integrata
Cegos Group

inspire
qualify
change

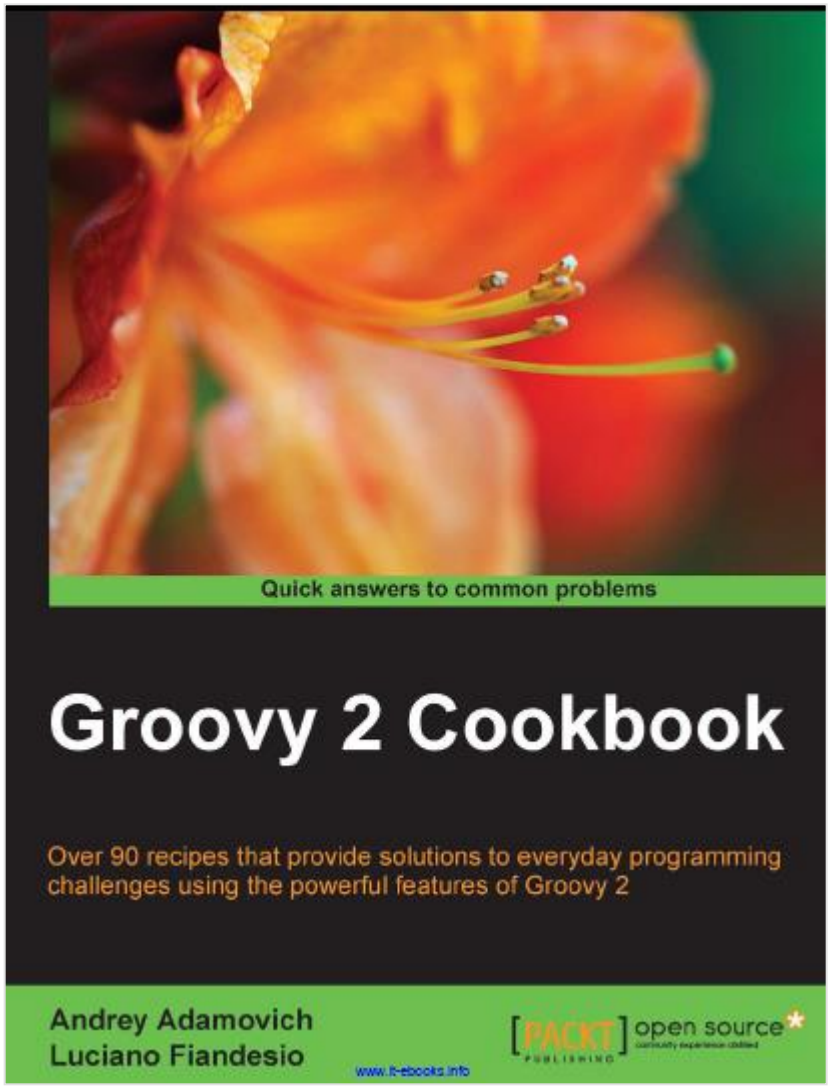
Groovy

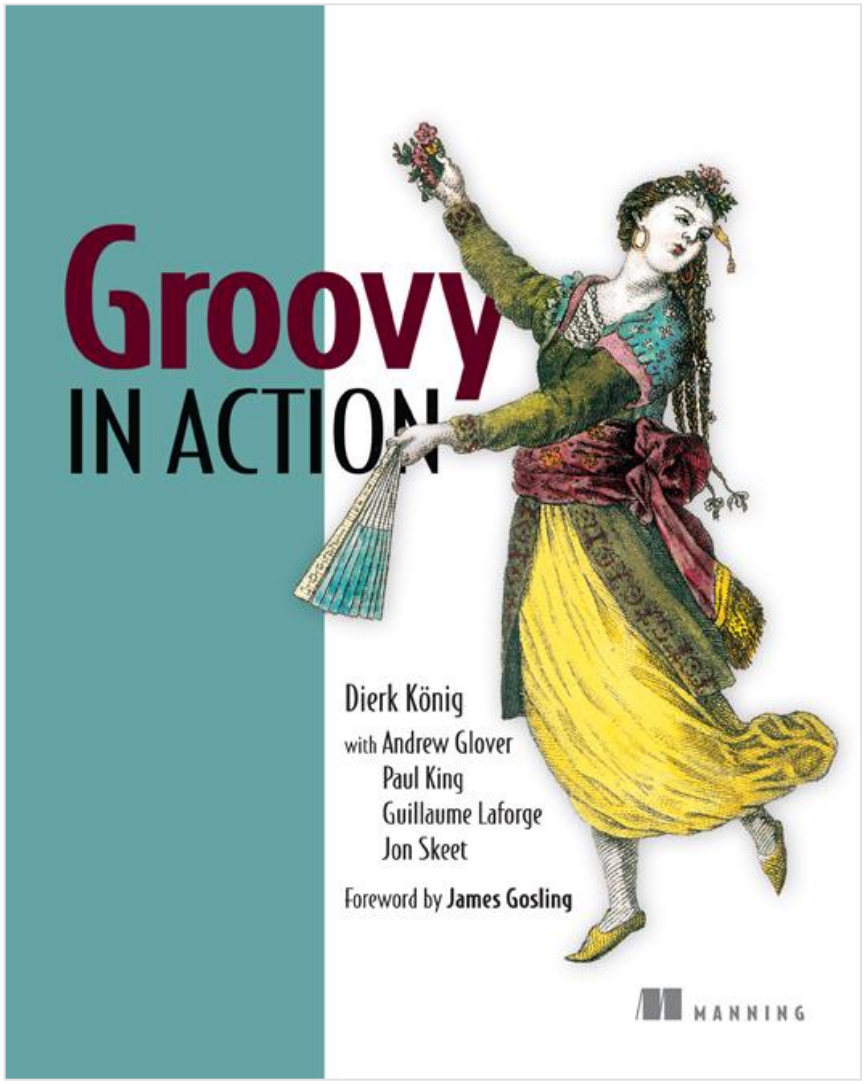
Grundlagen der Programmierung

Ein Seminar für BITBW Stuttgart

Literatur und Quellen

integrata
Cegos Group
inspire
qualify
change





Einige Hinweise

integrata Cegos Group | inspire qualify change

- 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/JavacreamTraining/org.javacream.training.groovy`
- Konventionen
  - Befehle werden in `Courier-Schriftart` dargestellt
  - Dateinamen werden in *`kursiver Courier-Schriftart`* dargestellt
  - Links werden in `unterstrichener Courier-Schriftart` dargestellt

Copyright und Impressum

Javacream

Dr. Rainer Sawitzki

Alois-Gilg-Weg 6
81373 München

eMail: training@rainer-sawitzki.de

Inhalt

integrata Cegos Group
inspire qualify change

Einführung	6
Programmieren in Groovy	22
Groovy und Java	45
Objektorientierung	57
Dynamische Programmierung	79

1

EINFÜHRUNG

1.1

HINTERGRUND

Wer steckt hinter Groovy?

integrata Cegos Group
inspire qualify change

- 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

1.2

EINSATZBEREICHE

Übersicht: Einsatzbereiche

integrata Cegos Group
inspire qualify change

- Skriptsprache
- Ergänzung zur Programmiersprache Java
- Vollwertige eigene Programmiersprache

Einsatzbereiche: Skriptsprache

integrata Cegos Group
inspire qualify change

- 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

Skriptsprachen: Notwendiges Know-How

- 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

Einsatzbereiche: Ergänzung zur Programmiersprache Java

integrata Cegos Group
inspire qualify change

- 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

Groovy für Java-Entwickler: Notwendiges Know-How

integrata Cegos Group
inspire qualify change

- 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

Einsatzbereiche: Vollwertige eigene Programmiersprache

- Hier sind die wichtigen Inhalte beider vorheriger Ansätze gleichermaßen relevant

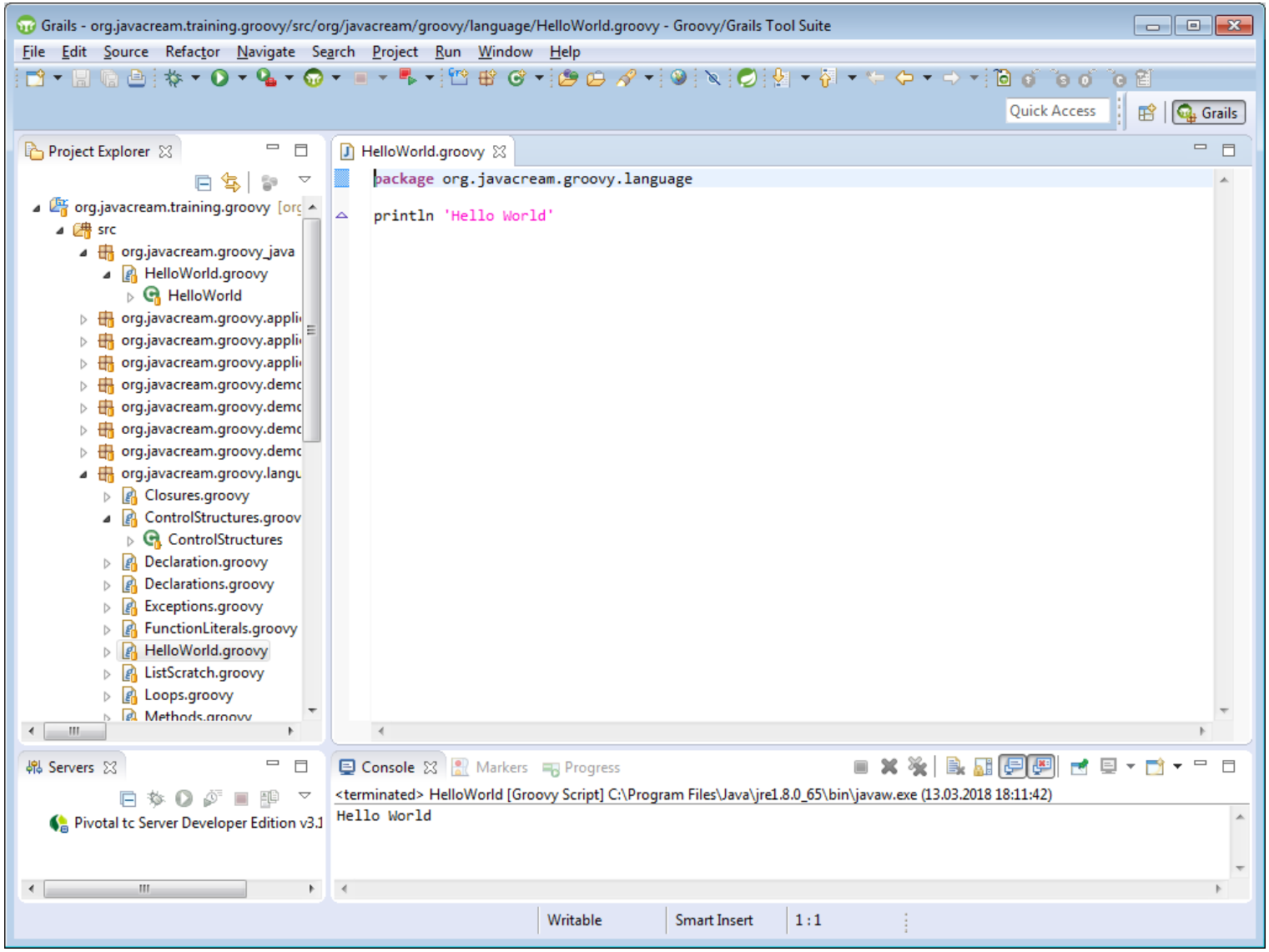
1.3

INSTALLATION UND WERKZEUGE

Integration in Entwicklungsumgebungen

integrata Cegos Group
inspire qualify change

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


Grails - org.javacream.training.groovy/src/org/javacream/groovy/language/HelloWorld.groovy - Groovy/Grails Tool Suite
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Grails
Project Explorer
org.javacream.training.groovy
src
org.javacream.groovy_java
HelloWorld.groovy
HelloWorld
org.javacream.groovy.langu
Closures.groovy
ControlStructures.groov
ControlStructures
Declaration.groovy
Declarations.groovy
Exceptions.groovy
FunctionLiterals.groovy
HelloWorld.groovy
ListScratch.groovy
Loops.groovy
HelloWorld.groovy
package org.javacream.groovy.language
println 'Hello World'
Servers
Pivotal tc Server Developer Edition v3.1
Console
Markers
Progress
<terminated> HelloWorld [Groovy Script] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (13.03.2018 18:11:42)
Hello World
Writable
Smart Insert
1 : 1

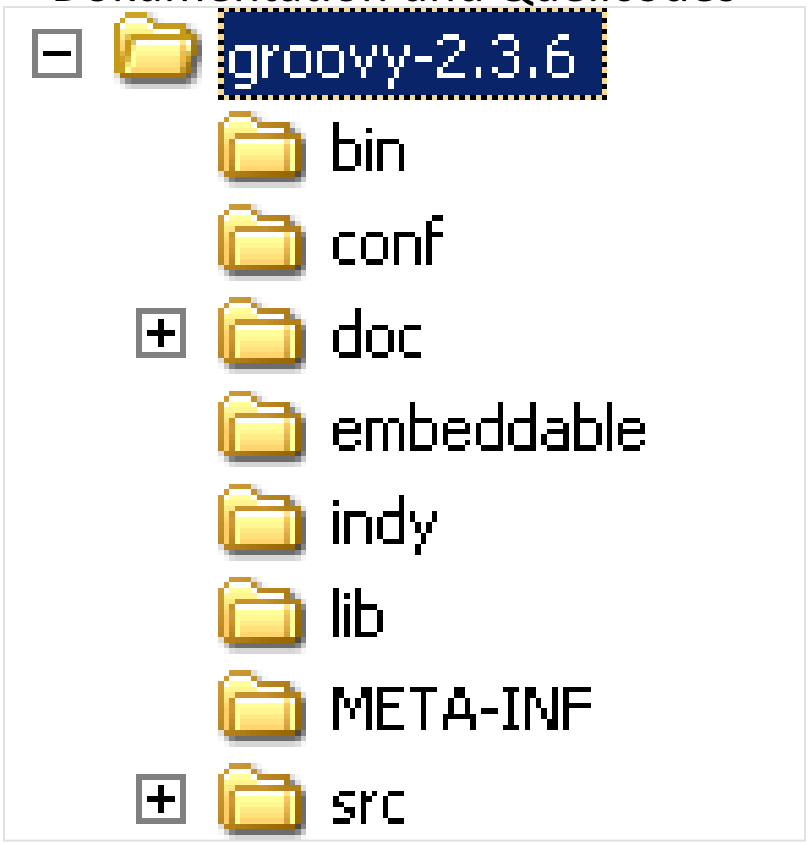
Installation ohne Entwicklunsumgebung

- 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

Installationsverzeichnis

integrata Cegos Group | inspire qualify change

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



Werkzeuge

integrata Cegos Group
inspire qualify change

- `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 alla anderen Skripte
  - Startet einen Java-Prozess

2

PROGRAMMIEREN IN GROOVY

2.1

GRUNDLAGEN DER PROGRAMMIERUNG

Groovy-Skripte

- 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

Beispielprogramme

- 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`

Erste Sprach-Elemente

- 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

2.2

VARIABLEN UND LITERALE

Definitionen

- 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`

Literale: Zeichenketten

integrata Cegos Group | inspire qualify change

- 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

Literale: Zahlen

integrata Cegos Group | inspire qualify change

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

Literale: boolean

integrata Cegos Group
inspire qualify change

- 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`

Operatoren

- 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
```

Operatoren

integrata Cegos Group
inspire qualify change

- Vergleichsoperatoren

  `>` `<` `>=` `<=` `!=` `==`

- Logische Operatoren

  `&&` Und (binär)
  `||` Oder (binär)
  `!` Nicht, logische Negation (unär)

- Bit-Operatoren

  `>>`
  `>>>`
  `<<`
  `&`
  `|`
  `^`
  `~`

Fallunterscheidung mit if

integrata Cegos Group
inspire qualify change

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

Groovy und das wahre Bedingungen

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

Fallunterscheidung mit switch

integrata Cegos Group
inspire qualify change

```
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 Schleifen


```
while( boolescher Ausdruck )
          Anweisung

do
          Anweisung
     while( boolescher Ausdruck );
```

for Schleifen


```
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

FUNKTIONEN

Deklaration

- 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

integrata Cegos Group
inspire qualify change

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

Aufruf einer Funktion

- 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

Beispiel: Funktionsaufruf

integrata Cegos Group
inspire qualify change

```
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
}
```

DATEN-CONTAINER

Übersicht

- `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`

Beispiel: Liste


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

Beispiel: Map

integrata
Cegos Group
inspire
qualify
change

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

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

Ranges

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

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

Iteration über Listen

- 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

Beispiel: Listen-Iteration


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

REFERENZEN

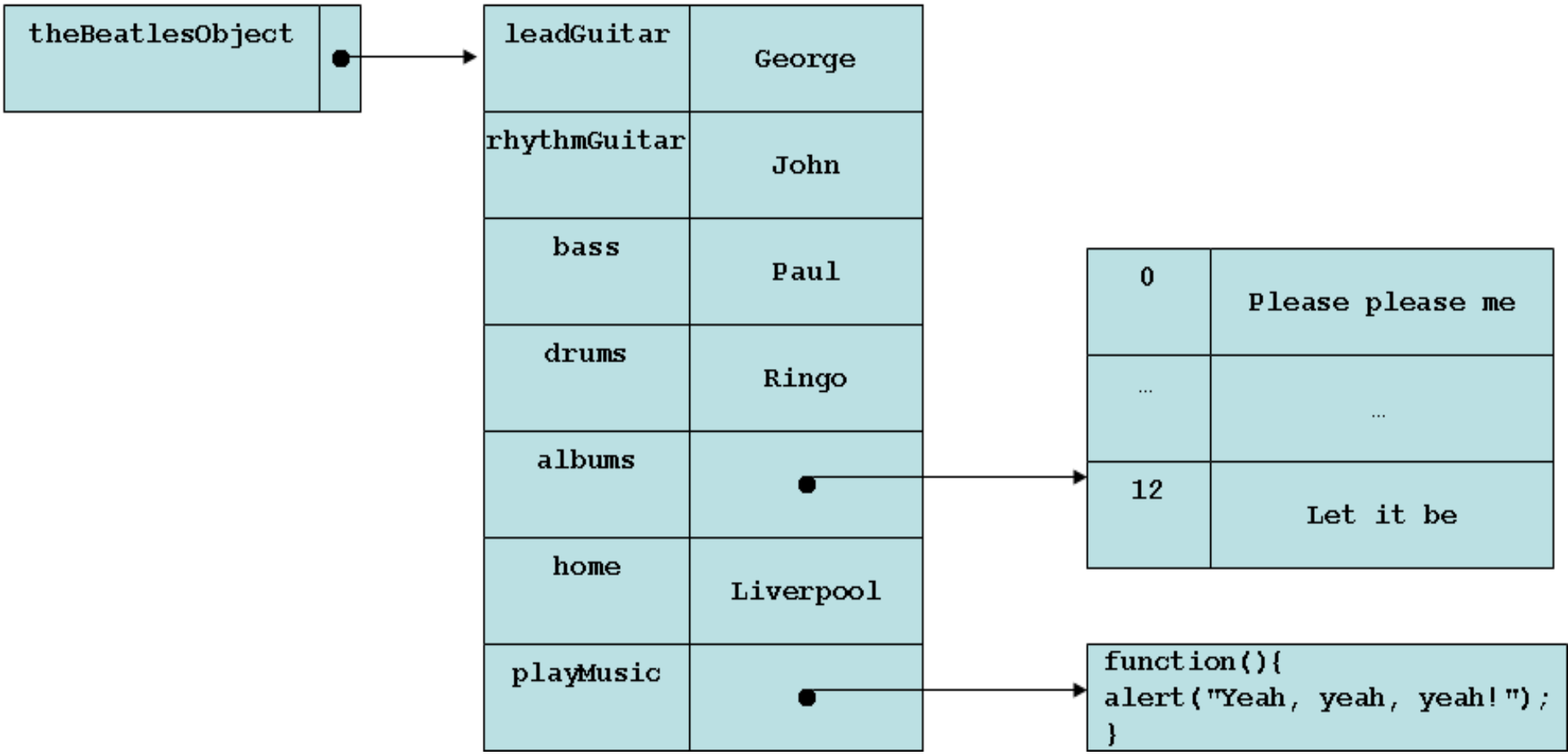
Referenzen und Objekte

integrata Cegos Group
inspire qualify change

- 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

Referenzen und Objekte

integrata Cegos Group
inspire qualify change



FUNKTIONEN IM DETAIL

Funktionen

integrata Cegos Group
inspire qualify change

- 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

Expertenwissen: Closures

integrata Cegos Group | inspire qualify change

- 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 `innnerVar`

Expertenwissen: Closures und Gültigkeit von Variablen

integrata Cegos Group | inspire qualify change

- 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()
```

- `innnerVar` 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

ANWENDUNGSPROGRAMMIERUNG

Übersicht

- In den folgenden Abschnitten werden jeweils kurze Beispiele für exemplarische Algorithmen in Groovy gezeigt
- `https:GitHub.com/JavacreamTraining/org.javacream.training.groovy`
  - Verzeichnis `snippets`

2.3

SYNTACTIC SUGAR

Was ist „Syntactic Sugar“?

integrata Cegos Group
inspire qualify change

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

Beispiele in Groovy

integrata Cegos Group
inspire qualify change

- 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")}`

3

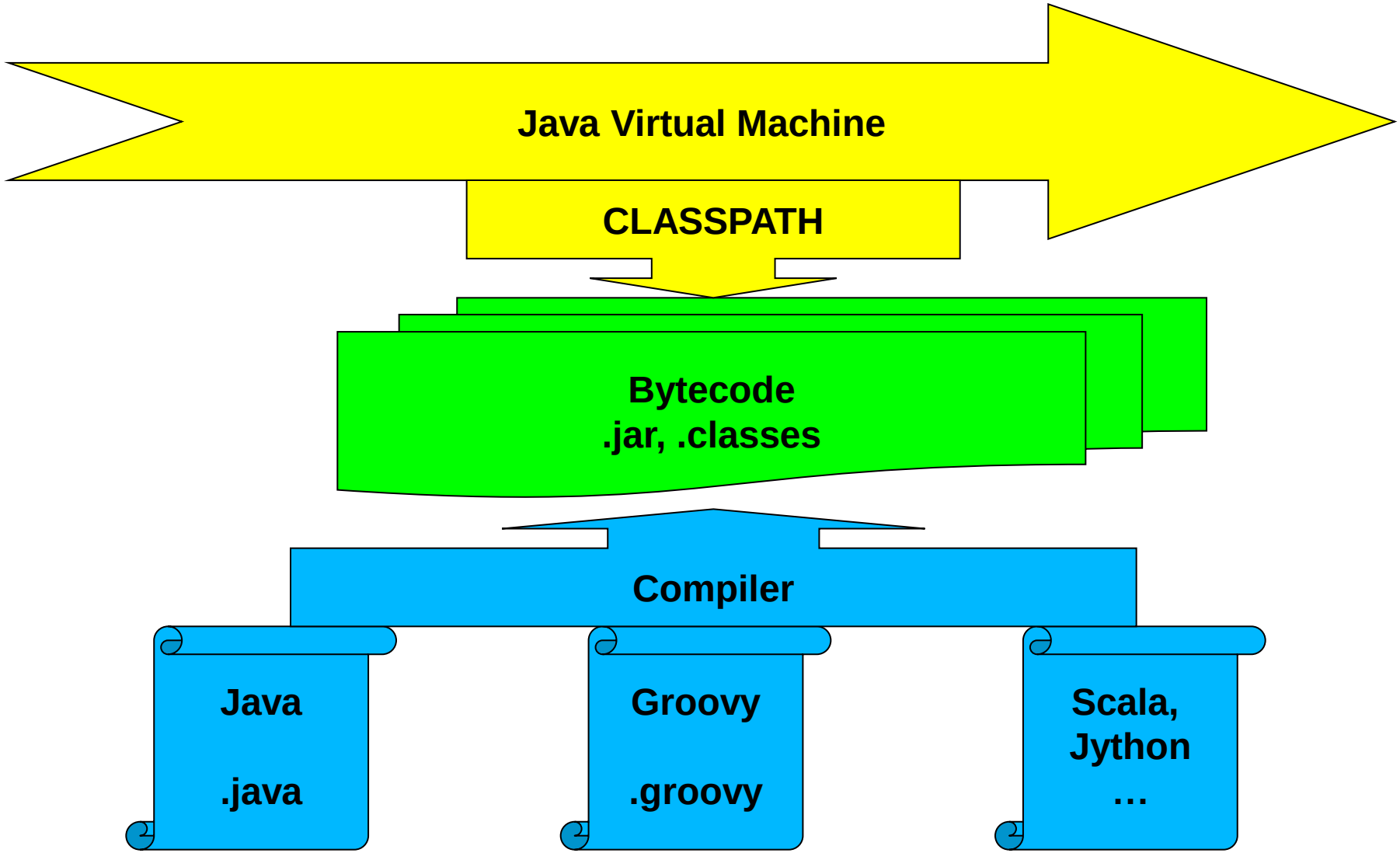
GROOVY UND JAVA

3.1

INTEROPERABILITÄT

Eine weitere Sprache für die Java Runtime

integrata Cegos Group
inspire qualify change



Groovy: Eine Skriptsprache?

integrata Cegos Group
inspire qualify change

- `print "Hello World!"`
  - Ist eine vollständige und funktionierende Groovy-Anwendung
- Aber: Bytecode ist auf Klassenobjekte ausgerichtet
- Problemlösung: Der Grovvy-Compiler erstellt aus einem Skript eine ausführbare Klasse
  - Beweise
    - Darstellung des Outlines des Skriptes in Eclipse

Interoperabilität

- Die Java Virtuelle Maschine macht keinen Unterschied, wie der Bytecode erzeugt wurde
  - Damit „sehen“ zur Laufzeit Groovy-Programme sofort Java-Klassen und umgekehrt
  - Vorsicht: Java-Anwendungen benötigen Zugriff auf die Groovy-Bibliotheken
    - Die `groovy-all.jar` muss sich im Klassenpfad befinden
- Auch die Compiler beruhen auf Bytecode
  - Damit kann der Compiler auch automatische Typ-Prüfungen durchführen
- Ebenso Entwicklungsumgebungen
  - Auto-Vervollständigung!

FALLSTRICKE FÜR JAVA-ENTWICKLER

Mischung von Skript und Klassendefinition

integrata Cegos Group
inspire qualify change

```
/*
 * File HelloWorld.groovy
 */
print "Hello World!"

class HelloWorld{

}
```

Impliziter Rückgabewert


```
int myFunc(state){
  if (state){
      result = "Hugo"
  }else{
      return 42;
  }
}

println myFunc(false)
println myFunc(true)
```

3.2

BIBLIOTHEKEN

JDK

- Alle Klassen des Java Development Kits stehen uneingeschränkt zur Verfügung
  - Diese Klassen sind bei Bedarf teilweise drastisch erweitert worden!
    - Dokumentation: /doc/html/groovy-jdk/index.html
    - Bei Bedarf können auch eigene Erweiterungen durchgeführt werden
      - Das ist jedoch ein fortgeschrittenes Thema, siehe Groovy Dokumentation
- Groovy-Klassen stehen über normale Archive auch Java zur Verfügung
  - Zwei Varianten
    - `lib`-Verzeichnis enthält die Groovy-Bibliotheken für ältere JVM-Versionen
    - `indy`-Verzeichnis enthält die Groovy-Bibliotheken für JVM-Versionen >= 7
      - Mit Unterstützung des `invokedynamic`-Befehls

Beispiel: Einfaches Multithreading mit Groovy

integrata Cegos Group | inspire qualify change

```
class SimpleWorker {
   CyclicBarrier barrier = new CyclicBarrier(4);
   def work(String action){
     println "start working for action $action"
     Thread.start{subUnitOfWork1(action)}
     Thread.start{subUnitOfWork2(action)}
     Thread.start{subUnitOfWork3(action)}
     barrier.await();
     println "work finished for action $action"
   }

   private def subUnitOfWork1(action){
     println "starting subUnitOfWork1 for action $action"
     Thread.sleep(200);
     println "finished subUnitOfWork1 for action $action"
     barrier.await();
   }
   //…
}
```

Community, Treiber, Produkte

- Alle weiteren Java-Produkte stehen ebenfalls zur Verfügung
  - Datenbanktreiber
  - Apache-Bibliotheken
  - Frameworks
    - JUnit
    - Spring
    - …

Beispiel: Datenbankzugriff mit Groovy

integrata Cegos Group
inspire qualify change

```
class Dao {
   DataSource source
   {
     source = new org.hsqldb.jdbc.jdbcDataSource()
     source.database = 'jdbc:hsqldb:hsql://localhost'
     source.user = 'sa'
     source.password = ''
   }

   def doInStatement(Closure c){
     def con = source.getConnection()
     Statement statement = con.createStatement()
     def result = c.call statement
     statement.close()
     con.close()
     return result
   }
}
```

Unit-Tests für Groovy

integrata Cegos Group
inspire qualify change

- Klassisch

```
public class SimpleUnitTest {
  @Test void simpleTest(){
    Assert.assertTrue(true)
  }
}
```

- Mit Groovy-Erweiterungen

```
public class SimpleGroovyUnitTest extends GroovyTestCase{
  void testSimple(){
    this.assertLength(2, ["A","B"] as String[])
  }
}
```

Groovy Community

- Grails
  - Ein Web Framework
    - Analogie zu Ruby on Rails nicht zufällig
- GPars
  - Realisierung paralleler Systeme

4

OBJEKTORIENTIERUNG

4.1

DIE KLASSENBIBLIOTHEK

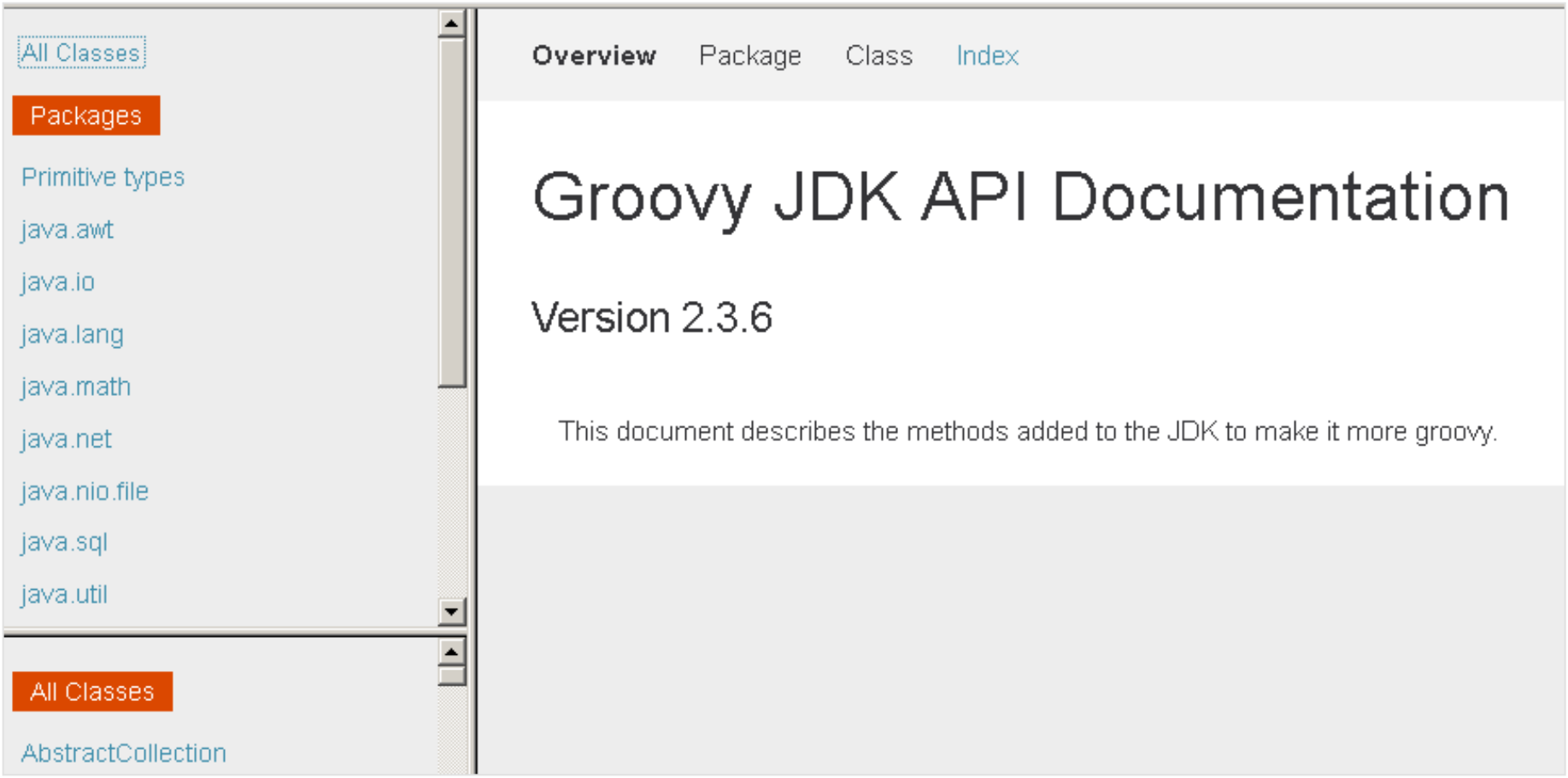
Groovy und die Java-Klassenbibliothek

integrata Cegos Group
inspire qualify change

- 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}`

doc/html/groovy-jdk/index.html

integrata
Cegos Group
inspire
qualify
change



Java 8

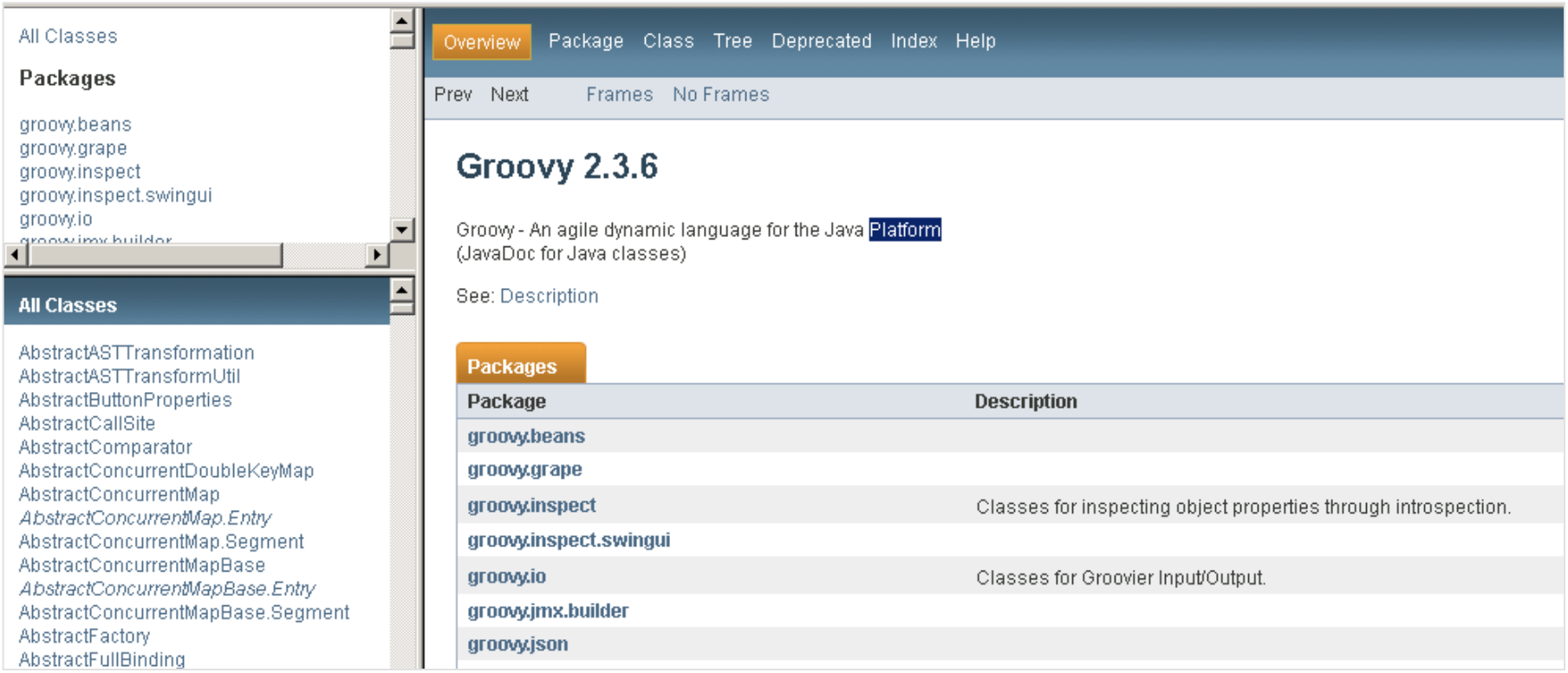
- Java 8 hat einige Feature von Groovy übernommen
  - Funktionale Programmierung
  - Erweiterte Collections
- Die Groovy-Syntax unterstützt jedoch noch nicht Java 8
  - Geplant für die Version 3
  - http://groovy-lang.org/releasenotes/groovy-3.0.html

Groovy 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

/doc/html/api/index.html

integrata
Cegos Group
inspire
qualify
change



4.2

EIGENE KLASSEN

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`

Ein kleiner Vorgriff

- „Unterstützt Groovy auch Interfaces?“
  - Ja, aber dieses Feature hat in Groovy bei weitem nicht die Bedeutung wie in Java
- „Unterstützt Groovy auch Mehrfachvererbung für Klassen?“
  - Nein, aber Groovy 2.3 führt die so genannten Traits ein, die etwas vereinfacht Schnittstellen mit Attributen und Methoden darstellen
- „Ist Groovy als Script-Sprache dynamisch, das heißt, können Objekten zur Laufzeit neue Attribute und Methoden hinzugefügt werden?“
  - Nicht automatisch, aber es gibt mehrere Sprachfeatures, genau dies zu erreichen
    - `Expando`-Objekte
    - Manipulation der Meta-Klasse eines Objekts oder einer Klasse
    - Categories und MixIns
    - Dynamische Properties (`get`, `set`-Methode) und Methoden (`invokeMethod`)

Syntactic Sugar für OOP

- 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`
  - …

OOP und Operatoren

- 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

integrata Cegos Group
inspire qualify change

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

Eigene Klassen und Operatoren

- 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
```

Delegation statt Vererbung

integrata Cegos Group | inspire qualify change

- 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()
  }
```

4.3

TRAITS

Was sind Traits?

integrata Cegos Group
inspire qualify change

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

Beispiel


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

Benutzung

integrata Cegos Group | inspire qualify change

```
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
}
```

Einige Details

- 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

DYNAMISCHE PROGRAMMIERUNG

5.1

MOTIVATION

Der statische Ansatz

integrata Cegos Group
inspire qualify change

- 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 `Person`en
    - Studierende Arbeiter?
    - Ein Student wird zum Arbeiter?

Was ist "dynamisch"?

integrata Cegos Group | inspire qualify change

- 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.”*

Was ist besser?

integrata Cegos Group
inspire qualify change

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

Grundlage der dynamischen Programmierung

integrata Cegos Group
inspire qualify change

- 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

Ein DynamicObject

integrata Cegos Group
inspire qualify change

```
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"
  }
}
```

groovy.util.Expando

- 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 = {//…}`

groovy.util.Expando: Beispiel


```
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()
```

5.2

CATEGORIES UND MIXINS

Statisch oder dynamisch: Beides!

- Groovy unterstützt zwei Möglichkeiten, Dynamik und Typsicherheit zu vereinen
  - Kategorien
  - MixIns

Category

integrata Cegos Group | inspire qualify change

- Eine Kategorie ist eine Klasse mit besonders deklarierten Methoden
  - statisch
  - erster Parameter ist das Objekt, das die Kategorie benutzt, `self`
    - Dieser Parameter kann typisiert werden
- Beispiel

```
class DemoCategory {
  static def answer(String self){
    return 57
  }
  static def answer(List self){
  return 58
  }
  static def answer(Integer self){
  return 59
  }
}
```

Benutzung einer Category

integrata Cegos Group | inspire qualify change

- `use`-Schlüsselwort
  - definiert einen Block, in dem die Category angewendet wird
  - Alle Objekte bekommen die Methoden der Category zugeordnet
    - der erste Parameter wird intern gesetzt
    - Falls der `self`-Parameter typisiert angegeben wurde wird die Category auf diese Typen beschränkt
- Beispiel

```
def list = []
def s = "Hugo"
def i = 1
def b = true
use (DemoCategory.class){
  println list.answer()
  println s.answer()
  println i.answer()
  // println b.answer() geht nicht!
}
```

Mixins

- Die Funktionen jeder vorhandenen Klasse können anderen Klassen oder Objekten zugeordnet werden
  - Pro Klasse: `@Mixin`
    - Mittlerweile zu Gunsten von Traits deprecated
  - Pro Objekt
    - `mixin`-Methode der Metaklasse
      - Metaklassen werden später detaillierter besprochen

@Mixin

integrata Cegos Group
inspire qualify change

```
class MixedIn {
  def mixedInMethod(){
      println "called mixedInMethod"
  }
}
@Mixin(MixedIn)
class Book{
  //...
}
Book b =new Book()
b.mixedInMethod()
```

mixin-Methode

integrata Cegos Group
inspire qualify change

```
class MixedIn {
  def mixedInMethod(){
      println "called mixedInMethod"
  }
}
class Person{
  //...
}
Person p = new Person()
Person p2 = new Person()
p.metaClass.mixin(MixedIn)
p.mixedInMethod()
//p2.mixedInMethod()
```

5.3

METAKLASSEN

Überblick

- Metaklassen enthalten die Definitionen der Eigenschaften für alle Groovy-Objekte
  - Für eine rein statische Programmierung ist die Metaklasse prinzipiell gleich der Klassendefinition
- Metaklassen können verändert werden
  - Damit ändert sich zur Laufzeit das Verhalten der Objekte
- Jedes Groovy-Objekt referenziert seine Metaklasse über die Property `metaClass`
  - Auch Klassenobjekte selber haben eine Metaklasse
  - Metaklassen können auch gesetzt werden

Metaklassen und Instanzen

integrata Cegos Group | inspire qualify change

- Nach der Instanzierung referenziert `metaClass` des Objekts die Metaklasse der erzeugenden Klasse
  - allerdings nicht direkt, sondern über die `HandleMetaClass`
  - im Wesentlichen eine Liste von `MetaClass`-Implementierungen
- Sobald die Metaklasse eines Objektes manipuliert wird, wird eine spezielle Metaklassen-Implementierung erzeugt, die Erweiterungen zulässt
  - `ExpandoMetaClass`
  - Diese wird in die `HandleMetaClass` eingetragen
    - Und dadurch stehen die neuen Eigenschaften zur Verfügung
- Vorsicht: Eine Änderung der Metaklasse der Klasse selber wirkt sich nicht auf die bereits erzeugten Instanzen aus!
  - Die Klasse bekommt eine neue `MetaClass`-Referenz
  - Diese wird nicht in die `HandleMetaClass` der Instanzen übertragen

Beispiel

integrata Cegos Group | inspire qualify change

```
class Dvd {//...}

Dvd d1 = new Dvd()
Dvd d2 = new Dvd()
println "d1.metaClass: " + d1.metaClass
println "d2.metaClass: " + d2.metaClass
println "DVD.metaClass: " + Dvd.metaClass
d1.metaClass.myMethod = {-> println "d1: " + delegate}
Dvd.metaClass.myMethod = {-> println "Dvd: " + delegate}
println "d1.metaClass: " + d1.metaClass
println "d2.metaClass: " + d2.metaClass
println "DVD.metaClass: " + Dvd.metaClass
Dvd d3 = new Dvd();
println "d3.metaClass: " + d3.metaClass
d1.myMethod()
// d2.myMethod() geht nicht!
d3.myMethod()
```

Ausgabe des Beispiels

integrata Cegos Group | inspire qualify change

```
d1.metaClass:
   org.codehaus.groovy.runtime.HandleMetaClass@589375[groovy.lang.MetaClassImpl@58937
   5[class org.javacream.groovy.demo.dynamic.extend.Dvd]]
d2.metaClass:
   org.codehaus.groovy.runtime.HandleMetaClass@589375[groovy.lang.MetaClassImpl@58937
   5[class org.javacream.groovy.demo.dynamic.extend.Dvd]]
DVD.metaClass:
   org.codehaus.groovy.runtime.HandleMetaClass@589375[groovy.lang.MetaClassImpl@58937
   5[class org.javacream.groovy.demo.dynamic.extend.Dvd]]
d1.metaClass:
   org.codehaus.groovy.runtime.HandleMetaClass@149f65b[groovy.lang.ExpandoMetaClass@1
   49f65b[class org.javacream.groovy.demo.dynamic.extend.Dvd]]
d2.metaClass:
   org.codehaus.groovy.runtime.HandleMetaClass@589375[groovy.lang.MetaClassImpl@58937
   5[class org.javacream.groovy.demo.dynamic.extend.Dvd]]
DVD.metaClass:
   groovy.lang.ExpandoMetaClass@ccf2e0[class
   org.javacream.groovy.demo.dynamic.extend.Dvd]
d3.metaClass:
   org.codehaus.groovy.runtime.HandleMetaClass@ccf2e0[groovy.lang.ExpandoMetaClass@cc
   f2e0[class org.javacream.groovy.demo.dynamic.extend.Dvd]]


d1: org.javacream.groovy.demo.dynamic.extend.Dvd@acb798
Dvd: org.javacream.groovy.demo.dynamic.extend.Dvd@7e34db
```