order in arithmetic evaluation: **P**arentheses, **E**xponents, **M**ultiplication and **D**ivision, **A**ddition and **S**ubtraction. In Java exponents are not represented with an idiomatic operator, but as function.

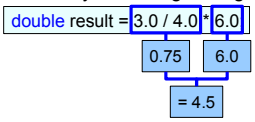
Operators – Part 15 – Associativity

- Having clarified the term precedence, what's about priority among operators of the same precedence group?

```
double result = 3.0 / 4.0 * 6.0;
```

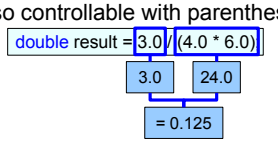
  - What is the result of this expression? Is it **(3.0 / 4.0)** * 6.0 = 4.5 or 3.0 / **(4.0 * 6.0)** = 0.125?

- Associativity defines evaluation among expressions of the same precedence.
  - The associativity of precedence groups defines the "direction", in which order expressions are evaluated.
  - Because "/" and "*" reside in precedence group 12, we've to apply the associativity "left to right" to get the result:



  - So, the result is 4.5!

- Associativity is also controllable with parentheses:



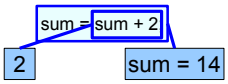
Precedence Group	Operators	Type	Associativity
15	() []	Parentheses Array subscript ...	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to Left
13	++ -- + - !	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation...	Right to Left
12	* / %	Multiplication Division Modulus	Left to Right
11 ...	+ - ...	Addition Subtraction ...	Left to Right

Operators – Part 16 – Combined Assignment

- In Java we can add sum up a variable by a value very easily:

```
int sum = 12;
sum = sum + 2; // Add and assign.
```

- The statement `sum = sum + 1;` looks weird, but by applying precedence/associativity, we understand what is going on here:



  - "=" has lower precedence than "+" and "=" is right-associative, i.e. "*sum + 2*" is evaluated first, which yields the value 14.
  - Finally, "=" assigns the result 14 to *sum*, which overwrites the 12 formally stored in *sum*.

- However, the syntax above has a little problem: the variable *sum* occurs twice:
  - (1) as summand in the addition operation, and
  - (2) as target for the assignment operation.
  - If this code is modified afterwards, one could exchange only one of two occurrences of *sum* and introduce an error.

- To make such operations more foolproof, Java offers combined assignment, that combines arithmetic and assignment operators:

```
sum = sum + 2;   →   sum += 2; // combined assignment and addition
```

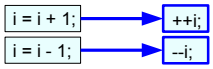
  - Java provides combined assignment for most arithmetic (e.g. -=, *= ...) and bit-wise (e.g. |=, &= ...) operators.
  - *String*s can be concatenated with the operator + and combined assignment and addition is also possible with +=.

Operators – Part 17 – Increment and Decrement

- Programming often requires to add 1 to the value of an integer.
  - We'll use this operation a lot, when we discuss loops to count things in a program.
  - The operation to add 1 to an integer is often called incrementation, subtracting 1 is called decrementation.
  - (There also exist terms like "incrementation of two", i.e. incrementation and decrementation are sometimes more general terms.)

```
// Increment as  combined assignment and addition:
int i = 1;
i = i + 1; // increments i by 1
// i = 2
```

- Java provides an even more condensed syntax as `i = i + 1;` for incrementation/decrementation of an int by 1:




```
// Increment as addition operation:
int i = 1;
++i; // increments i by 1
// i = 2
```

- We call the operators ++/-- increment/decrement operators.

- ++/-- can also be applied on variables of floaty type, but this is very uncommon.

Operators – Part 18 – Increment and Decrement

- Increment and decrement operator-expressions come in super handy, because they are very compact.

- However, there is a big difference to unary expressions we have discussed to now: "++" and "--" have side effects. – What?
  - Well, we can just write `++i;` as a single statement, which internally modifies *i*, i.e. the assignment to *i* is hidden!
  - Think: `++i;` is like `i = i + 1;`! – It is not like `i + 1`, because `++i` has a side effect!
  - If an expression or statement modifies variables, it is said to have a side effect on that variables.

```
i + 1;      // evaluates to the value of i + 1
++i;        // changes i to the value of i +1
```

- The increment and decrement operators come in two syntactical and semantical flavors: prefix and postfix.
  - As the wording implies, the syntax on how operator/operand placement is different:

```
++i; // prefix increment
```

```
i++; // postfix increment
```

  - The effective semantic difference lies in the value, which is the result of their expressions respectively:

```
int i = 1;
int result = ++i; // prefix increment
// i = 2
// result = 2
```

```
int i = 1;
int result = i++; // postfix increment
// i = 2
// result = 1
```

  - Both expressions ++*i* and *i*++ have the same side effect on *i*, after the expression was evaluated respectively.
  - But the expression results differ: *i*++ yields the value before the incrementation, ++*i* yields the value after the incrementation.
  - Notice: the side effect of prefix and postfix increment/decrement is the same, but the yielded results are slightly different.

Advice: ++i (pre-increment) should be your first choice, not i++ (post-increment), as it is shown in many books and school exercises! Only use i++, if its result (the value before incrementation) is needed and this is rarely the case!

Operators – Part 19 – Precedence, Associativity and Order of Evaluation

- Apart from precedence and associativity, there is still another aspect of operator evaluation: the order of evaluation.
  - Consider this snippet:

```
int i = 0;
int result = (i = 2) * i++;
```

  - In which order do the subexpressions in the expression "*(i = 2) * i++*" evaluate? What is the value of *result* and *i*?

```
// result = 0; i = 2
```

```
// result = 4; i = 3
```

  - Well, the correct answer is *result* = 4 and *i* = 3. – Why?

- Actually, neither precedence nor associativity can answer the question, because *i* is modified twice in a single expression!
  - Virtually, *i*++ has a higher precedence than *, but the left hand side of *, *i* = 2 is executed first? – But why?

- The answer is order of evaluation: If a variable is modified more than once in one expression keep these rules in mind:
  - The order of evaluation within expressions is strictly defined in Java.
  - The order is from left to right and from inner to outer.

- The order of evaluation is not strictly defined in C++, which is a serious source of bugs!

Operators – Part 20 – Precedence, Associativity and Order of Evaluation

- How do precedence, associativity and order of evaluation work together?

- It's sufficient to keep this in mind: precedence takes effect first, then associativity and then maybe order of evaluation.
  - Example: `a = b = c = 1 + 2 + 3;` $(a = (b = c)_C)_D = ((1 + 2)_A + 3)_B;$
    - "=" has a lower precedence than "+", so the associativity of "=" takes effect ("=" has the lowest precedence at all)
    - "=" is right associative
    - "+" is left associative, so A is executed before B
    - back to "=", which is right associative in (a = b = c), C is executed before D
  - The order of evaluation is only relevant, if a specific variable is modified more than once or methods with side effects are called.
    - Order of evaluation is always done from left to right and from inner to outer.

- Guidelines for associativity: unary operators, =-operators and ?: are right-associative, others left-associative.

- It makes no sense to learn precedence/associativity tables by heart.
  - If in doubt, just use parentheses to control precedence and associativity explicitly.
  - Parentheses are usually also used by experienced programmers, it has nothing to do with lack of expertise!
  - With parentheses we can not control order of execution.

32

- What is "order of execution"? Why is it relevant?
  - E.g. if we have an expression like *h(g(), f())* or *d*() * *s*() how can we know, which functions are being called first? This order of execution is strictly defined in Java! It is relevant to know that, because the function calls can have side effects!

Operators – Part 21 – Other Operators and Operator Overloading

- So far our journey through Java's operators

- Honorable mentions of operators we have used alongside:
  - Operator ?: – The ternary conditional operator.
  - Operator new – The operator to create instances of user defined types (UDTs).
  - The cast operator "()".

- There exist some more specialized operators, which we will discuss in future lectures:
  - Operator [] – Brackets to define and access arrays.
  - Operator instanceof – Operator to check the type of an instance.
  - Additionally Java defines some special symbols, which are no operators per se, to deal with methods:
    - The lambda operator: ->.
    - Separators: ( ) { } [ ] ; , ... @ ::

- Java does not permit to define new or redefine (overload) operators for UDTs!
  - (For UDTs, the method *toString()* can be overridden and the operator + will call *toString()* to convert an object to a *String*.)
    - (So, in a sense we can overload operator + for *String*-conversion by overriding *toString()*.)

Control Structures – Iteration

- In programming we often have to code the same series of statements over and over.
  - E.g. reading a couple of numbers from the console:

```
System.out.println("Please enter 1. number:");
int number1 = inputScanner.nextInt();
System.out.println("You entered "+number1+"!");

System.out.println("Please enter 2. number:");
int number2 = inputScanner.nextInt();
System.out.println("You entered "+number2+"!");

System.out.println("Please enter 3. number:");
int number3 = inputScanner.nextInt();
System.out.println("You entered "+number3+"!");
```

- But that is not a good practice!
  - It is needed to repeat identical code parts all over in the method *main*().
  - There exist a general concept concerning programming: Don't Repeat Yourself! DRY

- We can handle repeating parts of code by programming a loop to execute the same block repeatedly:

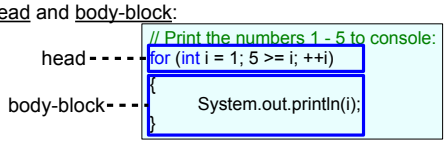
```
for (int i = 1; i <= 3; ++i) { // This for-loop executes the block below for three times.
    System.out.println("Please enter " + i + ". number:");
    int number = inputScanner.nextInt();
    System.out.println("You entered " + number + "!");
}
```

Control Structures – Iteration – for Loop – Part 1

- What we just have seen is the most versatile, but also most complex loop statement in Java: the for loop.

```
// Print the numbers 1 - 5 to console:
for (int i = 1; 5 >= i; ++i) {
        System.out.println(i);
}
```

- The for loop is a flexible, counting and head controlled loop statement. What does that mean?
  - It consists of two parts: head and body-block:



  - The head contains the control statements of the loop. This part looks complex, but that is the price for for's flexibility.
    - At least we have learned what the individual statements in the head do for us!
  - The body-block contains the statements, which are executed repeatedly as it is instructed in the head.

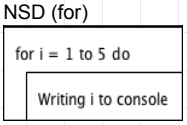
- With loops we introduce another control structure: called iteration (of statements).

- However, for now we're going to dissect the complexity of the for loop's head.

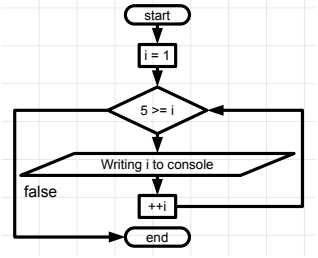
Control Structures – Iteration – for Loop – Part 2

- Counting and head controlled loops like the for loop do also have an NSD and flowchart representation:

NSD (for)



Flowchart (for)



Good to know
The counter variable of a for loop is an exception to the rule, that variables should have descriptive names. It is commonly accepted that these variables have very short names like *i* (maybe for "index"?), *k*, *j* and so forth.)
This tradition was taken over from maths, esp. from index variables.

- We can use the flowchart diagram to analyze, how the control statements of the for loop's head work:
  - Initialization expression or statement: `int i = 1;`
  - Conditional expression: `5 >= i;`
  - Update statement: `++i;`

```
// Print the numbers 1 - 5 to console:
for (int i = 1; 5 >= i; ++i) {
        System.out.println(i);
}
```

- The for loop as shown here works only, if the conditional expression and the update statement refer to the same state!
  - The state is given as the value *i*, which is evaluated by the conditional expression and modified by the update statement.

- The initialization statement can define multiple variables, but all must be of the same type (it makes sense, because in a Java statement we can generally only define multiple variables of the same type).

Control Structures – Iteration – for Loop – Part 3

- The for loop's complexity: the statements in the head and the body-block are not executed in the order they are written!
  - (1) If the conditional expression evaluates to true, the body-block will be repeated as next step, then the update expression will be evaluated.
  - (2) Then the conditional expression is evaluated anew and we goto (1).
  - The block of the head controlled for loop is only entered, if the conditional expression evaluates to true: we call it a pre-test-loop.

```
// Print the numbers 1 - 5 to console:
for (int i = 1; 5 >= i; ++i) {
        System.out.println(i);
}
```

- We stated, that the for loop is a counting loop. This is due to the fact, that we use a counter variable to control the loop.
  - Rule for orientation: use for if counting is involved, e.g. because we must know the number of the current iteration.

- Here some variations:
  - Use the counter *i* with an increment of 2 starting with 0, until 6 is reached; write the counter values to the console:

```
// Print the numbers 0 - 6 to console:
for (int i = 0; i <= 6; i += 2) {
        System.out.println(i);
}
```

  - We can write an infinity loop with for, if we only write empty statements in the loop head. – It is still required to write semicolons:

```
// Runs forever and writes this text to the console until the program is forcibly stopped:
for (; ;) {
        System.out.println("To infinity and beyond!");
}
```

Good to know:
A statement only consisting of a semicolon, i.e. w/o any expression is called null-statement. A null-statement just does nothing.

  - Generally, a program "rotating" in an infinity loop needs to be stopped forcibly, e.g. by stopping program execution or stopping the debugger session.

- A Java program running in the JVM can usually stopped by Ctrl-c from within the console.

Control Structures – Iteration – for Loop – Closing Notes

- for loops are executed sequentially, i.e. one repetition after another, not in parallel, e.g. not on multiple CPU cores.
  - Java also supports to do repetitions in parallel, but this is a topic for a future lecture.

- for loops are most universal and can imitate the functionality of any other loop!
  - We'll discuss other loop forms in Java in short.
  - A basic rule for orientation: generally use the for loop, if counting is involved, i.e. if a counter like *i* is required.

- for loops are popular to iterate arrays by index (filling, processing and output), which we will discuss in a future lecture.
  - This is so, because we can use the increasing counter variable's value as index for array-access.

- Java also supports for each loops, so called enhanced for-loops, but we'll discuss them in a future lecture!
  - Using for each loops would make handling arrays very simple in many cases, but we need to understand more background.

- For completeness: for loops can be cascaded of course:

```
// Prints 25 numbers to the console:
for (int i = 1; 5 >= i; ++i) {
        for (int j = 1; 5 >= j; ++j) {
            System.out.println(i * j);
        }
}
```

Control Structures – Iteration – while and do Loop – Part 1

- while loops are head controlled like for loops, but the syntax and its semantics is different, maybe even simpler.
  - Here we formulated the for loop printing ints from 1 to 5 to the console with a while loop:

```
// Print the numbers 1 - 5 to console:
for (int i = 1; 5 >= i; ++i) {
        System.out.println(i);
}
```

head

body-block

```
// Print the numbers 1 - 5 to console:
int i = 1;
while (5 >= i) {
        System.out.println(i);
        ++i;
}
```

  - Using the while loop, the state variable *i* is initialized before the while statement, i.e. it is not part of the head.
  - while's conditional expression is near the while keyword similar to if statements, it is the only part of the head.
  - The update statement is executed in while's body block, i.e. it is not part of the head.
  - The while loop is so simple to understand syntactically (head, body block), that no more explanations are given here.
  - => The important thing to note here, is that the statements in the while loop are executed in the order they are written!
- There is a key difference to for loops: the update-operation controlling the loop condition is done in the body block.

```
for (int i = 1; 5 >= i; ++i) {
        System.out.println(i);
}
```

```
int i = 1;
while (5 >= i) {
        System.out.println(i);
        ++i;
}
```

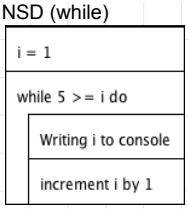
  - A typical beginners' error using while loops: the update-statement in while's body-block was forgotten, which leads to an infinity loop
  - There exist exceptions to the rule, that the update-operation needs to be done in the body block, we won't discuss here.
  - for and while are both head controlled/pre-test-loops: their body is only entered, if the conditional expression evaluates to true.

- In which situations is the update of the loop condition not required within the loop?
  - If we have a side effect in the conditional expression in the head or foot (e.g. ++*i*).
  - If the data that is checked in the loop condition will be somehow modified from "outside" (e.g. from different threads).

Control Structures – Iteration – while and do Loop – Part 2

- The flowchart representation of the while loop is equivalent to the that of the for loop, however, the NSD looks like this:

NSD (while)



- We can write a while infinity loop, if we ensure the conditional expression to evaluate to true forever:

```
// Runs forever and writes this text to the console until the program is forcibly stopped:
while (true) {
        System.out.println("To infinity and beyond!");
}
```

  - Remember that a program "rotating" in an infinity loop needs to be stopped forcibly.

- For completeness: while loops can be cascaded of course:

```
// Prints 25 numbers to the console:
int i = 1;
while (5 >= i) {
        int j = 1;
        while (5 >= j) {
                System.out.println(i * j);
                ++j;
        }
        ++i;
}
```

Control Structures – Iteration – while and do Loop – Part 3

- Remember, when we discussed console input via *Scanner*, esp. checking user input.
  - With the programming skills we had then, we were not able to deal with user input of unexpected data type appropriately:

```
System.out.println("Please enter your age:");
// Pre-check input: did the user input an int?
if (!inputScanner.hasNextInt()) {
        System.out.println("Your age must be a valid number!");
        inputScanner.next(); // Ignore the invalid user input.
}
```

Terminal

```
NicosMBP:src nico$ java Program
Please enter your age:
Nico
NicosMBP:src nico$
```

  - The problem: After the user entered invalid input, the program exits!
  - However, it would be better, if we were able to repeatedly asks the user to enter correct input and continue the program.

- However, with while we can ask the user for input until it is valid and then continue program execution:

```
System.out.println("Please enter your age:");
while (!inputScanner.hasNextInt()) {
        inputScanner.next(); // Clear Scanner's buffer from the invalid input.
        System.out.println("The age must be an int!");
        System.out.println("Please enter your age:");
}
int yourAge = inputScanner.nextInt();
```

Terminal

```
NicosMBP:src nico$ java Program
Please enter your age:
Miranda
The age must be an int!
Please enter your age:
```

Control Structures – Iteration – while and do Loop – Part 4

- The last loop variant we are going to discuss is the do while loop (sometimes also called do-loop).

- In opposite to while loops, do while loops are foot controlled.
  - Let's reformulate our first example using the while loop with a do while loop:

```
// Print the numbers 1 - 5 to console:
int i = 1;
while (5 >= i) {
        System.out.println(i);
        ++i;
}
```

```
// Print the numbers 1 - 5 to console:
int i = 1;
do {
        System.out.println(i);
        ++i;
} while (5 => i);
```

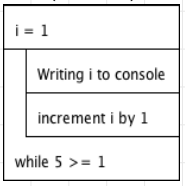
  - Like in the while loop, the update-operation controlling the loop condition is often done in the body block.
    - (There exist exceptions to the rule, that the update-operation needs to be done in the body block, we won't discuss here.)
  - However, different to while loops, the loop's control condition is specified in the foot section of do while's syntax.

- But with foot controlled loops, we also have a different way the control flow is executed:
  - (1) The code in do while's block is executed.
  - (2) Then the loop condition is evaluated. If it evaluates to true, the control flow starts over at (1).
  - The block of a foot controlled loop is entered at least once, therefor we call them post-test-loops.

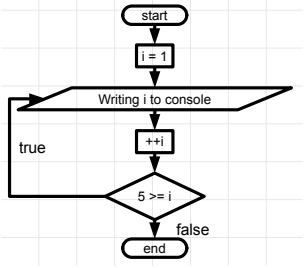
Control Structures – Iteration – while and do Loop – Part 5

- The flowchart and NSD representations of the do while loop reflects its foot controlled idea:

NSD (do while)

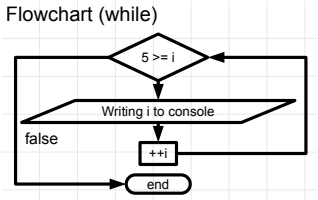


Flowchart (do while)

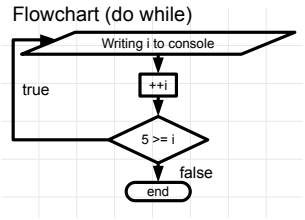


- Head controlled while loop vs foot controlled do while loop: notice the positions of the control condition in the control flow:

Flowchart (while)



Flowchart (do while)



Control Structures – Iteration – while and do Loop – Part 6

- We can write a do while infinity loop, if we ensure the conditional expression to evaluate to true forever:

```
// Runs forever and writes this text to the console until the program is forcibly stopped:
do {
        System.out.println("To infinity and beyond!");
} while (true);
```

  - Remember that a program "rotating" in an infinity loop needs to be stopped forcibly.

- For completeness: do while loops can be cascaded of course:

```
// Prints 25 numbers to the console:
int i = 1;
do {
        int j = 1;
        do {
              System.out.println(i * j);
              ++j;
        } while (5 >= j);
        ++i;
} while (5 >= i);
```

- There is a remarkable point: do while loops must be written with blocks! Also empty do while loops must use blocks:

```
// WRONG!
int i = 1;
do
        ++i;
while (5 >= i);
```

→

```
// Correct!
int i = 1;
do {
        ++i;
} while (5 >= i);
```

```
// WRONG!
int i = 1;
do
        ;
while (5 >= i);
```

→

```
// Correct!
int i = 1;
do {
        ;
} while (5 >= i);
```

Control Structures – Iteration – while and do Loop – Part 7

- Let's review our example using a while loop to handle valid user input:

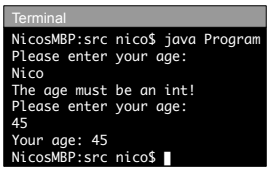
```
System.out.println("Please enter your age:");
while (!inputScanner.hasNextInt()) {
        inputScanner.next(); // Clear Scanner's buffer from the invalid input.
        System.out.println("The age must be an int!");
        System.out.println("Please enter your age:");
}
int yourAge = inputScanner.nextInt();
```

Advice: do while is excellent to write menus in a console program: "do show the menu while the program was not quitted".

  - However, there are issues: we have to write the code to print the prompt for two times!

- Using do while we can rewrite this code to deal with the prompt more efficiently:

```
boolean wasValidInput = false;
do {
        System.out.println("Please enter your age:");
        wasValidInput = inputScanner.hasNextInt();
        if (!wasValidInput) {
            inputScanner.next(); // Clear Scanner's buffer from the invalid input.
            System.out.println("The age must be an int!");
        }
} while (!wasValidInput);
int yourAge = inputScanner.nextInt();
```



  - To be frank, we have to use more code, but it better reflects the idea: "do the prompt while input is invalid".

Control Structures – Iteration – Loop Control – Part 8

- Occasionally it is required to control the execution of loops in a finer grained manner to execute loops only partially.
- Java provides the continue statement to skip iterations and uses the break statement, known from switch, to stop loops.
- Let's assume a loop, which prints the first 100 int numbers to the console, but no numbers, which are multiples of 10.
  - There're at least two ways to do it in Java: (1) print only non-multiplies of 10 or (2) skip console output, if we have a multiple of 10:

```
// (1) print only non-multiplies of 10:
for (int i = 0; 100 >= i; ++i) {
        if (0 != i % 10) {
            System.out.println(i);
        }
}
```

```
// (2) skip console output, if we've a multiple of 10:
for (int i = 0; 100 >= i; ++i) {
        if (0 == i % 10) {
            continue;
        }
        System.out.println(i);
}
```

- The continue statement does the trick, it skips the current iteration and continues with the next iteration.
  - (a) if *i* reaches the value 10, the condition *0 == i % 10* evaluates to true and continue is executed
  - (b) continue skips the console output and executes for's head, the loop condition evaluates to true and *i* is incremented to 11
  - (c) for's body block is executed …

- continue can be used with for, while and do while loops (also with for each loops).

Control Structures – Iteration – Loop Control – Part 9

- Java also provides a means to stop a running loop completely with the break statement.

- Assume an infinity loop, which prints all ints beginning from 0 to the console, unless the user stops this cycle:

```
int i = 0;
while (true) {
        System.out.println(i);
        System.out.println("Print next number? (n/y)");
        if (inputScanner.nextLine().equals("n")) {
            break;
        }
        ++i;
}
System.out.println("OK, loop stopped!");
```

```
Terminal
NicosMBP:src nico$ java Program
0
Print next number? (n/y)
y
Print next number? (n/y)
N
OK, loop stopped!
NicosMBP:src nico$
```

- Here, the break statement does the trick, it stops the loop completely and control flow continues after the loop.

Good to know:
Because infinity loops don't apply conditions, only break, continue, return and throw can end an infinity loop programmatically.

- A Java program running in the JVM can usually stopped by Ctrl-c from within the console.

Control Structures – Iteration – Loop Control – Part 10

- It should be said, that the same behavior can be achieved, by getting rid off the dangerous infinity loop:

```
// (1) Uses an infinity loop and break:
int i = 0;
while (true) {
        System.out.println(i);
        System.out.println("Print next number? (n/y)");
        if (inputScanner.nextLine().equals("n")) {
            break;
        }
        ++i;
}
System.out.println("OK, loop stopped!");
```



```
// (2) The same behavior without break:
int i = 0;
boolean userAbortedOutput = false;
while (!userAbortedOutput) {
        System.out.println(i);
        System.out.println("Print next number? (n/y)");
        if (inputScanner.nextLine().equals("n")) {
            userAbortedOutput = true;
        } else {
            ++i;
        }
}
System.out.println("OK, loop stopped!");
```

  - In (2) we just use the status variable *userAbortedOutput* to control the loop, instead of an infinity loop.

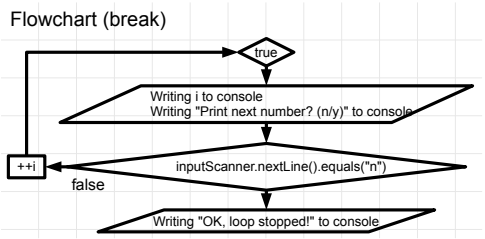
- break can be used with for, while and do while loops (also with for each loops).

Control Structures – Iteration – Loop Control – Part 11

- Code containing break and/or continue can basically only be illustrated with flowchart diagrams:

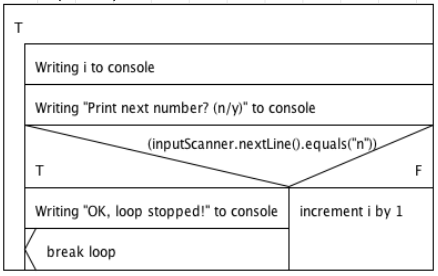
```
// Infinity loop and break:
int i = 0;
while (true) {
        System.out.println(i);
        System.out.println("Print next number? (n/y)");
        if (inputScanner.nextLine().equals("n")) {
            break;
        }
        ++i;
}
System.out.println("OK, loop stopped!");
```

Flowchart (break)



- Sometimes, people use the "EXIT" symbol to express break in NSDs:
  - (Variations of that symbol (different edges etc.) are sometimes used to express continue, but those are not standardized for NSDs).

NSD (break)



Control Structures – Iteration – Loop Control – Part 12

- Java provides more not so well-known means to control loop execution, when we use break/continue together with labels.

- What is a "label"?
  - Java code can be marked with special identifiers, i.e. labels, to give the code a more "row-wise" structure.

```
acceptInput:
System.out.println("Enter your name:");
String name = inputScanner.nextLine();
checkInput:
if (name.isEmpty()) {
        System.out.println("Aha, Mr. noname!");
} else {
        allFine:
        System.out.println("OK!");
}
```

  - In the code above, we have three labels: *acceptInput*, *checkInput* and *allFine*.
  - As can be seen, a label is a valid and unique Java identifier followed by a colon, which can be placed almost anywhere in the code.
    - E.g. they must be written before another statement.
  - Labels per se have no functional meaning for the program, but only have a documentary meaning like comments.
    - Labels can be thought of as replacement for "line numbers" as it is known from some Basic dialects.

- As mentioned, we can do more with labels, when loops, esp. cascaded loops come into play.

Control Structures – Iteration – Loop Control – Part 13

- In programming, we will often encounter cascaded loops.
  - Let's assume a cascaded loop, which counts up two numbers that are multiplied.
  - The user is prompted to stop generating the products and console output after each product respectively.

```
for (int i = 1; i < 10; ++i) {
        for (int j = 1; j < 10; ++j) {
            System.out.println(i * j);
            System.out.println("Print next product? (n/y)");
            if (inputScanner.nextLine().equals("n")) {
                break;
            }
        }
}
System.out.println("OK, loop stopped!");
```

```
Terminal
NicosMBP:src nico$ java Program
1
Print next number? (n/y)
y
2
Print next number? (n/y)
n
2
Print next number? (n/y)
```

- No, it does not work! The program keeps generating products, even though the user answered with "n"!
  - After the user entered "n", we would have assumed, that program prints "OK, loop stopped!" to the console.
  - The problem: break does only stop the closest loop, i.e. the for loop counting up *j*!

Control Structures – Iteration – Loop Control – Part 14

- For exactly such cases, i.e. cascaded loops need to be stopped from the innermost loop, we can use break with a label.

```
endOutput:
for (int i = 1; i < 10; ++i) {
        for (int j = 1; j < 10; ++j) {
            System.out.println(i * j);
            System.out.println("Print next product? (n/y)");
            if (inputScanner.nextLine().equals("n")) {
                break endOutput;
            }
        }
}
System.out.println("OK, loop stopped!");
```

```
Terminal
NicosMBP:src nico$ java Program
1
Print next product? (n/y)
y
2
Print next product? (n/y)
n
OK, loop stopped!
NicosMBP:src nico$
```

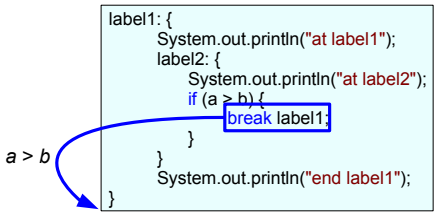
  - As can be seen, we have prefixed the outermost loop of the cascaded loops with a label *endOutput*.
    - Then this label marks the loop, which we will "break" effectively.
  - This new variant of the break statement, addresses the label *endOutput*.
  - When the break statement is executed, control flow will jump **behind** the loop, which was marked with the label *endOutput*.
    - Indeed, with break+label the control flow doesn't jump before the "labelled" loop, but right **behind** it!

- In Java, we also have continue with a label, which would lead to a jump to the label **before** the labelled loop.

- Tip: break/continue with labels should not be used, because the code look/behaves weird.
  - Better rewrite your code!

Excursus: forced Branching – Part 1

- Actually, we can do more with labels apart from branching: we can use them to code forced branching.
- Here an (contrived) example:



  - if a > b, then the control flow jumps right below the block marked with *label1*:

```
Terminal
NicosMBP:src nico$ java Program
at label1
at label2
NicosMBP:src nico$
```

  - if a <= b, then the control flow just continues to the end.

```
Terminal
NicosMBP:src nico$ java Program
at label1
at label2
end label1
NicosMBP:src nico$
```

Excursus: forced Branching – Part 2

- Remarks on forced branching with labels:
  - Forced branching is different to conditional branching, in that it uses the if keyword and jumps to labels.
  - Using forced branching practice is really unconventional and leads to weird code! → Don't use it!
  - (But such code could be result of automatic code generation. – It is simple to generate such code.)
  - Only the break statement can be used.
    - The continue statement can only be used in loops.

- Forced branching is very similar to unconditional branching, i.e. gotos.
  - However, Java disallows unconditional branching with labels:

```
label1: {
        System.out.println("at label1");
        label2: {
            System.out.println("at label2");
            break label1; // Notice, there is no if statement.
        }
        System.out.println("end label1");
}
```

Unreachable statement

    - This won't compile, because the break statement would always (unconditionally) lead the control flow to "jump" below *System.out.println("end label1")*.
    - If this coding style, the spaghetti style (control flow looks like intertwined cooked spaghetti), was legal in Java, it would introduce a whole class of bugs
  - Java defines the keyword goto, but the language forbids its usage currently, so it has just no function in Java.

Control Structures – Exiting Iteration – Part 1

- There are even more ways to "exit" a loop in Java, which will discuss in future lectures.

- The return statement exits the surrounding method and thus all surrounding loops.

```
public class Program {
	public static void main(String[] args) {
		for (int i = 1; i < 10; ++i) {
			for (int j = 1; j < 10; ++j) {
				System.out.println(i * j);
				System.out.println("Print next product? (n/y)");
				if (inputScanner.nextLine().equals("n")) {
					return;
				}
			}
		}
	}
}
```

  - A return statement could also transport a value to the caller of the method, which acts as a result of the method.
  - We will discuss methods in depth soon.

- Similar to return statements are throw statements, in that they also exit the surrounding method and all surrounding loops.
  - throw statements are used to emit exceptions. Exceptions are a powerful way to deal with run time errors.

Control Structures – Exiting Iteration – Part 2

- Special statements for loop control:
  - continue – skips the current loop.
  - break – leaves the next surrounding loop completely.

- labels and break
  - Using labels and break, Java supports a limited way of forced branching.
  - But this is basically never required.

- return and throw
  - Return from a method with or without value.
  - Throw an *Exception*, which leaves the method as well.

We should always use Blocks and Indentation – True Example using goto

- With a solid understanding of conditional code and blocks, the hideous goto stresses their relevance.

- Apple introduced a serious and hard-to-see bug in 2014 in their SSL/TLS library (C/C++), the code looked like this:

Hint
C and C++ support goto statements, whereas Java doesn't!

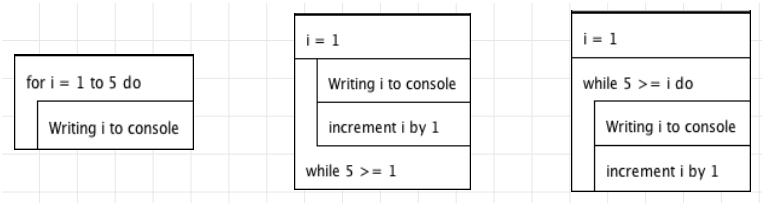
```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) // (1)
        goto fail;
        goto fail; // (2)
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
        // ...
fail:
```

  - The problem: the second goto in (2) was always executed, because it was not executed under the condition of (1)!
  - Apple's developers avoided blocks to structure the code, which might have unleashed this **serious bug**:

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) { // (1)
        goto fail;
}
goto fail; // (2) Oups!
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0) {
        goto fail;
}
// ...
fail:
```

- Indentation is only a visual help to understand the code, but it is important to understand code from a distance.
  - Blocks are important to understand the scope of statements, this example shows how the goto goes horribly wrong!

Control Structures – Iteration – Summary



- General remarks on loops:
  - for, while and do while are statements, which can be cascaded in any thinkable way like if statements.
  - Beware of unwanted infinity loops!
    - Infinity loops are a very common error due to wrong conditional expressions or unwanted modifications of the counter variable in the body-block.
    - You should know how to stop program execution from the console, on the IDE and platform you use (Unix/Linux/macOS/Windows).
  - Basically, you can use all loop variants for everything, however a certain loop variant can be beneficial over the others.
  - break and continue should be used sparingly. Often, loops can be rewritten without those control statements!
  - return and throw statements can also be used to exit the surrounding method and all surrounding loops.
    - We will discuss methods in a future lecture.
  - for, while and do while loops are executed sequentially, not in parallel, e.g. not on multiple CPUs or cores.

- Later, we'll discuss the for each loop, that can solve problems like infinity loops and counter issues in for loops.

- Java's *Stream* API will also simplify code a lot to avoid many loops and it supports parallel loops running on multiple CPU cores.

Imperative Programming – Closing Words

Definition
Imperative programming means to program with statements being executed sequentially to change the state of the program.

- Statements are executed in the order they where written.
  - Whereas in functional programming so-called lambdas and deferred execution this is not necessarily the case.

- Statements change the state of the program by modifying variables.
  - Those modified variables control the execution of following statements, esp. control structures, e.g. if's and loops.
  - We say, that variables communicate via side-effects with the statements, that are executed next.
  - Where in functional programming side effects are disallowed and control structures work completely differently, if any.

Structured Programming

- Programming only using sequences, loops and branches is called structured programming.
  - Very traditionally, structured programming means, that code using unconditional branching (gotos) is prohibited.
  - Code with gotos often evolves to so called spaghetti code, i.e. very intertwined code of low maintainability(/quality).

- The initiative of structured programming begun in the 1970ies and was coined by Mr. Edsger Wybe Dijkstra [ˈdɛɪkˌstra].
  - During that time the era of HLLs rose.
    - Lower level (programming) languages like assembly languages use jump/branch statements, i.e. unconditional branching, inherently.
  - That time also the need to document algorithms in a structured way, with flowcharts and esp. NSDs came into play.

- Spaghetti code: program execution is on position x and I know, from where I jumped here, and now program execution will jump to y.

Thank you!