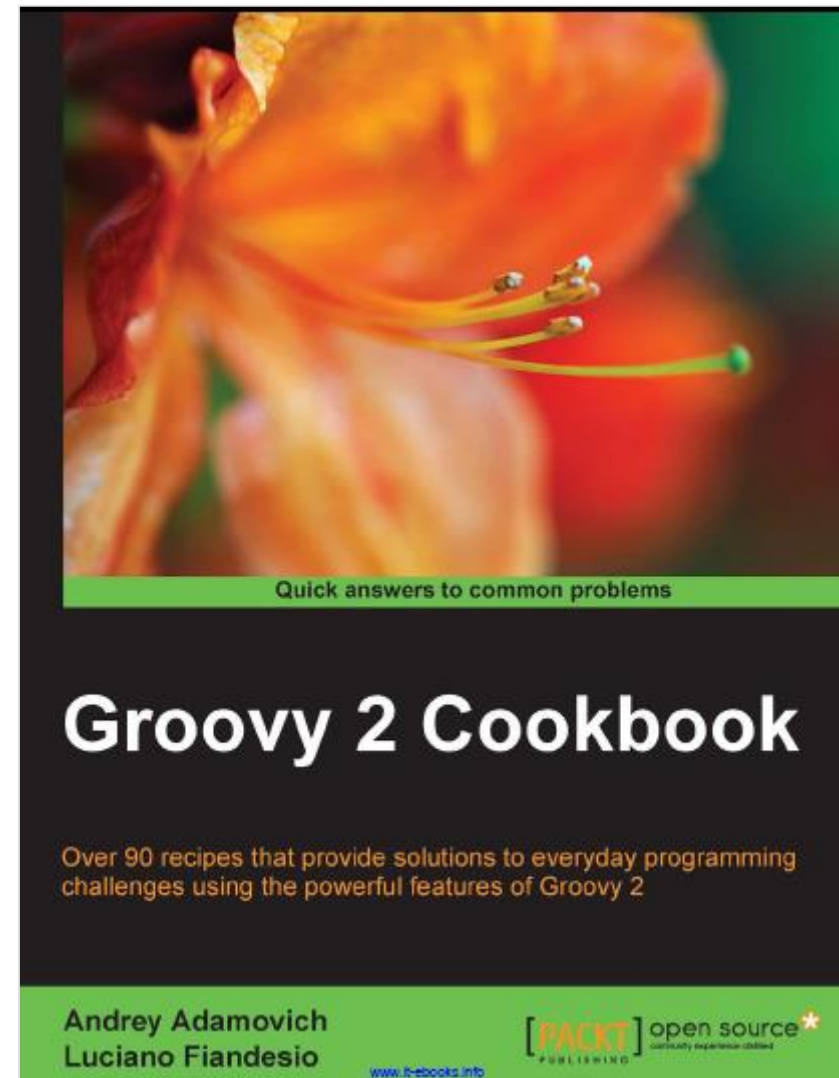
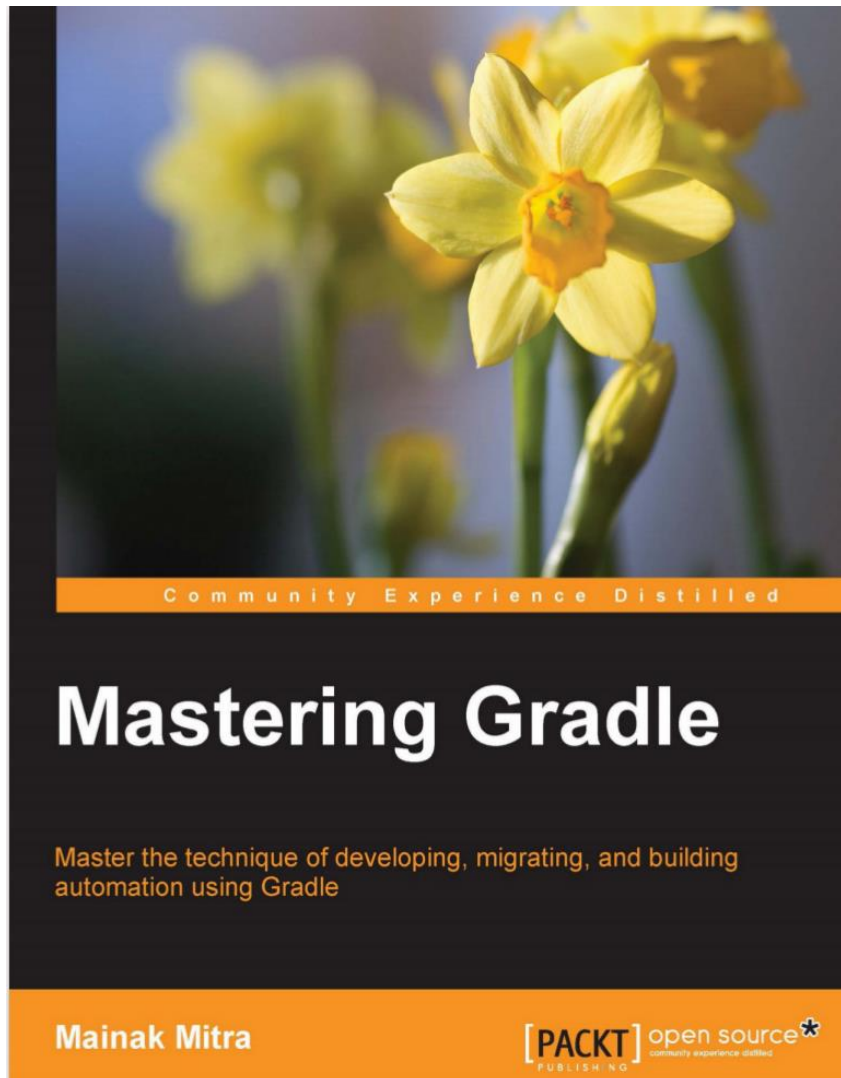


Gradle und Groovy

Ein Seminar für G & D

15.-16.10.2018 in München

Referent: Rainer Sawitzki



- Die in diesem Seminar verwendete Werkzeuge und Frameworks sind Open Source
 - LPGL Lizenzmodell
- Dies ist ein Programmier-Seminar
 - Damit werden die Inhalte durch Übungen vertieft und verinnerlicht
 - Musterbeispiele werden zur Verfügung gestellt
 - Diese können am Ende des Seminars als ZIP-Datei kopiert werden
 - USB-Stick oder ähnliches
- Dokumentation und Ressourcen stehen auch im Internet zur Verfügung
 - Beispiele unter <https://GitHub.com/Javacream/org.javacream.training.gradle>
- Konventionen
 - Befehle werden in `Courier-Schriftart` dargestellt
 - Dateinamen werden in *kursiver Courier-Schriftart* dargestellt
 - Links werden in unterstrichener Courier-Schriftart dargestellt

© Javacream

Javacream

Dr. Rainer Sawitzki

Alois-Gilg-Weg 6

81373 München

eMail: training@rainer-sawitzki.de

Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.

Einführung Buildmanagement	6
Einführung in Gradle	35
Einführung in Groovy	45
Programmieren in Groovy	61
Fortgeschrittene Konzepte in Groovy	100
Gradle im Detail	124

1

EINFÜHRUNG BUILDMANAGEMENT

1.1

BEGRIFFE UND DEFINITIONEN

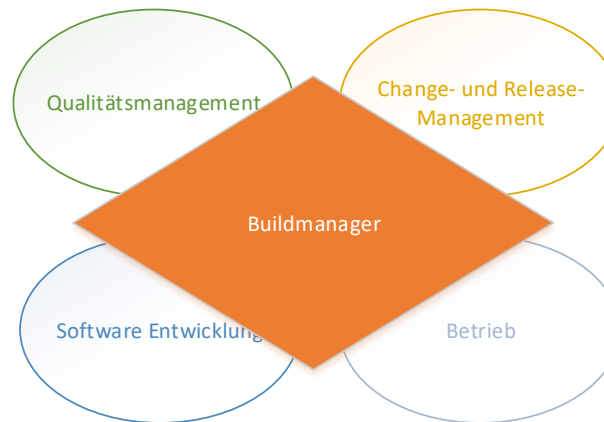
- Unter Buildmanagement verstehen wir alles, was dazu nötig ist aus einer Reihe von Werken ein oder mehrere Artefakte zu erzeugen. Dieser Vorgang soll dabei
 - automatisch
 - korrekt
 - und mit minimalem Aufwand
 - erfolgen.
- Buildmanagement umfasst dabei Werkzeuge, Infrastruktur sowie Prozesse und Verfahren und bildet eine Schnittmenge zu den verwandten Gebieten Release-Management und Qualitäts-Management.

**Paketieren und
Ausliefern von
(Release-)Versionen**

**Konfigurieren der
Zielumgebungen**

**Deployen der
Applikation auf den
Zielsystemen**

**Pflege und
Paketierung von
Datenbank-Skripten**



**Einbeziehen des
Betriebes zur
Entwicklungszeit**

Agile Prozesse

**Virtualisierungs- und
Cloudbasierte
Ansätze**

**Automatisierung von
Datacentern und
Konfigmanagement**

Projekt Buildmanager

- Verantwortlich für ein einzelnes Projekt
- Erlaubt individuelle Lösungen
- Erschwert Vereinheitlichung

Zentraler Buildmanager

- Projektübergreifend
- Bündelung von Know-How
- Knappe Ressource
- Individuelle Probleme werden nicht gewürdigt

Buildverantwortlicher

- „Wissender“ Entwickler
- Hat ausreichend Wissen, um das Tagessgeschäft umzusetzen
- Entlastet (vor allem den zentralen) Buildmanager

Werke und Artefakte

- Einzelne Dateien

Hilfswerkzeuge

- Generatoren/Konverter/Compiler
- Kommandos

Buildwerkzeug

- Ant/Maven/Gradle

Build Infrastruktur

- Buildserver
- Artifact-Repository

Build(-prozess)

Build-Schritt

- Ant: Target, Maven: Phase, Gradle: Task

Deployment

Projekt

Modul

1.2

DAS BUILDWERKZEUG

- Complete
- Repeatable
- Informative
- Schedulable
- Portable

- Nach dem Start des Buildprozesses ist kein Eingreifen mehr erforderlich.
- Anti-Beispiel
 - „Nach dem Kompilieren kopieren Sie die Datei y in das Verzeichnis z und starten dann den Packager.“

- Der Buildvorgang kann jederzeit (auch nach Jahren!) wieder gestartet werden
 - Das Ergebnis ist identisch
 - „Build-Zeitmaschine“
- Jede externe Abhängigkeit muss vollständig spezifiziert sein
- Jedes Endartefakt muss
 - eindeutig und leicht zu findend sein
 - und einen exakten Versionsstand beinhalten

- Informationen über jeden Build-Vorgang
 - Erfolgreich / Fehlgeschlagen
 - Gründe für den Fehlschlag
 - Außerdem
 - Ergebnisse der Unit-Tests
 - Metriken
 - Zyklomatische Komplexität (CC)
 - Afferent / Efferent Coupling
 - NCSS (Non-Commenting Source Statements)
 - Code Coverage
 - Statische Code Analysen (FindBugs, PMD, CheckStyle, Simian...)

- Der Buildprozess muss automatisiert startbar sein
- Start über die Kommandozeile
 - Gegenbeispiel: IDE-Builds

- Kein harten Environment-Abhängigkeiten
- Keine IP-Adressen
- Keine harten Dateipfade
- Muss auf jedem normalen Rechner laufen
 - Gegenbeispiel: „Magic Build Machine “
- Native Builds sind leider in der Regel nicht portabel

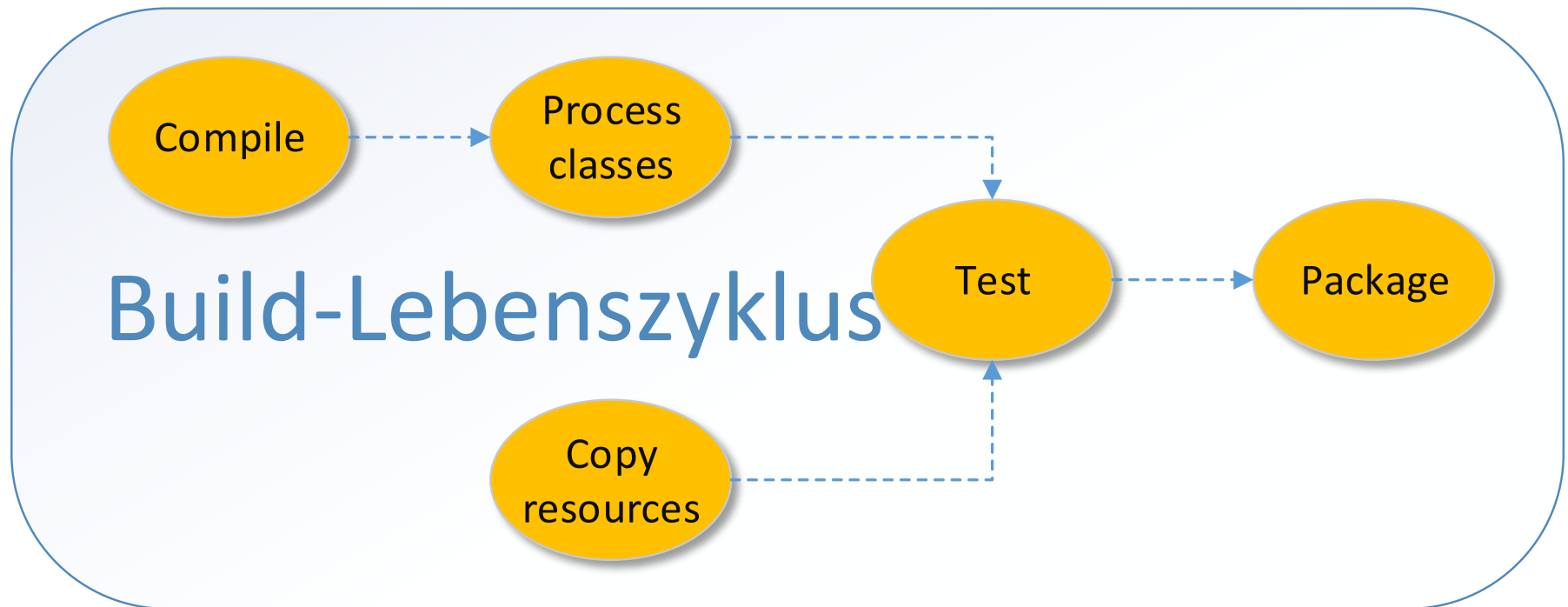
1.3

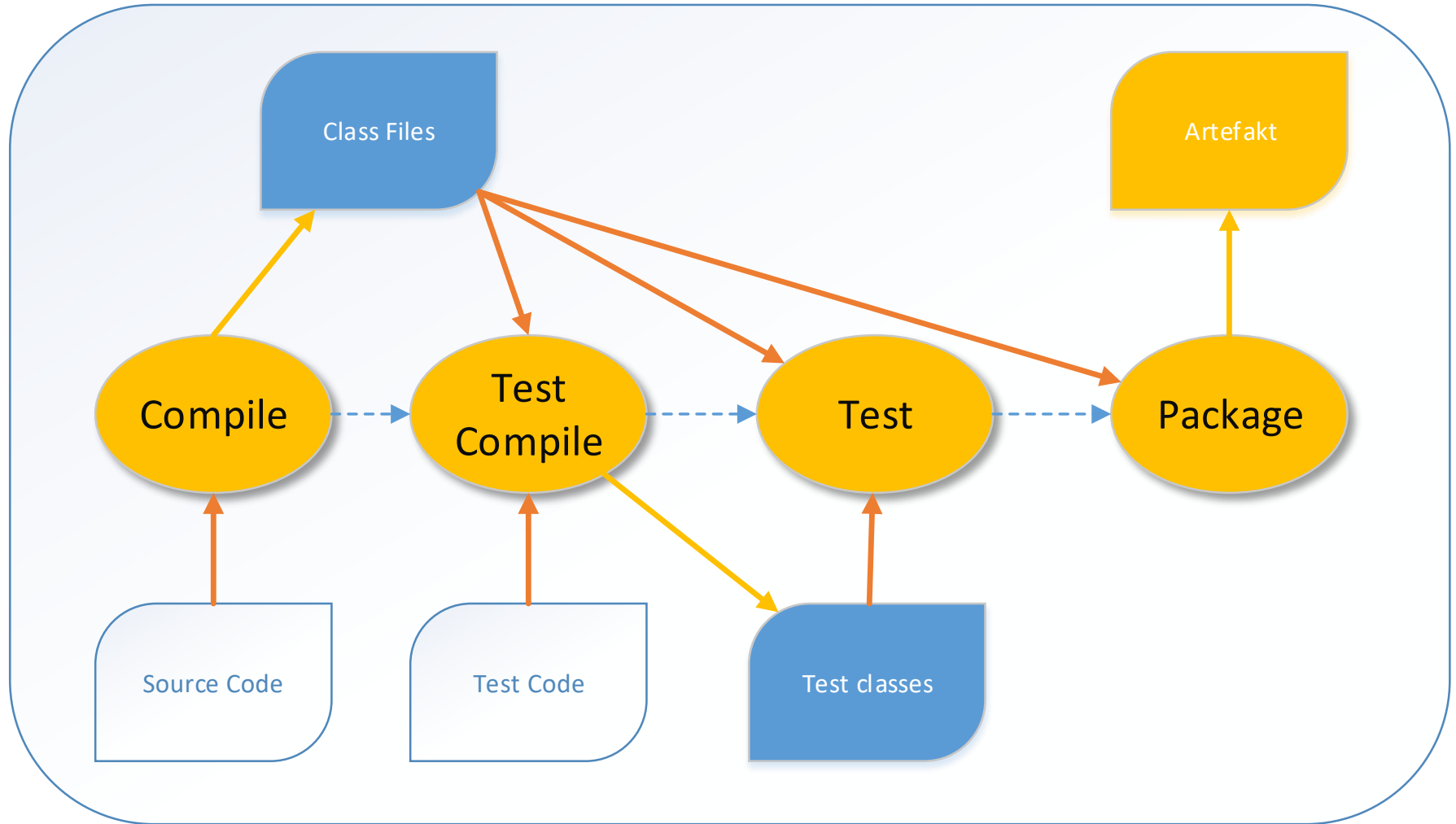
BUILDWERKZEUGE

- Der Buildvorgang wird programmiert
 - Shell-Skript
 - Make
 - Ant
 - Gradle

- Das Buildwerkzeug selbst definiert einen prototypischen Buildvorgang
- Jedes Projekt konfiguriert nur bestimmte Punkte:
 - Attribute (Projektname, Ergebnistyp)
 - Abweichungen vom Standard (Verzeichnisse)
 - Definierte Einsprungpunkte
- Apache Maven
- Gradle

- Damit gelten dafür die gleichen Regeln wie für jede andere Software auch
 - Wartbarkeit
 - Wiederverwendbarkeit
 - Keine Redundanzen
 - Ablage im Versionsverwaltungssystem

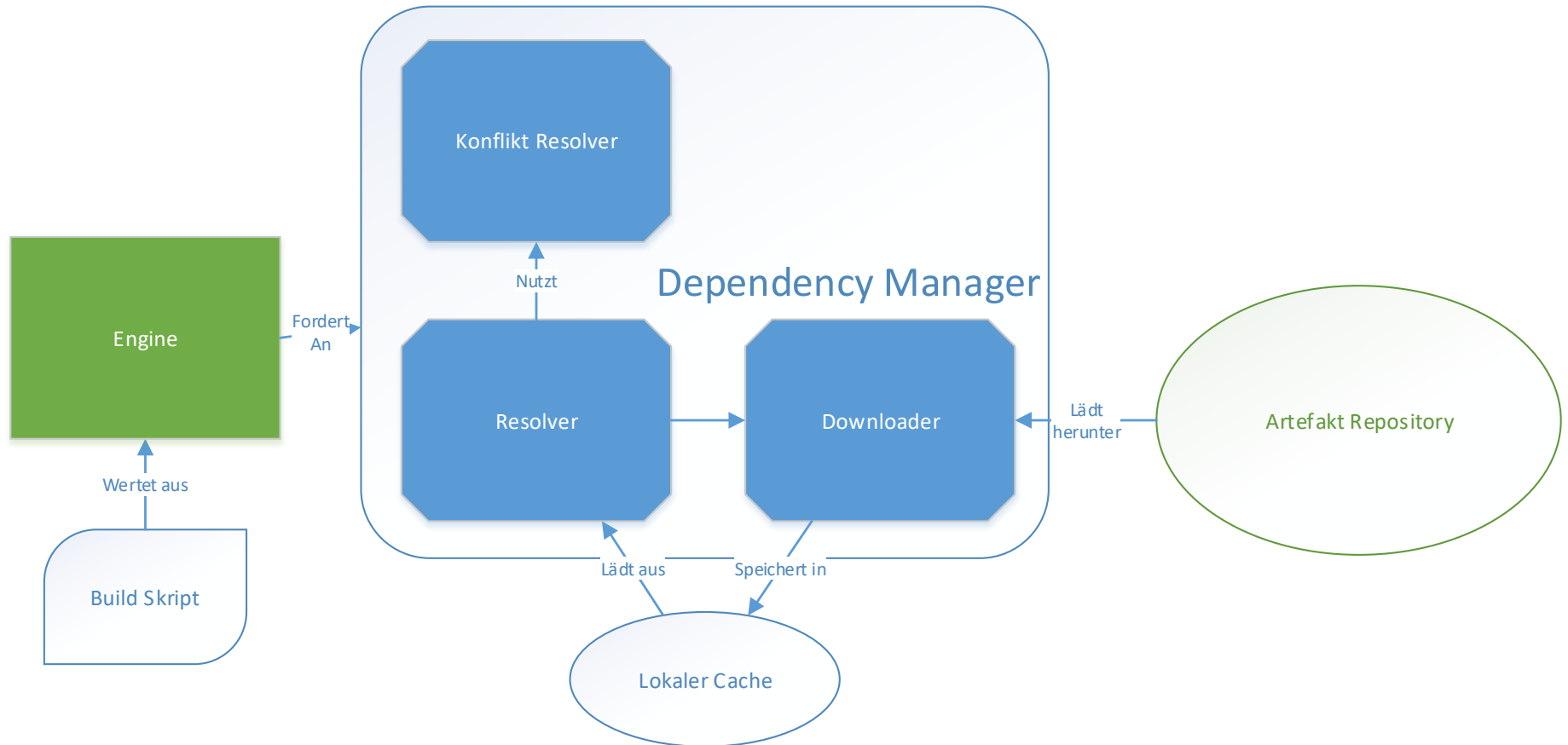




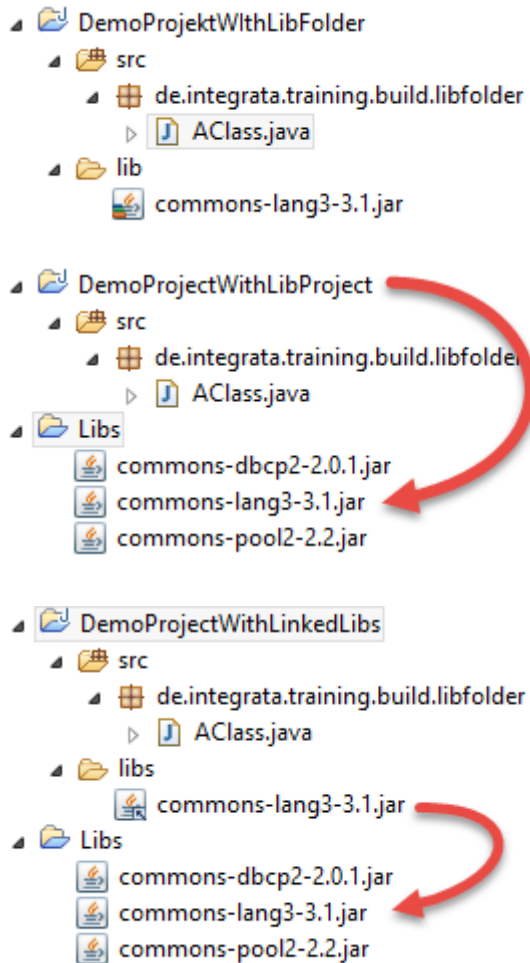
1.4

DEPENDENCY MANAGEMENT

Der Dependency Manager



- Lib-Folder
 - Abhängigkeiten werden mit dem Projekt eingecheckt
- Externe Lib-Folder
 - Abhängigkeiten werden in ein separates „Lib“ Projekt eingecheckt
- Verlinkte Libraries
 - Das Projekt selbst enthält Links auf die Abhängigkeiten im Lib Projekt
 - Für den Buildprozess sieht es aus, als sei es ein Lib-Folder



- Projekt enthält eine Informations-Quelle, in der die Abhängigkeiten eindeutig definiert sind
- Dazu wird eine Koordinate definiert
 - Gruppe
 - Häufig ein Domänen-Name
 - Artefakt-Name
 - Anwendungs-spezifisch
 - Version
- Der Dependency Manager lädt die Abhängigkeiten
 - aus einem lokalen Repository
 - ein Cache
 - oder, falls dort nicht vorhanden, von einem Repository-Server
 - Unternehmens-intern oder Interne

- **Maven**

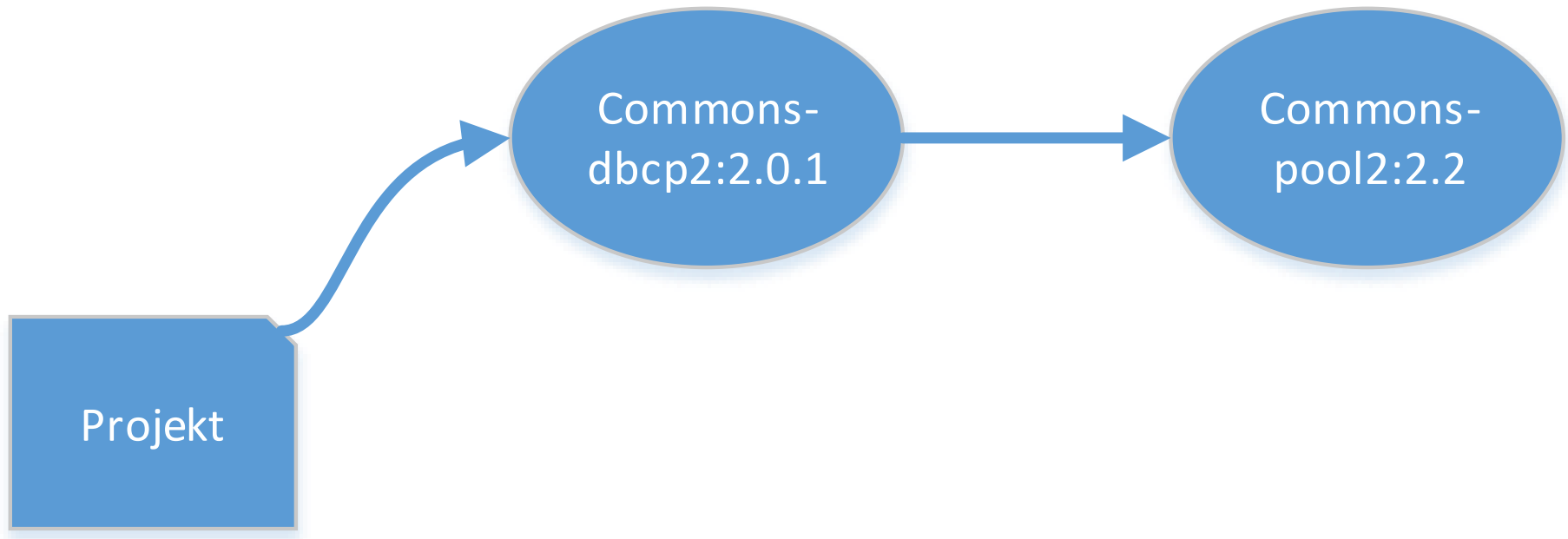
```
<dependency>  
  <groupId>com.google.guava</groupId>  
  <artifactId>guava</artifactId>  
  <version>15.0</version>  
</dependency>
```


- **ANT/Ivy**

```
<dependency org="com.google.guava" name="guava" rev="15.0">  
/dependency>
```

- **Gradle**

```
compile group: 'com.google.guava', name: 'guava', version:  
'15.0'
```



1.3.2-RC4

2

EINFÜHRUNG IN GRADLE

2.1

ÜBERBLICK

- Gradle ist ein konfiguratives Build-Werkzeug
- Die Build-Konfiguration ist hierarchisch gegliedert
 - Projekte
 - haben Eigenschaften
 - und gruppieren
 - Tasks
 - die wiederum
 - Attributes und Actions definieren
- Tasks werden von in einem Rahmenwerk ausgeführt
 - Gradle Kommandozeile
 - `gradle <task>`
 - Plugins
 - definieren eine Lebenszyklus
 - dieser führt Tasks in einer PlugIn-spezifischen Sequenz aus

2.2

INSTALLATION

- Standalone
 - Download und Installation einer Plattform-spezifischen Gradle-Distribution
- Integration in Entwicklungswerkzeuge
 - Plugins für Eclipse, Groovy, ...
- Die Benutzung von Gradle erfolgt dann entweder direkt über die Konsole oder die Benutzer-Führung der IDE

2.3

EIN ERSTES BEISPIEL

- **Das Skript**

```
task helloGradle doLast {  
    println 'Hello Gradle'  
}
```

- **Konsolen-Aufruf**

```
gradle helloGradle
```

- **Ausgabe**

```
> Task :helloGradle  
Hello Gradle
```

```
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed
```

- **Das Skript**

```
task helloGradle doLast {  
    println 'Hello Gradle'  
}
```

- **Konsolen-Aufruf**

```
gradle goodbyeGradle
```

- **Ausgabe**

```
FAILURE: Build failed with an exception.
```

```
* What went wrong:
```

```
Task 'goodbyeGradle' not found in root project 'first'.
```

```
* Try:
```

```
Run gradle tasks to get a list of available tasks. Run with -  
-stacktrace option to get the stack trace. Run with --info or  
--debug option to get more log output. Run with --scan to get  
full insights.
```

```
* Get more help at https://help.gradle.org
```

```
BUILD FAILED in 0s
```

```
<-----> 0% WAITING
```

- **Das Skript**

```
task helloGradle duLast {  
    println 'Hello Gradle'  
}
```

- **Konsolen-Aufruf**

```
gradle helloGradle
```

- **Ausgabe**

```
FAILURE: Build failed with an exception.
```

```
* Where:
```

```
Build file
```

```
'/home/rainer/git/org.javacream.training.gradle/org.javacream.training.gradle.basics/scripts/first/build.gradle' line: 1
```

```
* What went wrong:
```

```
A problem occurred evaluating root project 'first'.
```

```
> Could not find method duLast() for arguments
```

```
[build_7v885tnekjw91891xp40y359k$_run_closure1@216e4324] on task  
'helloGradle' of type org.gradle.api.DefaultTask.
```

```
* Try:
```

```
Run with --stacktrace option to get the stack trace. Run with --info or --  
debug option to get more log output. Run with --scan to get full insights.
```

```
* Get more help at https://help.gradle.org
```

```
BUILD FAILED in 0s
```

- Syntax des Skripts
 - Was ist das für eine Sprache?
- gradle-Aufrufskript
 - Aufrufoptionen?
 - Defaults?
- Ausgabe
 - Format?
 - Interpretation der Fehler-Meldungen?

3

EINFÜHRUNG IN GROOVY

3.1

HINTERGRUND

- Codehaus
 - Die Open Source Community ist bei der Weiterentwicklung von Groovy maßgeblich beteiligt
 - Offizielle Web Seite groovy.codehaus.org
- Pivotal
 - Spin-out unter Beteiligung von VmWare
- Seit 2016 ist Groovy ein Apache Projekt

3.2

EINSATZBEREICHE

- Skriptsprache
- Ergänzung zur Programmiersprache Java
- Vollwertige eigene Programmiersprache

- Skriptsprachen sind für eine schnelle und agile Softwareentwicklung geeignet
 - "Shell-Skripte"
 - Build-Prozesse
 - Testabläufe
- Weiterhin wird Groovy wird von Produktlösungen als integrierte Skript-Sprache benutzt
 - Reporting
 - Jenkins-Pipelines
 - Abfragesprache für Datenbanken

- Wichtig
 - Grundlegende Syntax
 - Datentypen
 - Zeichenketten
 - Zahlen
 - Logische Werte
 - Zeichenkettenverarbeitung
 - Rechnen
 - Datum und Uhrzeit
 - Arbeiten mit Daten-Containern, z.B. Listen
 - Dateibasierte Ein- und Ausgabe
- Eher unwichtig
 - OOP-Konzepte
 - Dynamische Erweiterungen

- Groovy und Java sind komplett interoperabel
 - Jede Java-Klasse kann in Groovy benutzt werden und umgekehrt
- Zusatz-Features von Groovy:
 - Erweiterungen der Java-Standardklassen
 - Einfache Realisierung eines untypisierten Programmiermodells
 - Dynamische Erweiterungen aller Klassen ohne Vererbung
 - "Open Classes"
- Java 8 hat einige Groovy-Features übernommen
 - Allerdings ist Groovy damit nicht obsolet

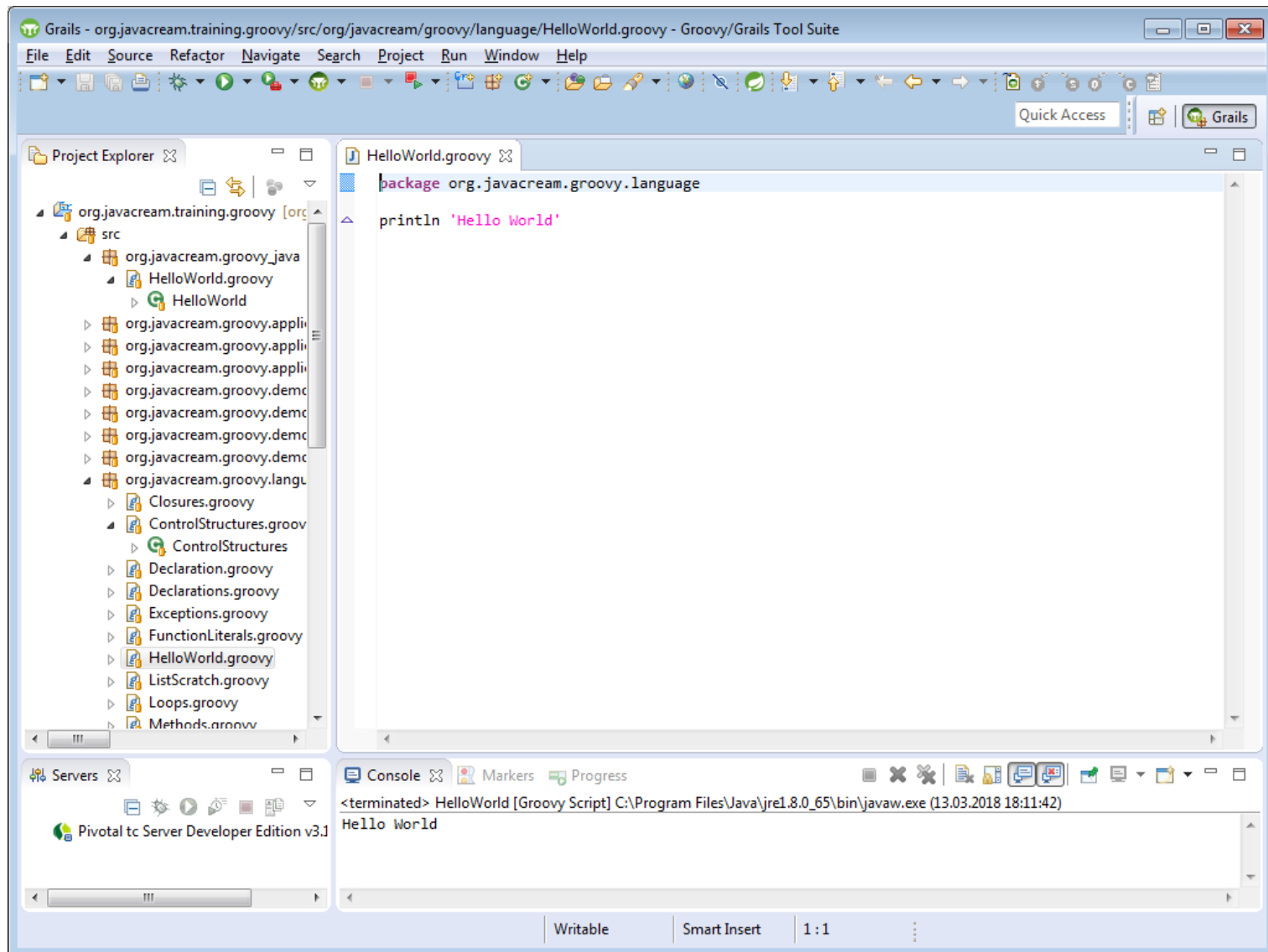
- Wichtig
 - Erweiterungen der Java-Standardklassen
 - Groovy-Bibliotheken
 - Ergänzende Ansätze der OOP
 - Untypisiertes Programmiermodell
 - Für Java-Versionen kleiner 8: Funktionale Programmierung
- Fast identisch zu Java und damit einfach
 - Elementare Syntax
 - Typ-System
 - Klassen-Definition

- Hier sind die wichtigen Inhalte beider vorheriger Ansätze gleichermaßen relevant

3.3

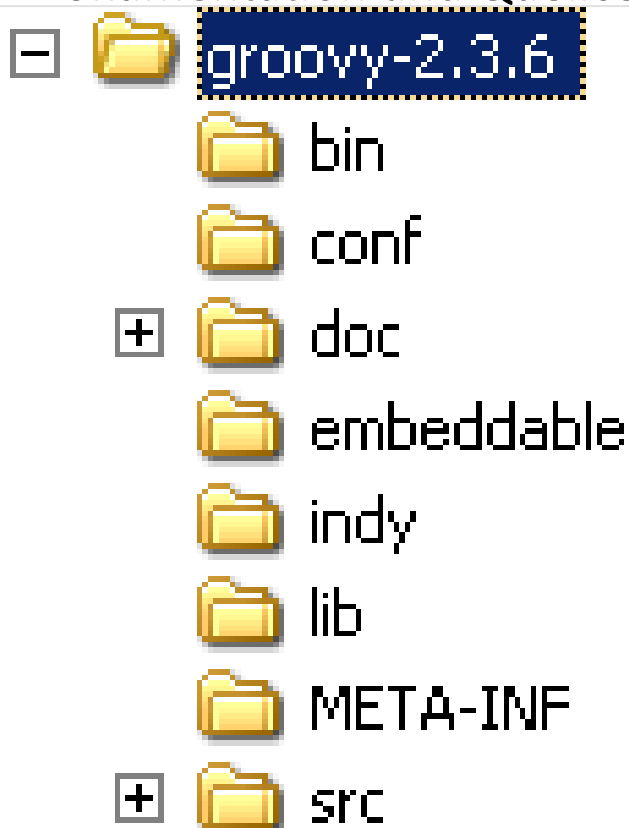
INSTALLATION UND WERKZEUGE

- Plugins für
 - Eclipse
 - IntelliJ
- Komplettpaket Groovy/Grails Tools Suite (GGTS)
 - ein Projekt der Spring-Community
 - basierend auf Eclipse
 - Groovy ist in der Distribution enthalten



- Notwendige Vorbedingung: Java Runtime
 - Damit ist die Groovy-Installation Plattform-unabhängig
- Die Groovy-Distribution wird als ZIP-Datei verteilt
 - Die eigentliche Installation erfolgt damit über ein simples Entpacken
 - Keine Administrator-Rechte etc. notwendig

- Die Groovy-Bibliotheken als Java-Archive (.jars)
- Einige Hilfswerkzeuge
- Dokumentation und Quellcodes



- `grape.bat`
 - Dynamisches Nachladen von benötigten Bibliotheken
- `groovy.bat`
 - Ausführen einer Groovy-Anwendung
- `groovyc.bat`
 - Aufruf des Groovy-Compilers
- `groovyConsole.bat`
 - Start der grafischen Groovy-UI
- `groovydoc.bat`
 - Erzeugen der Groovy-Dokumentation
- `groovysh.bat`
 - Start der interaktiven Groovy-Shell
- `java2groovy.bat`
 - Konvertieren einer Java-Anwendung nach Groovy
- `startGroovy.bat`
 - Allgemeines Start-Skript für alle anderen Skripte
 - Startet einen Java-Prozess

4

PROGRAMMIEREN IN GROOVY

4.1

GRUNDLAGEN DER PROGRAMMIERUNG

- Einfache Textdateien
 - Für die Programmierung mit Groovy kann damit theoretisch jeder beliebige Text-Editor benutzt werden
- Dateiendung .groovy
- Zur besseren Lesbarkeit kann das Skript fast beliebig formatiert werden
 - Leerzeichen
 - Tabulatoren
 - Absätze

- Liegen im master-Branch des Git-Repositories
 - <https://github.com/JavacreamTraining/org.javacream.training.groovy>
 - Das Repository kann gecloned oder als Zip-Datei geladen werden
- Alle Themenbereiche dieses Abschnitts werden in einem jeweiligen Skript dargestellt
 - Abgelegt im Ordner `scripts`

- **Kommentare**
 - `//Einzeilig`
 - `/* Potenziell Mehrzeilig */`
 - `/** Groovy-Doc-Kommentar */`
- **Blöcke**
 - Anweisungen können durch geschweifte Klammern `{...}` zu einem Block gruppiert werden
 - Blöcke definieren einen Scope/Gültigkeitsbereich für Variablen

4.2

VARIABLEN UND LITERALE

- Vor der Verwendung einer Variable, eines Parameters oder einer Funktion muss dies bekannt gemacht werden
 - Untypisiert mit `def`
 - `def myVar`
 - `def myFunc() {...}`
 - Typisiert mit Type-Angabe
 - `String myVar`
 - `String myFunc(...) {return "Hello World!"}`
 - `def myVar = 'B' as String`

- Einfache Anführungszeichen definieren einen normalen String
 - `' Hello '`
 - `''' Multiline '''`
- Doppelte Anführungszeichen unterstützen die String-Interpolation
 - `" Hello ${name}"`
 - `" Hello ${person.info()}"`
 - `" Hello $name.property "`
 - `""" Multiline """`
- Slashy Strings und Dollar slashy Strings benötigen kein Escaping von speziellen Zeichen
 - und sind damit für reguläre Ausdrücke geeignet
 - `/ Hel\ lo /`
 - String Interpolation wird unterstützt
 - Slashy und Dollar slashy Strings sind mehrzeilig

- **Angabe**
 - **dezimal**
 - 4.2
 - **hexadezimal**
 - 0xC7
 - **oktal**
 - 077
 - **binär**
 - 0b11001001
- **Unterstützte Typen sind**
 - `byte, char, short, int, long, java.lang.BigInteger`
 - `float, double, BigDecimal`
- **Unterscheidung implizit während der Zuweisung oder durch nachgestelltes Kürzel**
 - 42f
 - 122s

- Datentyp `boolean`
 - `true`
 - `false`
- Groovy Truth
 - Auch andere Datentypen können als logischer Wert interpretiert werden
 - `boolean thatsTrue = "Hello"`
 - `boolean thatsFalse = ""`
 - `boolean thatsTrue = 42`
 - `boolean thatsFalse = 0`

- Zuweisungsoperator

=

- Arithmetische Operatoren

a + b

Addition

a - b

Subtraktion

a * b

Multiplikation

a / b

Division

a % b

Modulo-Division (nur für Ganzzahlobjekte erlaubt)

- Kurzschreibweise

a += b

a -= b

a *= b

a /= b

a %= b

■ Vergleichsoperatoren

> < >= <= != ==

■ Logische Operatoren

&& Und (binär)

|| Oder (binär)

! Nicht, logische Negation (unär)

■ Bit-Operatoren

>>

>>>

<<

&

|

^

~


```
if (boolescher Ausdruck) {  
    Anweisungen  
    Anweisungen  
}  
  
else {  
    Anweisungen  
    Anweisungen  
}
```

- Groovy interpretiert die folgenden Bedingungen als `true`:
 - Nicht-leere Zeichenkette
 - Collections mit mindestens einem Element
 - Variablen, denen ein Wert zugewiesen wurde

```
switch( ganzzahliger oder char-Ausdruck oder auch  
    String )  
    {  
        case Konstante1:  
            Anweisung1.1  
            Anweisung1.n  
        case Konstante2:  
            Anweisung2.1  
            Anweisung2.n  
        default:  
            Anweisungd.1  
            Anweisungd.n  
    }
```

```
while( boolescher Ausdruck )  
    Anweisung  
  
do  
    Anweisung  
while( boolescher Ausdruck );
```

```
for( Ausdruck1;  Ausdruck2;  Ausdruck3 )  
    Anweisung
```

- Abbrüche mit `break` `continue`
- Hinweis: Iterationen über Collections werden in Groovy jedoch in den allermeisten Fällen über eine Closure formuliert
 - Kommt später

4.3

FUNKTIONEN

- Funktionen haben
 - einen Namen
 - eine Liste von Parametern
 - einen Block mit der Funktions-Implementierung
 - darin können beliebige weitere Elemente deklariert werden
 - Insbesondere lokale Variable
 - Die Parameter stehen innerhalb des Blocks als Variable zur Verfügung
 - einen Rückgabewert
 - Dieser wird mit return zurück gegeben und damit die Funktion abgeschlossen

Beispiel: Eine einfache Funktion

```
def myFirstFunction(def param1, def param2) {  
    def result = "OK"  
    println ("calling myFirstFunction, param1=${param1}",  
    param2=${param2}")  
    return result  
}
```


- Eine Funktion kann an beliebiger Stelle aufgerufen werden
- Dabei werden die Parameter an die Funktion übergeben
- Die Übergabe-Parameter stehen dann innerhalb der Funktion als Parameter zur Verfügung

```
def application() {  
    def number = 42;  
    def number2 = 21;  
    def result  
    result = checkNumberIsEvenOrOdd(number)  
    println resultFrom  
    result = checkNumberIsEvenOrOdd number2  
    println resultFromDemo  
}  
  
def checkNumberIsEvenOrOdd(def numberToCheck) {  
    def result = (numberToCheck%2 == 0)  
    return result  
}
```

4.4

DATEN-CONTAINER

- `List`
 - Listen enthalten Elemente, die über einen Index angesprochen werden
 - Der Index beginnt bei 0
- `Map`
 - Auch bekannt als "Dictionary" oder "assoziatives Array"
 - Maps enthalten Elemente, die über einen Key-Wert angesprochen werden
- `Set`
 - Eine Menge von Elementen ohne Duplikate
 - Ein Set hat keine innere Ordnung und deshalb keinen Zugriff über Index oder Schlüssel
- `Collections` ist der Überbegriff für `List`, `Map` und `Set`

```
def list = [element1, element2, element3]
list[0] // Ausgabe element1
list[2] //element3
//list[3] //Fehler
list[3] = "Neu"
list[3] //Neu
```

```
def colors = ['red': '#FF0000', 'green': '#00FF00',  
             'blue': '#0000FF']
```

```
Colors['blue'] // '#0000FF'
```

- Eine weitere Form von Listen sind Ranges
- Diese werden über das Range-Literal erzeugt

```
def range = 1..4
```

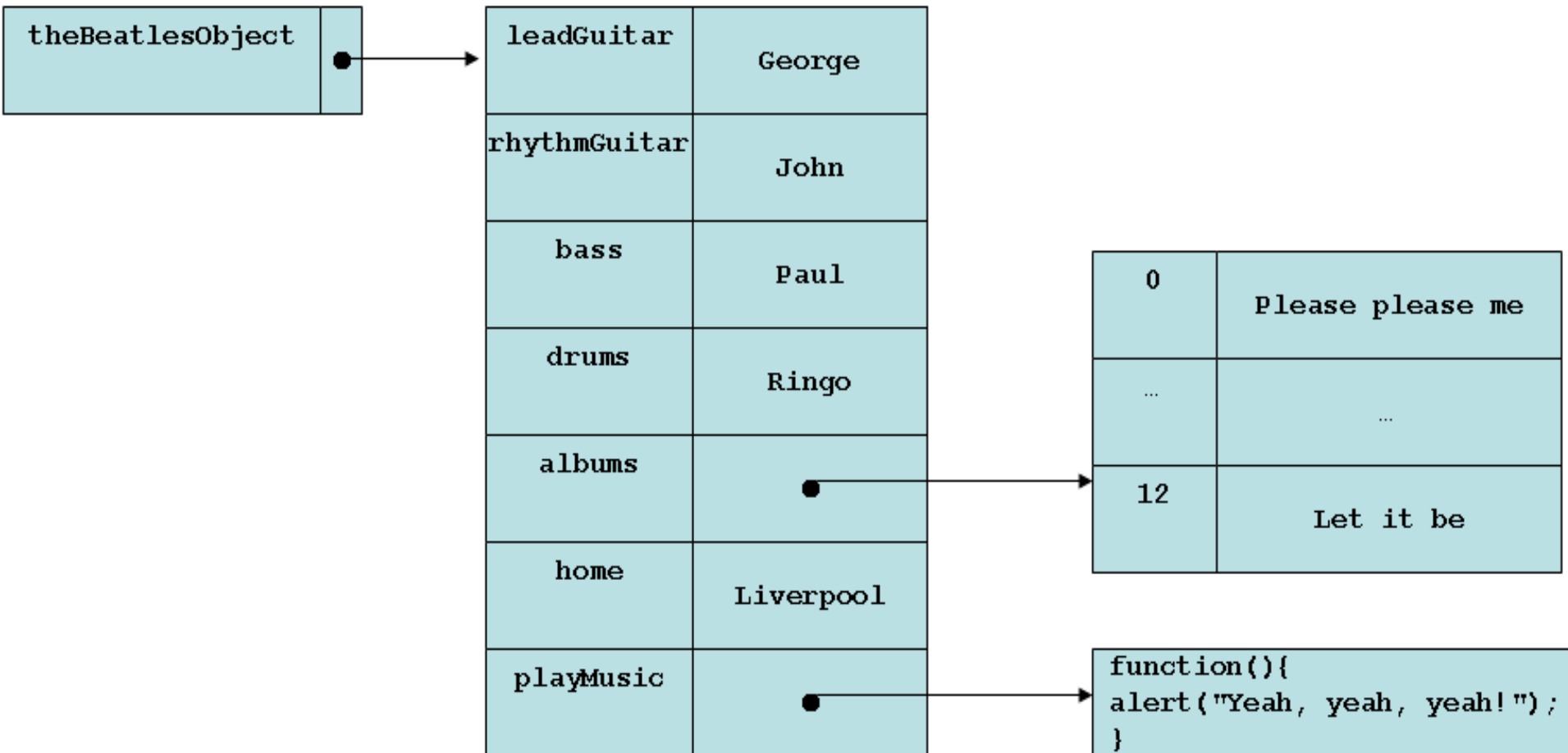
- Über Listen kann über den Index iteriert werden
- Problem:
 - Wie wird die Größe der Liste bestimmt, um einen Zugriff über die Grenzen des Arrays zu vermeiden?
- Lösung:
 - `list.size`
 - Diese Schreibweise wird im Anschluss detailliert erläutert


```
for (def i= 0; i < list.size; i++) {  
    println("Element ${i + 1}: ${list[i]}")  
}
```

4.5

REFERENZEN

- Alles in Groovy sind Objekte
 - und hat damit Attribute und Methoden
- Der Zugriff auf die Eigenschaften und Methoden erfolgt über den Punkt-Operator
 - `object.property`
 - `object.method()`
- Groovy legt alle Objekte im Heap-Speicher der Virtuellen Maschine ab
- Das Programm greift auf diese Informationen stets über eine Referenz zu
- Parameter und Rückgabewerte eine Funktion werden durch eine Kopie des Wertes der jeweiligen Referenz übertragen



4.6

FUNKTIONEN IM DETAIL

- Auch Funktionen sind Objekte

```
def myFunc = {x, y -> return x + y}  
println myFunc(1, 2)
```

- `myFunc` ist eine Referenz auf ein Objekt vom Typ `function`
 - und kann damit wie alle anderen Referenzen behandelt werden
 - In Zuweisungen
 - Als Parameter
 - Als Rückgabewert
- Funktionsobjekte werden beispielsweise bei der Verarbeitung von Collections exzessiv benutzt
 - Iteration
 - Filtern
 - Transformation

- Mit Hilfe der Funktionsobjekte können komplexere Situation entstehen:

```
def myFunc() {  
  def innerVar = 42  
  def innerFunc = {println ("From innerFunc: ${innerVar}")}  
  innerFunc()  
}  
myFunc()
```

- `innerFunc` hat Zugriff auf die Variable `innerVar`

- Dies bleibt immer noch gültig, wenn der Funktionsaufruf aus der Implementierung der äußeren Funktion herausgezogen wird:

```
def myFunc2() {  
    def innerVar = 42  
    def innerFunc = {  
        println ("From innerFunc2: ${innerVar}")  
    }  
    return innerFunc  
}  
def f = myFunc2()  
f()
```

- `innerVar` ist somit nicht, wie vermutet, eine lokale Variable der Funktion `myFunc2`, sondern eine Closure-Variable
 - Diese bleibt so lange gültig, bis keine Referenzen auf innerhalb von `myFunc2` definierten Funktionsobjekte mehr vorhanden sind

4.7

SYNTACTIC SUGAR

- Syntactic Sugar
 - soll die Menge überflüssiger Codezeichen verringern
 - die Lesbarkeit erhöhen
- Vorsicht: Syntactic Sugar kann auch zu sehr schwer erkennbaren Fehlern führen!

- Abschließende Semikolons sind optional
- Runde Klammern um die Parameter eines Funktionsaufrufs sind optional
 - Außer es ist zur Auflösung von Mehrdeutigkeiten notwendig
- Eine Klassendefinition ist optional
 - Sinnvoll für einfache Skripte
 - Die automatisch generierte Hüllklasse bekommt den Namen der Datei
- Einrückungen bzw. Whitespaces sind beliebig möglich
- Jede Funktion gibt implizit einen Wert zurück
- Eine Closure kann außerhalb der runden Klammern eines Funktionsaufrufs stehen, falls die Closure der letzte Parameter ist
 - Damit sind folgende Aufrufe äquivalent:
 - `list.findIndexOf(start, {element -> element.equals("X") })`
 - `list.findIndexOf start, {element -> element.equals("X") }`
 - `list.findIndexOf(start) {element -> element.equals("X") }`

5

FORTGESCHRITTENE KONZEPTE IN GROOVY

5.1

DIE KLASSENBIBLIOTHEK

- Groovy hat Zugriff auf alle Typen der zu Grunde liegenden Java-Runtime
 - Allerdings sind diese Klassen erweitert
 - Beispielsweise haben alle Collections eine Reihe von Funktionen, die mit Closures arbeiten
 - `list.each {element -> println element}`
- Die Groovy-Bibliothek enthält APIs und Hilfsklassen, die nicht Bestandteil der Java-Umgebung sind
 - Paket `groovy` und Subpakete
 - Diese sind Bestandteil der Groovy-Spezifikation
 - Paket `org.codehaus.groovy` und Subpakete
 - Groovy-Erweiterungen, deren API sich je nach Version ändern kann

All Classes

Packages

groovy.beans
groovy.grape
groovy.inspect
groovy.inspect.swingui
groovy.io
groovy.jmx.builder

All Classes

AbstractASTTransformation
AbstractASTTransformUtil
AbstractButtonProperties
AbstractCallSite
AbstractComparator
AbstractConcurrentDoubleKeyMap
AbstractConcurrentMap
AbstractConcurrentMap.Entry
AbstractConcurrentMap.Segment
AbstractConcurrentMapBase
AbstractConcurrentMapBase.Entry
AbstractConcurrentMapBase.Segment
AbstractFactory
AbstractFullBinding

Overview Package Class Tree Deprecated Index Help

Prev Next Frames No Frames

Groovy 2.3.6

Groovy - An agile dynamic language for the Java Platform
(JavaDoc for Java classes)

See: Description

Packages

Package	Description
groovy.beans	
groovy.grape	
groovy.inspect	Classes for inspecting object properties through introspection.
groovy.inspect.swingui	
groovy.io	Classes for Groovier Input/Output.
groovy.jmx.builder	
groovy.json	

5.2

EIGENE KLASSEN

- Eine Groovy-Klassendefinition wird durch das Schlüsselwort `class` eingeleitet
- Die im darauf folgenden Block deklarierten Variablen und Funktionen werden zu Attributen bzw. Methoden der Klasse
 - Standard: Attribute und Methoden stehen einer Instanz der Klasse zur Verfügung
 - `static`-Definitionen werden über die Klasse referenziert
- Mit `extends` kann eine Klasse eine Superklasse angeben
 - Von dieser werden die darin enthaltenen Attribute und Methoden geerbt
- Die Standard-Sichtbarkeit ist öffentlich/`public`
 - Daneben noch bekannt `private` und `protected`

- Mehrere Klassendefinitionen können in einer `.groovy`-Datei abgelegt werden
- Standardmäßig ein Konstruktor mit benannten Parametern erzeugt
 - Wird ein eigener Konstruktor geschrieben, wird dieser Standard-Konstruktor nicht mehr automatisch erzeugt!
- Standard-Imports
 - Klassen aus `java.util`, `java.io` etc sind automatisch importiert
 - Die vollständige Liste ist der Dokumentation zu entnehmen
- Annotations für Standard-Funktionen aus `groovy.transform`
 - `@ToString`
 - `@EqualsAndHashCode`
 - `@Canonical`
 - ...

- Die Operatoren in Groovy sind bis auf wenige Ausnahmen Syntactic Sugar:
 - `def result = 17 + 4`
 - ist vollkommen äquivalent zu
 - `def result = 17.plus(4)`
 - **oder** `def result = 17.plus 4`

Äquivalentes gilt für alle anderen Operatoren

Operator	Method	Operator	Method
<code>+</code>	<code>a.plus(b)</code>	<code>a[b]</code>	<code>a.getAt(b)</code>
<code>-</code>	<code>a.minus(b)</code>	<code>a[b] = c</code>	<code>a.putAt(b, c)</code>
<code>*</code>	<code>a.multiply(b)</code>	<code><<</code>	<code>a.leftShift(b)</code>
<code>/</code>	<code>a.div(b)</code>	<code>>></code>	<code>a.rightShift(b)</code>
<code>%</code>	<code>a.mod(b)</code>	<code>++</code>	<code>a.next()</code>
<code>**</code>	<code>a.power(b)</code>	<code>--</code>	<code>a.previous()</code>
<code> </code>	<code>a.or(b)</code>	<code>+a</code>	<code>a.positive()</code>
<code>&</code>	<code>a.and(b)</code>	<code>-a</code>	<code>a.negative()</code>
<code>^</code>	<code>a.xor(b)</code>	<code>~a</code>	<code>a.bitwiseNegative()</code>

- Implementieren eigene Klassen die entsprechenden Methoden, so werden damit die Operatoren unterstützt

```
class Money {  
    Money(amount) {this.amount = amount}  
    double amount  
    Money plus(Money other) {  
        Money money = new Money(this.amount + other.amount)  
        return money  
    }  
    Money minus(Money other) {  
        Money money = new Money(this.amount - other.amount)  
        return money  
    }  
}
```

- ...

```
Money money = money1 + money2
```

- Mit Hilfe der `@Delegate`-Annotation können Klassen auch ohne Vererbung Funktionen eines Delegations-Objekts bekommen
 - Diese Annotation ersetzt damit eine ganze Menge trivialer Methoden-Definitionen
 - Syntactic Sugar
- Beispiel

```
class MyList{
    @Delegate LinkedList delegate = new LinkedList()
    def myMethod(){println "called myMethod"}
}

class DelegateDemo {
    def delegateDemo(){
        MyList list = new MyList()
        list.add("element1")
        println list.size()
        list.myMethod()
    }
}
```

5.3

TRAITS

- Traits sind Interfaces
 - mit Implementierung und
 - Zustand
- Definition durch Schlüsselwort `trait`
- Verwendung in einer Klasse durch `implements`
- Traits ermöglichen damit die Mehrfachvererbung von Implementierungen


```
trait Addressable {  
    def city  
    def street  
    String getAddress() {  
        return "city: ${city}, street: ${street}"  
    }  
}
```

```
class TraitTest {
    @Test
    public void testTrait() {
        Person p = new Person(lastname: "Name", given_name: "Hans",
            city: "Frankfurt", street: "Stresemannallee 4")
        Hotel h = new Hotel(name: "park Inn", city: "Berlin", street:
            "Alexanderstrasse 18")
        println p.address
        println h.address
    }
}

class Person implements Addressable{
    def lastname
    def given_name
}

class Hotel implements Addressable{
    def name
}
```

- Traits unterstützen selbstverständlich Properties
- Traits können Bestandteil einer Hierarchie mit Vererbung sein
- Trait-Methoden können in der implementierenden Klasse überschrieben werden
- Traits können auch abstrakte Methoden deklarieren
- `this` bezieht sich innerhalb eines Traits auf die implementierende Instanz
- Konkurrierende Trait-Methoden
 - Werden in der Liste der implementierten Traits gleiche Signaturen gefunden, so wird die letzte Implementierung benutzt

5.4

DYNAMISCHE PROGRAMMIERUNG

- Die bisher benutzten Objekte wurden statisch definiert
 - Fester Satz von Attributen und Methoden
 - deklariert in einer Klasse
- Die Klassendefinition legt einen Typen fest
 - und zwar auf der Ebene des Quellcodes/Compilers
 - Werkzeugunterstützung
 - Compiler-Fehler
 - Autovervollständigung
- Der Typ ist zur Laufzeit unveränderbar
 - Jedes Objekt referenziert sein Klassenobjekt
 - Sicherlich praktisch, aber nicht immer:
 - `Student` und `Worker` sind `Personen`
 - Studierende Arbeiter?
 - Ein Student wird zum Arbeiter?

- Die Eigenschaften eines Objektes können sich zur Laufzeit beliebig ändern
 - Damit sind sowohl Attribute als auch Methoden gemeint
- Für dynamische Programmierung sind keine Klassendefinitionen notwendig!
 - Jedes Fachobjekt startet sein Leben als simples Objekt und wird im Laufe der Zeit so modifiziert, bis der gewünschte Zustand erreicht ist
 - Damit verschwimmt der Unterschied zwischen einer Map und einem Object!
 - `println book.isbn`
 - `println book["isbn"]`
- Duck Typing
 - *“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”*

- Hängt ab von
 - Fachliche Anforderungen
 - Programmierer-Ausbildung und damit Programmiersprache
 - Programmierer-Präferenz
- Groovy unterstützt beide Möglichkeiten
 - sogar mit fließendem Übergang!

- Der Zugriff auf Properties und Methoden eines Objektes erfolgt stets durch den Aufruf zentraler Methoden aus Object
 - `getProperty(String name)`
 - `invokeMethod(String name, Object arguments)`
- Diese Methoden können selbstverständlich überschrieben werden


```
class DynamicObject {
    Object getProperty(String name){
        return 'it is a property!'
    }
    void setProperty(String name, Object value){
        println("setting property ${name} to ${value}")
    }
    def invokeMethod(String name, def args){
        println('executing invokeMethod')
        return "OK"
    }
}
```

- Die `Expando`-Klasse implementiert diese Methoden bereits und stellt somit eine dynamische Klasse zur Verfügung
 - Auch der `Map`-Zugriff funktioniert!
- Damit können einer `Expando`-Instanz durch simple Zuweisung hinzugefügt werden:
 - **Eigenschaften**
 - `expando.prop = 'Value'`
 - **Methoden**
 - `expando.method = { //... }`

```
def book = new Expando(isbn: "isbn1", title: "title1",  
    pages:200, price:19.99);  
//...  
book.topic = "Science"  
book.year = 8  
def saveClosure = {  
    println "saving book... " + delegate.title  
}  
book.save = saveClosure  
//...  
book.save()
```

6





















GRADLE IM DETAIL

6.1

UMGEBUNG

- Ein Gradle-Skript wird in der Programmiersprache Groovy geschrieben
 - Damit stehen sämtliche syntaktische Konstrukte zur Verfügung
 - Aber auch die gesamte Klassenbibliothek
- Beim Aufruf des Gradle-Befehls wird das Skript ausgeführt
 - Im Detail wie auch bei Groovy komplizierter
 - Groovy wird zu Bytecode kompiliert und von einer Java Virtual Machine ausgeführt

- Beim Start des Gradle-Skripts werden erst einmal eine ganze Reihe von Eigenschaften und Funktionen definiert
- Diese stellen eine Umgebung zur Verfügung, die das Erzeugen von Build-Sequenzen vereinfacht

this		
	<code>buildEnvironment task</code>	<code>BuildEnvironmentReportTask</code>
	<code>cleanIdea task</code>	<code>Delete</code>
	<code>cleanIdeaModule task</code>	<code>Delete</code>
	<code>cleanIdeaProject task</code>	<code>Delete</code>
	<code>absoluteProjectPath(String path)</code>	<code>String</code>
	<code>afterEvaluate(Closure closure)</code>	<code>void</code>
	<code>afterEvaluate(Action<? super Project> action)</code>	<code>void</code>
	<code>allprojects(Closure configureClosure)</code>	<code>void</code>
	<code>allprojects(Action<? super Project> action)</code>	<code>void</code>
	<code>ant(Closure configureClosure)</code>	<code>AntBuilder</code>
	<code>ant(Action<? super AntBuilder> configureAction)</code>	<code>AntBuilder</code>
	<code>apply(Closure closure)</code>	<code>void</code>
	<code>apply(Map<String, ?> options)</code>	<code>void</code>
	<code>apply(Action<? super ObjectConfigurationAction> action)</code>	<code>void</code>
	<code>artifacts(Closure configureClosure)</code>	<code>void</code>
	<code>artifacts(Action<? super ArtifactHandler> configureAction)</code>	<code>void</code>
	<code>beforeEvaluate(Closure closure)</code>	<code>void</code>
	<code>beforeEvaluate(Action<? super Project> action)</code>	<code>void</code>
	<code>buildscript(Closure configureClosure)</code>	<code>void</code>
	<code>build (default package)</code>	

6.2

TASKS

- Innerhalb eines Gradle-Files können beliebig viele Tasks definiert werden
- Innerhalb eines Tasks werden Eigenschaften und Methoden definiert
 - Diese werden für alle Tasks bei Ausführung des Gradle-Skripts ausgeführt
- doFirst und doLast
 - Jeweils eine Liste von Funktionen
 - Diese werden nur dann ausgeführt, wenn der entsprechende Task ausgeführt wird

```
def scriptVar = "Var defined in Script"
println("In Script: ${scriptVar}")

task step1 {
    def param = "Param in Step1"
    println("In step1 ${param}")
    doFirst {println("doFirst in step1, param=" + param)}
    doFirst {println("another doFirst in step1, param=" +
param)}
}

task step2 {
    def param = "Param in Step2"
    println("In step2 ${param}")
    doFirst {println("doFirst in step2, param=" + param)}
}
```

```
In Script: Var defined in Script
```

```
In step1 Param in Step1
```

```
In step2 Param in Step2
```

```
> Task :step1
```

```
second doFirst in step1, param=Param in Step1
```

```
doFirst in step1, param=Param in Step1
```

```
In Script: Var defined in Script
```

```
In step1 Param in Step1
```

```
In step2 Param in Step2
```

```
> Task :step2
```

```
doFirst in step2, param=Param in Step2
```

- `task` ist eine Funktion
 - `task("demo", {})`
- Die übergebene Closure hat Zugriff auf die internen Eigenschaften der Task-Umgebung
 - `doFirst` und `doLast` sind auch wieder Funktionen
 - die übergebene Closure wird von der Gradle-Runtime entsprechend ausgeführt
- Der erste Parameter des `task`-Aufrufs ist wieder eine Funktion
 - Parameter: Ein Konfigurationsobjekt
 - `task step2(dependsOn: 'step1')`
- Der Name der Funktion wird als neue Eigenschaft der Gradle-Umgebung hinzugefügt
 - Damit können Task-Eigenschaften und Methoden auch dynamisch nachträglich geändert werden

```
def scriptVar = "Var defined in Script"
println("In Script: ${scriptVar}")

task step1 {
    def param = "Param in Step1"
    println("In step1 ${param}")
    doFirst {println("doFirst in step1, param=" + param)}
    doFirst {println("another doFirst in step1, param=" + param)}
}

task step2(dependsOn: "step1") {
    def param = "Param in Step2"
    println("In step2 ${param}")
    doFirst {println("doFirst in step2, param=" + param)}
}

step1.doFirst {println("one more doFirst in step1, param is not
available")}
```

- Typische Aufgaben eines Build-Scripts müssen nicht mit eigenen Tasks implementiert werden
- Die Gradle-Community definiert eine große Anzahl fertig benutzbarer Tasks, die nur noch konfiguriert werden müssen
- <https://docs.gradle.org/current/dsl/>


```
task copyDocs(type: Copy) {  
    from 'src/main/doc'  
    into 'build/target/doc'  
}
```

6.3

PLUGINS

- PlugIns definieren fertige Build-Lifecycles und Tasks für verschiedene Umgebungen
- https://docs.gradle.org/current/userguide/plugin_reference.html