(1) Java Introduction:
Introducing the Java Programming Language

Nico Ludwig (@ersatzteilchen)

TOC

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!

Why do we develop new Software?

- Before we start to understand how programming works, we have to understand why we do software development.
- Usually, the basic thing programs do, is solving a problem, which is hard/cumbersome/impossible to be done manually.
  - In most cases, if not all, this problem must be solved repeatedly.
- In other words, if we know an existing program that already solves the problem in question, we have nothing to program!
  - It sounds silly, but good programmers know, when they have nothing to program at all!
  - Many existing programs are reusable for a class of similar problems.
- So called standard software, like office products are extreme examples of reusable programs.
- A so called expert system is software working as a knowledge-based system.
  - Such a system needs to be "charged" with pretty simple basic rules of the expert knowledge domain.
  - Those rules are evaluated, e.g. via a predicate logic, to, e.g. derive a certain illness from a bunch of symptoms.
  - E.g. the system derives knowledge from its pre-defined rules. – An expert system is an implementation of artificial intelligence.
- But in this course we'll more or less only write specialized programs to solve our special problems/assignments.
  - We're not going to learn how to write standard software!

How do we develop Software?

- Development of software is classically done in phases. Following phases could be identified for example:
- Phase 1: Formulation of the problem as functional specification or one or more user stories or one or more epics.
- Phase 2: Analysis of the problem to extract the algorithm.
  - Identify input data, the result (i.e. output data) and the processing approach.
- Phase 3: Graphical representation of the processing approach.
  - Graphical representations of processing approaches with flowcharts or Nassi–Shneiderman diagrams (NSDs or "structograms") can be useful.
- Phase 4: Implement the extracted algorithm in a programming language.
  - During programming, so called dry runs with value tables are useful to check algorithms for correctness.
- Phase 5: Program test
- Phase 6: Documentation

- The functional specification is called "Plichtenheft" in German.
- A dry run is called "Schreibtischtest" in German.

Phase 1: Formulation of the Problem – Challenge: Interpret an Update Log

- Real world programs can be quite complex, therefor it is required to define the customer requirement (c-requirement/Lastenheft).
  - The c-requirement can be very detailed, so that customer/contractor have a common understanding of the problem to be solved.
  - The c-requirement is provided by the customer, the contractor "answers" with a developer requirement (d-requirement/Pflichtenheft) "how will it be implemented".
  - => This follows the standardized definition of c- and d-requirement of the Software Requirements Specification (SRS) after the IEEE.
- Another popular way to formulate a smaller problem to be solved with a smaller program, is the definition of a user story.
  - A user story describes the problem to be solved in plain language from the perspective of the user, who has the problem.
- Let's assume this user story for the following discussion:
  - "As an administrator I want to know the latest updates on a PC, in order to get a report of that PC's state!"
- It is needed to interpret the file "Software Update.log", which looks like this:

```
2009-04-06 12:43:04 +0200: Installed "GarageBand Update" (5)
2009-04-06 12:44:34 +0200: Installed "iMovie Update" (8)
2009-07-30 13:11:28 +0200: Installed "iMovie Update" (9)
2009-04-06 12:43:31 +0200: Installed "iTunes" (8)
. . .
```

6

- How can we solve this problem?

- The "Pflichtenheft" will usually be part of the contract, esp. when following the idea of the German "Werkvertrag", when a typical project with the waterfall approach is to be done. With waterfall the "Pflichtenheft" is done in the analysis phase(s). When teams follow an agile approach, usually a "Dienstvertrag" is applied, because the further-development of a product should rather be paid over time as we have no actual "ready" product, at least not very early.

Phase 2: Analysis of the Problem to extract the Algorithm – Input Data

- Initially, the file to be analyzed looks quite complex, but a closer look shows, that we only need to inspect few data.
  - Analysis of input data requires us to answer two questions basically.

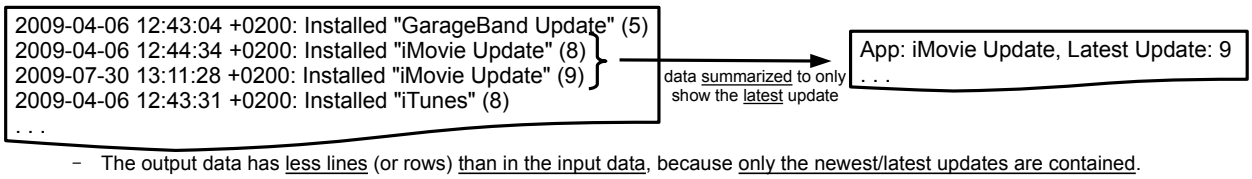
```
2009-04-06 12:43:04 +0200: Installed "GarageBand Update" (5)
2009-04-06 12:44:34 +0200: Installed "iMovie Update" (8)
2009-07-30 13:11:28 +0200: Installed "iMovie Update" (9)
2009-04-06 12:43:31 +0200: Installed "iTunes" (8)
. . .
```

- Question 1: What input data is required to solve the problem?
  - Each line represents a software update and each line is independent of the other lines.
  - The important portion of data can be found in the last parts (or columns) of each line, there we can find:
    - (1) the software name of the software being updated (blue boxes),
    - and (2) the version number of the update (red boxes) – the higher the number, the newer/later the update.
    - => The rest of the data in a specific line can be ignored for the analysis.
- Question 2: Of which data type is the input data?
  - Another important point is that the software name is textual data and the software version is numeric data.
- Mind, that all updates of a specific software item are listed in that file.
  - E.g. the above snippet contains two entries for "iMovie Update", one for version 8 and one for version 9.
  - This also means, that we have to analyze each line of the file, because we want to find the latest version reliably!

Phase 2: Analysis of the Problem to extract the Algorithm – Output Data

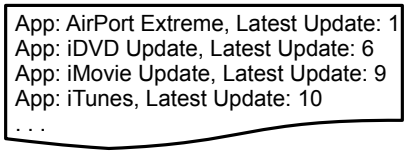
- Now we have to exactly define the presentation of the solution of the problem to the user, which satisfies the user story.
  - In "mathematical" terms the solution is the result of the program, in programing terms, it is the output data of the program.

- Only relevant data is presented, superfluous data eliminated



  - The output data has less lines (or rows) than in the input data, because only the newest/latest updates are contained.
  - Each row in the output data has only two portions of data: the name of the app and its newest/latest update's version number.

- The output data should be formatted and sorted by the software's (app) name respectively. Then we'll have this output:



Phase 2: Analysis of the Problem to extract the Algorithm – Processing Approach

- Now we have to think about the processing steps and their order to get and output the desired output data.

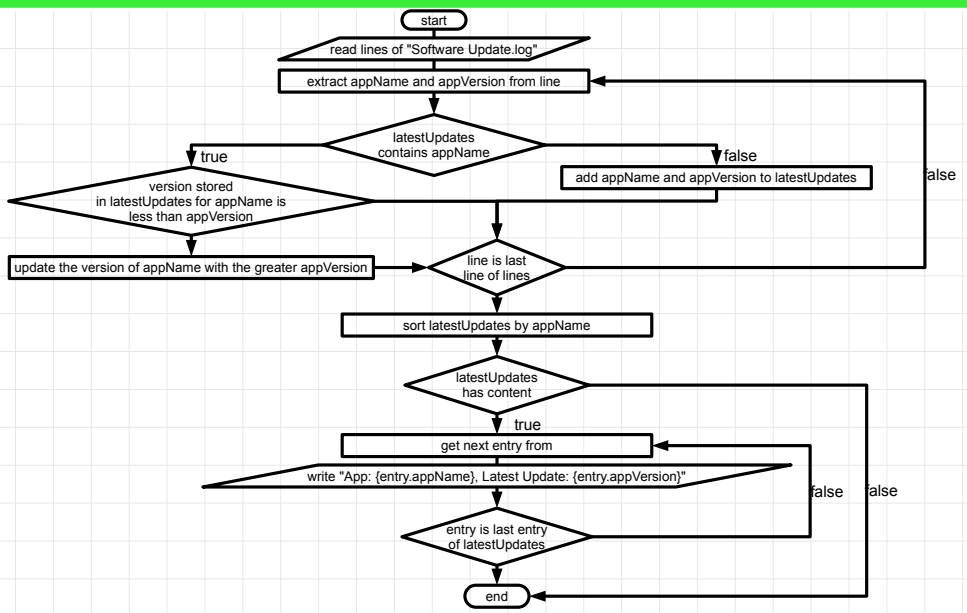
- (Step 1) We have to ready each line of the log file.
  - (Step 2) For one line extract the software name and the software version number.
    - (Step 3) Then store the extracted software name and extracted software version number in a dictionary:
      - (Step 4) If the extracted software name is already stored in the dictionary check the version:
        - (Step 5) If it is less than the extracted version replace the stored version with the extracted version.
        - (Step 6) Else: do nothing!
      - (Step 7) Else: If the extracted software name is not stored in the dictionary:
        - (Step 8) Store the extracted software name and extracted software version number in the dictionary
    - (Step 9) If that line was not the last line, repeat all steps from (2) with the next line.
  - (Step 10) If that line was the last line:
    - (Step 11) Sort the data by the software name.
    - (Step 12) For each dictionary's entry:
      - (Step 13) Format the data with "App: <software name>, Latest Update: <latest update>"
      - (Step 14) Print the formatted entry to standard out, so that the user can see the result.
- (Step 15) Program end.

Phase 3: Graphical Representation of the Processing Approach – Overview

- Sometimes, it can be useful to discuss the processing approach in a graphical way. There are some benefits:
  - There are some standardized diagram types, which are completely independent of a certain programming language.
  - The resulting diagrams can be more human readable.

- Now we'll look at two graphical representations:
  - Flowchart diagrams
  - Nassi-Shneidermann Diagram

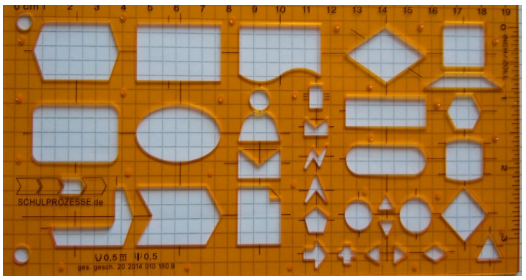
Phase 3: Graphical Representation of the Processing Approach – Flowchart



Phase 3: Graphical Representation of the Processing Approach – Some Words on Flowcharts

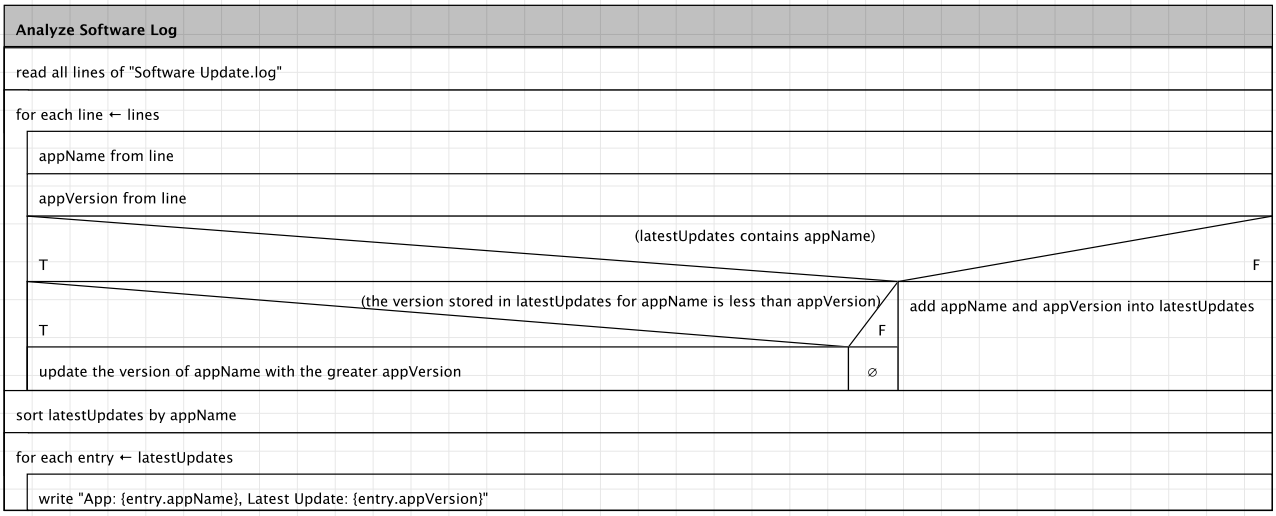
- Flowcharts are diagrams after the standard ISO 5807/DIN 66001, which show the flow of control in a "process".
  - The diagram just connects different symbols with connectors (w/ or w/o arrow tips) to depict the flow of control.
  - Flowcharts are not only used to visualize algorithms, but all kinds of control/information flow, e.g. business processes.

- Benefits:
  - They can be easily drawn without software by hand. Special stencils are also available.
  - Flowcharts can also easily be modified with rubber and pencil.
  - Because of its simplicity it is the ideal means to design algorithms on the fly.



- Downsides:
  - People say, that it is simple to design unstructured algorithms, which contain unconditional jumps (so called gotos) with flowcharts.
    - The diagram would in this case have many connection lines, which unconditionally connect rectangular statement symbols.

Phase 3: Graphical Representation of the Processing Approach – NSDs



Phase 3: Graphical Representation of the Processing Approach – Some Words on NSDs

- Nassi-Shneidermann Diagram (NSD, structogram) after the standard DIN 66261.
  - Introduced by Isaak Nassi and Ben Shneidermann in 1972/1973.

- Benefits:
  - NSDs can be read top down basically. With flowcharts instead, we have to follow the lines.
    - NSDs are more structured than flowcharts, hence the name "structogram".
  - NSDs support a very strict design, it helps implementing structured programs.

- Downsides:
  - NSDs are relatively difficult to draw. Changes in a diagram often leads to a complete redraw.
  - Alas, software engineers will have to cope with NSDs in school at a certain point in time.
    - … and the need to draw an NSD is a frustrating task in an assignment or exam, because the draw process doesn't forgive errors!
    - Tip: If you can decide to draw a diagram (esp. NSD) or to write pseudo code in an exam choose to write pseudo code!

Phase 3: Graphical Representation of the Processing Approach – the Reality

- Nowadays, graphical representations of processing approaches at the design phase are used rarely.
- Diagrams are not suited for complex algorithms:
  - The required space is just too large, loosing its most important benefit being human readable.
  - Such diagrams are difficult to maintain.
- Generally, algorithms are different today:
  - Flowcharts and NSDs have been designed to support imperative, esp. structured programming.
    - Structured programming means to program only using sequences, branches and loops.
  - The usage of procedural, functional and object-oriented means to solve problems renders flowcharts and NSDs inappropriate.
  - Parallel programming to exploit the multicore processing power of modern CPU is often difficult to depict.
    - However, NSDs have been extended to express parallel algorithms in a limited way.
- Software development (SD) is different today:
  - Flowcharts and NSDs clearly rather support waterfall approaches on SD, but nowadays agile SD approaches are applied.
- Flowcharts still have a certain relevance to show a higher level of solution domains: service interaction, business processes and other workflows.
  - Mind that those modern incarnations of graphical representations base on flowcharts and not on NSDs!

15

- Nowadays a lot of (free) Computer Aided Software Engineering (CASE) tools are available, which allow drawing flowcharts and NSDs. Such tools also allow converting code to a diagram and vice versa.

Phase 4: Implement the extracted Algorithm in a Programming Language – Part 1

- Sure, to solve the problem defined in the user story we're going to implement a program. But: what is a program?
  - A program is a logical series of commands to define work instructions to complete a task, which can be executed as algorithm.
- All right! And what is an algorithm exactly? An algorithm is
  - a series of exact commands (we already know this),
  - which certainly and repeatedly yields correct solutions for the problem
  - with a finite set of steps.
- A program should be written in a common way, that it can be applied for similar problems. – It should be reusable!
- => We are going to implement the analyzed problem in a symbolic, i.e. "real" programming language (language).
  - A symbolic language allows to express processing steps of an algorithm with a set of clearly defined symbolic command names.
  - Usually, such a symbolic language is also called High-Level Language (HLL, German: "Hochsprache").

16

- Which programming languages are known to the audience?
- Now let's inspect how we can solve this problem with a computer program. – We are going to discuss solutions in some high level languages (HLLs) that are appropriate to solve this problem.
- What is a HLL?
  - A HLL allows expressing solutions in a more human readable way. Hardware near languages (e.g. assembly) are hardware specific and not standardized, HLLs are in opposite hardware independent and standardized (so is Java).
- What is a low level language?
  - A low level language is a machine near language that works only on one type of machine (assembly), so they are often not portable.

Phase 4: Implement the extracted Algorithm in a Programming Language – Part 2

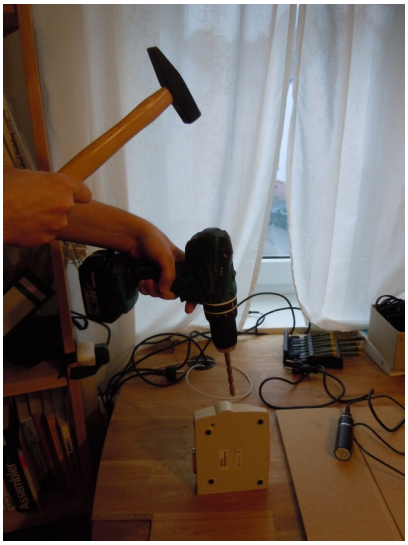
- But … which programming language (language) should we use to code this algorithm?

- Choosing a language is often driven by personal preferences.
  - Languages are just tools!
  - Craftspersons use different tools for different problems – a screwdriver isn't better than a hammer, its different!
  - When you only know how to use hammers, every problem might look like a nail – but not all problems are nails!
  - To be frank, it can be hard to switch from the present using language: IDEs and build environments must be switched as well.

- On the following slides, we implement our problem with different languages.
  - Ideally, we should compare and choose languages regarding the problem we're going to solve. – Choose the right tool!
  - We'll also see, that our problem can be solved simpler in one language than in another.

- But before looking at other tools, let's first take a look at Java, the tool, we are going to learn in this course.
  - Yes, we choose also Java to solve this problem, because we want to learn Java in this course!

Phase 4: Implement the extracted Algorithm in a Programming Language – Part 3



Avoid the hammer-effect: try to understand, what the problem is, not how it can be solved with a certain tool!

Java 17 (Stored in the File Program.java)

```
import java.nio.file.*;
import java.util.*;
import java.util.regex.*;

public class Program {
	public static void main(String[] args) throws Exception {
		// check arguments from console and check the file
		if (0 < args.length && Files.exists(Paths.get(args[0]))) {
			// open the file, read all lines
			final Map<String, String> latestUpdates = new TreeMap<>(String.CASE_INSENSITIVE_ORDER);

			final Pattern parseRex = Pattern.compile("[^\"]*\"(?<appName>[^\"]*)\"\\s*\\((?<versionNo>[^(]*)\\)");
			for (final String line : Files.readAllLines(Paths.get(args[0]))) {
				// interpret the line: e.g. 2009-04-06 12:42:58 +0200: Installed "Digital Camera Raw Compatibility Update" (2)
				final Matcher matcher = parseRex.matcher(line);
				if (matcher.matches()) {
					// store the interpreted data
					final String key = matcher.group("appName");
					final String value = matcher.group("versionNo");
					if (!latestUpdates.containsKey(key) || 0 < value.compareTo(latestUpdates.get(key))) {
						latestUpdates.put(key, value);
					}
				}
			}
			// output the collected data to console
			for (final String appName : latestUpdates.keySet()) {
				System.out.printf("App: %s, Latest Update: %s %n", appName, latestUpdates.get(appName));
			}
		}
	}
}
```

Step 1: read all lines

Step 2: appName and appVersion from line

Steps 3-8: complete logic to care for the latestUpdates dictionary

Step 11: automatically sorts the latestUpdates dictionary

Steps 12-14: write each entry to standard out

- We can identify certain pieces in the code along the steps of the processing approach we have determined.
  - Esp. because this is a symbolic code, it resembles partially spoken (English) language.
  - The indentation of this code was also done along the processing steps to give the code a structure.
  - The indentation of this code does also resemble the cascaded style of the NSD we have designed.
- Following a structure of steps in program code is called the "imperative programming paradigm".

Phase 4: Implement the extracted Algorithm in a Programming Language

- Before we can put the presented Java program into effect, it needs to be compiled.
  - Compilation means in this context, that the symbolic Java code is transformed in a kind of machine code, so called byte code.
  - Java code is just written in a text file with the extension .java. The file must have the same name as the contained "class".
    - Thus the name of the java-file to be compiled is Program.java.
  - A compiler is a program, which performs this transformation. The Java compiler "javac" can be started from the command line.

- Compile the Java program using "javac", the compiled machine code is put into a binary file named Program.class

```
Terminal
NicosMBP:src nico$ javac Program.java
NicosMBP:src nico$
```

Good to know
javac is usually pronounced [ˈdʒɑːvəziː]

- After compilation, we can execute Program and pass the path to "Software Update.log" as argument:

```
Terminal
NicosMBP:src nico$ java Program "Software Update.log"
App: AirPort Extreme Update 2008-002, Latest Update: 1
App: Front Row Update, Latest Update: 1
App: J2SE 5.0 Release 4, Latest Update: 4
App: Keynote Update, Latest Update: 3
...
```

- On following slides, we will see how our user story can be programmed in other programming languages.

Groovy 2.1.5

```
import java.nio.file.*

// check arguments from console and check the file
if (args[0] && Files.exists(Paths.get(args[0]))) {
    def pattern = /.*\"([^\"]*)\"\s*\(([^\)]*)\)/
    def latestUpdates = new TreeMap<String, String>(String.CASE_INSENSITIVE_ORDER)
    // open the file, read a line
    new File(args[0]).each {
        // interpret the line: e.g. 2009-04-06 12:42:58 +0200: Installed "Digital Camera Raw Compatibility Update" (2)
        def matcher = it =~ pattern
        if (matcher) {
            // store the interpreted data
            def key = matcher[0][1]
            def value = matcher[0][2]
            if (!latestUpdates[key] || value < latestUpdates[key]) {
                latestUpdates[key] = value
            }
        }
    }
    // output the collected data to console
    latestUpdates.each { appName, versionNo -> println "App: $appName, Latest Update: $versionNo" }
}
```


- As can be seen, this code resembles the Java code we have just seen:
  - It was written to have an indented style. It also uses braces.
  - The same steps are also present, reading all lines of the file etc.
  - The similarity of Groovy to Java is no coincidence, Groovy is structurally (in programmers' terms "syntactically") based on Java and does even use the JVM to run its code.
- Designed/developed by James Strachan and Guillaume LaForge, it appeared in 2003.
- A multi-paradigm language. It does esp. also support dynamic typing.
- Can be interpreted or compiled, so it can be used as scripting language.
- Based on the JVM.
  - It is counted as the second JVM language besides Java. Groovy is a first class citizen on the same VM, parallel to Java.
  - The resulting byte code is 100% compatible to Java's byte code. Java can directly use Groovy classes and vice versa.
  - Is a kind of Java++, because it addresses Java's shortcomings with new features: closures and an intuitive syntax for the most often used Java collections (lists and maps). – One could just take some Java code and "groove it up".
  - Groovy tidies up some verbose syntax of Java. It kind of has borrowed more features from Ruby and Python than from C. – Esp. it borrowed the ability to deal with dynamic types from these languages.
  - Attention: Java code cannot be directly ported to Groovy! E. g. an expression like *c* == *t* compares references in Java, but calls *equals()* in Groovy and in Groovy some literals are automatically converted to reference types (e.g. int literals are represented as objects of reference type *Integer*), which yields to a different behavior.
- Groovy is esp. interesting to build Domain Specific Languages (DSLs) on top of it with so called "builders". Other features of language support creating DSLs: we can leave away certain imports, return statements and some parentheses on method calls, sometimes operators can be used instead of method calls.

BeanShell 2.0

```
import java.nio.file.*;
import java.util.regex.*;

// check arguments from console and check the file
if (0 < bsh.args.length && Files.exists(Paths.get(bsh.args[0]))) {
	// open the file, read all lines
	latestUpdates = new TreeMap<>(String.CASE_INSENSITIVE_ORDER);

	parseRex = Pattern.compile("[^\"]*\"(?<appName>[^\"]*)\"\\s*\\((?<versionNo>[^\\(]*)\\)");
	for (line : Files.readAllLines(Paths.get(bsh.args[0]))) {
		// interpret the line: e.g. 2009-04-06 12:42:58 +0200: Installed "Digital Camera Raw Compatibility Update" (2)
		matcher = parseRex.matcher(line);
		if (matcher.matches()) {
			// store the interpreted data
			key = matcher.group("appName");
			value = matcher.group("versionNo");
			if (!latestUpdates.containsKey(key) || 0 < value.compareTo(latestUpdates{key})) {
				latestUpdates{key} = value;
			}
		}
	}
	// output the collected data to console
	for (appName : latestUpdates.keySet()) {
		System.out.printf("App: %s, Latest Update: %s %n", appName, latestUpdates{appName});
	}
}
```


- A scripting language running on the JVM.
- The syntax is very Java-near, only a few syntactic simplifications were added.
- It is used for scripting, configuration and automation purposes Esp. to offer macro features, e.g. for Apache JMeter and Apache Ant. The integration into Java is seamless, so that Java-types can be used and also extended from BeanShell code.
- BeanShell supports dynamic typing and closures. – The idea to bind closures to this, allows prototype-based polymorphism.
- The BeanShell interpreter/parser can also be used from within Java code, hence its use for automation purposes as explained above.
- BeanShell is in a kind of maintenance mode currently, in the meantime. BeanShell2 was a fork created by Google, but also in maintenance mode right now.
- However, currently, Groovy seems to be the better way to script on the JVM.

Kotlin

```
import java.nio.file.*
import java.util.*

fun main(args: Array<String>) {
	// check arguments from console and check the file
	if (args.isNotEmpty() && Files.exists(Paths.get(args[0]))) {
	// open the file, read all lines
	val latestUpdates = TreeMap<String, String>(String.CASE_INSENSITIVE_ORDER)

	val parseRex = Regex("[^\"]*\"(?<appName>[^\"]*)\"\\s*\\((?<versionNo>[^\\(]*)\\)")

	for (aLine in Files.readAllLines(Paths.get(args[0]))) {
			// parse the line: e.g. 2009-04-06 12:42:58 +0200: Installed "Digital
			// Camera Raw Compatibility Update" (2)
			val matchResult = parseRex.find(aLine)
			if (null != matchResult) {
				// store the parsed data
				val key = matchResult.groups["appName"]!!.value
				val value = matchResult.groups["versionNo"]!!.value
				if (!latestUpdates.containsKey(key) || value.toInt() > latestUpdates[key]!!.toInt()) {
					latestUpdates[key] = value
				}
			}
	}
	// output the collected data to console
	for (appName in latestUpdates.keys) {
			println("App: $appName, Latest Update: ${latestUpdates[appName]}")
	}
	}
}
```


- Kotlin is a language designed by the company JetBrains from 2011 to 2016, which is also the manufacturer of the IDE IntelliJ IDEA.
  - JetBrains wanted to develop a language for more productive programming in comparison to Java.
    - First they looked at Scala, but they found Scala being hardly "toolable", esp. because of its rich feature-set.
  - IntelliJ IDEA has of course excellent support for Kotlin.
  - Kotlin is the name of an island near St. Petersburg.
  - Open source with Apache 2 license.
- JetBrains plans to strategically place Kotlin (at least) as the second language (after Java) for Android programming.
  - It can be assumed, that Kotlin was also a reaction to Apple's Swift a simpler "iOS-language" in comparison to Objective-C.
  - Google recommends implementing new Android apps with Kotlin (2020).
- The idea after Kotlin's design is to have a language, which allows more productive development processes. – This came to the price, that Kotlin's syntax is not compatible with Java, i.e. a different approach like Groovy's.
  - The quality of Kotlin (less boilerplate code, less syntactic fluff) stems from the fact, that JetBrains as developer of IDEs has a good knowledge of the typical problems Java developers encounter during programming.
  - Therefor one can say, that Kotlin is rather a language for "switchers" from Java.
- Some features:
  - Similar to Groovy present JDK classes got extended.
  - Also similar to Groovy is its ability to act as basis for DSLs.
  - Similar to Swift, Kotlin enables a kind of "second type system", which disallows values being null.
  - Kotlin features reified generics, i.e. generics for which the type argument is known at run time.

Clojure

```
(use 'clojure.java.io)
(with-open [rdr (reader (first *command-line-args*))]
  (let [lines (line-seq rdr)]
    (doall
      (map (fn [result] (printf "App: %s, Latest Update: %s %n" (first result) (apply max (second result))))
        (sort
          (reduce (fn [map item]
                    (if (contains? map (get item :appname))
                      (assoc map (get item :appname) (conj (get map (get item :appname)) (get item :versionno)))
                      (assoc map (get item :appname) (list (get item :versionno))))) {}
            (map (fn [matcher] {:appname (.group matcher "appName") :versionno (Integer/parseInt (.group matcher "versionNo"))})
              (filter (fn [matcher] (.matches matcher))
                (map (fn [line] (re-matcher #"[^\"]*\"(?<appName>[^\"]*)\"\s*\((?<versionNo>[^\(]*)\)" line))
                  lines)))))))))
```

24

- Clojure was mainly designed by Rich Hickey starting in 2007.
- Clojure looks completely different compared to Java and Groovy!
  - We can not even spot the steps of the processing approach we have discussed.
  - Also the indentation is completely different.
- This example does also underscore that the same problem can be solved with very different processing approaches!
- Clojure is a language, which follows a programming paradigm different from that we have seen in the Java code. Its paradigm is called functional programming. – Clojure is a functional programming language.
  - Clojure is derived from the programming language Lisp, i.e. it is "a Lisp dialect for the JVM".
    - A syntactic peculiarity of Lisp we also see in Clojure is the frequent usage of parentheses, which define lists. Lists are the core idiom of Lisp, hence the name, Lisp stands for "list processing".
  - A program written following the functional paradigm can generally not be expressed as a series of processing steps or visualized with, e.g. an NSD.
  - Functional code rather describes the result, than listing the commands to calculates the result.
  - Functional programs are rather to be understood as a gigantic mathematical formula. – In such formulas we can roughly spot individual steps.
- Some special features:
  - We can also create scripts with Clojure, which avoids the creation of byte code (compilation).
  - REPL (repeat-eval-print-loop) is supported, which allows interactive development. – This makes a lot of sense, because functional programming languages disallow side effects, which are to be avoided with REPL except the print step.
  - Besides Clojure's support of dynamic typing, user defined types are not only based on the idea of classes taken from Java, but also based on maps: objects, JavaBeans and POJOs can be handled as a map of key-value pairs, which make an object's fields and their values. – Handling objects like maps allows to apply all sorts of collection operations an them, which is also an

Scala

```
import scala.io.Source
import scala.util.matching.Regex

Source.fromFile(args(0))
    .getLines
    .map{line => new Regex("[^\"]*\"([^\"]*)\"\\s*\\(([^\\(]*)\\)", "appName", "versionNo").findFirstMatchIn(line).get}
    .map{mtch => (mtch.group("appName"), mtch.group("versionNo"))}
    .foldLeft(Map[String, List[String]]()) {(collector : Map[String, List[String]], tuple : (String, String)) =>
        if (collector contains tuple._1) collector updated (tuple._1, (collector(tuple._1) :+ tuple._2))
        else collector updated (tuple._1, List(tuple._2))
    }
    .toList
    .sortBy{tuple => tuple._1}
    .foreach{tuple => println("App: %s, Latest Update: %s" format (tuple._1, tuple._2.max))}
```


- Scala was mainly designed by a team lead by Martin Odersky from 2001 – 2004. He is also a main contributor of the development of Java's javac compiler.
- Scala is a multi paradigm language, i.e. it supports the functional paradigm, which is shown in the code above, and also other paradigms, esp. the imperative, procedural and object oriented paradigms. In the code above we use it like Clojure, which is a functional language, so it slightly resembles the Clojure code, but there are also some elements in the code, which resemble the original Java code.
  - Scala was strongly influenced by Haskell.
  - However, the core idea behind Scala is a mixture between the functional and object-oriented paradigm.
- In opposite to Clojure's dynamic typing, Scala provides a very mighty and expressive static type system:
  - The type system features a mighty type inference and pattern matching, which can make compilation taking longer.
  - This mighty inference is also required to put Scala's idea into effect, that every statement is an expression. E.g. if/else returns a value, whose type must be inferred.
  - Scala features a kind of multiple inheritance with so called traits.
  - It offers better support for generics than Java, esp. also reified generics, which leads to using special collection types in Scala, that are not present in Java. – This makes porting Scala to Java not very simple.
  - Also a macro system is part of Scala.
- This code looks like the desired result is described in a functional-programming-way, but the it is described with a series of tasks: getLines, then map, then map again, then foldLeft etc.
- In comparison to Java, Scala has more features and is more complex.
  - Even object-orientation is a bit ahead in Scala, esp. JVM and language-elements like packages are represented as objects.
  - As to Mr. Odersky the idea is to use some (esp. more advanced) features as a library-developer and other features rather as library-consumer.
    - => The Play framework is a full-stack framework, which was written in Scala, but can be used by any JVM-language.

Phase 5: Program Test

- After having the coded processing approach and the code also compiles successfully, we have to check the program.
- The test-input data should cover extreme and edge cases, which make up critical data:
  - wrong data, wrong/unexpected format
  - "Update Software.log" is empty or very large
  - infrastructure problems: "Update Software.log" not existent, or we have no read access
- To check the program, we can use value tables and the code to make dry runs.
  - With a dry run we check the correctness of a program mentally, with a value table. I.e. we do not use a computer!
- Then the real software product will be tested on a real PC:
  - Tests on the supported operating systems (OS).
  - Different input data, incl. extreme and edge cases under "real" conditions.
  - In this phase of testing it is important, that people different from the original implementors perform the tests.
  - => Such tests are usually done by special personnel of the quality assurance department.

Phase 5: Program Test – Dry Run and Value Table – Part 1

- Now we'll make a dry run of this simplified part of our program (e.g. w/o output, but only the processing algorithm):

```
1:    for (String line : Files.readAllLines(Paths.get(args[0]))) {
2:          Matcher matcher = parseRex.matcher(line); // Extracts the app name and version from line.
3:          if (matcher.matches()) { // Checks, if the extraction of app name and version was successful
4:                String key = matcher.group("appName");
5:                String value = matcher.group("versionNo");
6:                if (!latestUpdates.containsKey(key) || 0 < value.compareTo(latestUpdates.get(key))) {
7:                      latestUpdates.put(key, value);
8:                }
9:          }
10:   }
```

- Input values: a smaller version of the "Software Update.log" file with these two lines:

```
2009-04-06 12:43:04 +0200: Installed "GarageBand Update" (5)
2009-04-06 12:44:34 +0200: Installed "iMovie Update" (8)
```

- Following the program execution would result in such a value table.

Executed Program Line	Variables	Values	Conditions
1	*line*	"2009-04-06 12:43:04 +0200: Installed \"GarageBand Update\" (5)"	
4, 5	*key* *value*	"GarageBand Update" "5"	
6	*latestUpdates*	{empty}	*latestUpdates.containsKey(key)* is false
7	*latestUpdates*	{"GarageBand Update" : "5"}	
1	*line*	"2009-04-06 12:44:34 +0200: Installed \"iMovie Update\" (8)"	
4, 5	*key* *value*	"iMovie Update" "8"	
6	*latestUpdates*	{"GarageBand Update" : "5"}	*latestUpdates.containsKey(key) is* false
7	*latestUpdates*	{"GarageBand Update" : "5", "iMovie Update" : "8"}	

Phase 5: Program Test – Dry Run and Value Table – Part 2

- Benefits
  - It could be done without a computer.
  - Can be used for documentation.
- Downsides
  - It is a lot of work! E.g. a line of code could do many algorithmic things at once.
  - Esp. syntactic errors can almost not be spotted.
- Today we rather use:
  - (1) Log/trace messages, which print the values of variables of an algorithm to standard output, a file or a graphical user interface.
    - "printf()-debugging"
    - A downside of this approach is, that adding new messages requires to change the code and to be able to change the code at all.
  - (2) A debugger to see the values of an algorithm during it is running on the computer (i.e. no dry run, but a "wet" run).
    - A debugger allows line-by-line step-through execution of source code while the program is running, to examine or even modify variables.
    - There exist visual debuggers for IDEs and non-visual debuggers for console usage, e.g. on remote or customer machines.
    - Debuggers are easy and fast/spontaneous to use, esp. they work in "real time".
    - Most debuggers also allow to add logging messages, without changing the code at all!
  - Nowadays, log/trace outputs or debuggers (in IDEs) should be used in favor to dry runs and value tables.

- Some older Basic dialects allowed to turn on the "trace mode". Tracing then output the number of the executed line on the screen respectively. – Tracing was esp. needed to check the correctness of gotos, which tended to be incorrect quite often. The trace mode was activated with the TRON (TRace ON) command, the term "TRON" is actually the inspiration of the science fiction movie "Tron" in 1982. (Tracing was deactivated with the TROFF (TRace OFF) command.)

Phase 6: Program Documentation

- Technically, for other developers to allow maintenance and further development:
  - The initial user story.
  - The symbolic program code itself.
  - Flowcharts and NSDs.
  - Test plans, test input data

- User-facing documentations:
  - Installation documentation
  - User manual.

- => To be frank software engineering is a holistic activity and spans other phases below phase 1 and above phase 6.

Literature on Java

- There exists a plethora of books...

- The Java language:
  - Bruce Eckel, Thinking in Java
  - Christian Ullenboom, Java ist auch eine Insel (German standard literature)
  - Christian Ullenboom, Java SE 9 Standard-Bibliothek (German standard literature)
  - "The Java Tutorials" from Oracle: https://docs.oracle.com/javase/tutorial/

- Computing in general:
  - Charles Petzold, Code
  - Rob Williams, Computer System Architecture

Thank you!