Groovy 1.8
für Fortgeschrittene
CamelCaseCon 7.9.2011

Dr. Joachim Baumann

Joachim.Baumann@cirquent.de

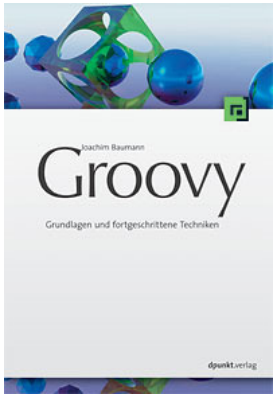
your archiTect
cirquent | NTT DATA

Agenda (45 Minuten)

- Über mich
- Interessante Neuigkeiten in Groovy 1.8
- Groovy++

Dr. Joachim Baumann

IT-Management Consultant / Competence Center Manager



Ausbildung

1986–1992 **Informatik, Universität Stuttgart**

1994-1999 **Promotion, Universität Stuttgart, Universität Genf**

1999-2000 **Postdoc, Universität Stuttgart**

Beruflicher Werdegang

1993	ANT Bosch Telecom, Software-Entwickler
2000	COSS Softwaretechnik AG, Bereichsleiter Produktentwicklung und Forschung
2001	Junidas GmbH, Geschäftsführer
2005	Xinaris GmbH, Geschäftsführer
2007	Cirquent GmbH, Leitung Competence Center iBPM Integration Solutions (Themenbereiche EAI und SOA), Vice President IT & Methods, Director Shoring

Schwerpunkte

- Architektur großer Systeme
- Open-Source-Produkte
- Kommunikationsnetze, Mobile Computing, Verteilte Systeme
- Training, Coaching, Vorträge
- Agile Methoden

Referenzen (Auszug)

- Nürnberger Versicherungsgruppe, Nürnberg
  - Absatzplanungssystem der Nürnberger Versicherung
  - Technischer Projektleiter
  - Architekt
- FIDUCIA IT AG, Karlsruhe
  - Architekturberatung
  - Übernahme kritischer Projekte
  - Qualitätssicherung
  - Coaching
- Trust International GmbH, Frankfurt
  - Hochgradig konfigurierbares Web-Frontend-Framework für Hotelbuchungen
  - Technischer Projektleiter
  - Architekt
- Artikel, Bücher, Vorträge
  - 2 Bücher
  - Über 40 Artikel
  - Vorträge auf int. und deutschen Konferenzen
  - Aktuelle Beispiele
    - „Groovy: Grundlagen und fortgeschrittene Techniken“, d.punkt Verlag, 600 Seiten
    - „Groovy: Grundlagen“ in iX-Sonderheft Java Kompakt 1/2009

Interessante Neuigkeiten in Groovy 1.8

- GPARS ist ab sofort Teil von Standard-Groovy
  - Parallele Programmierung mit Groovy (benötigt einen eigenen Vortrag)
- Elegante DSLs durch Befehlsverkettung
- Zusätzliche Funktionalität für Closures
- Neue AST-Transformationen
- Berechnungen mit primitiven Datentypen deutlich beschleunigt
- Verschiedenes
  - Eingebaute JSON-Unterstützung
    - Es gibt neue Builder, um JSON-Objekte elegant erzeugen zu können
  - Verwendung des Unterstriches in Zahlenfolgen: 1_000_000_000
  - Direkte Binärnotation: 0b01010101
  - Fangen mehrerer Ausnahmen (Exceptions) mit einer Anweisung
    catch (IOException | NullPointerException e)

Neue Möglichkeiten zur Befehlsverkettung erlauben elegantere DSLs (Domänenspezifische Sprachen)

- Eine Änderung der Grammatik erlaubt elegante Verkettung von Methodenaufrufen
  - Punktoperator kann in den meisten Fällen entfallen
  - Klammern können in den meisten Fällen entfallen
  - Modifizierte Versionen von Befehlen wie println() unterstützen dies
- Erlaubt einfacher lesbare domänenspezifische Sprachen in Groovy

Beispiel von Guillaume LaForge

```
show = { println it }
sqrt = { Math.sqrt(it) }

def please(action) {
    [ the: { what ->
        [of: { n -> action(what(n)) }]
    }]
}

please show the sqrt of 100
```

please(show).the(sqrt).of(100)

Zusätzliche Funktionalität für Closures

- Komposition für Closure
- Auflösung von Endrekursion fürClosure
- Aufruf-Caches für Closures
- Verbesserungen bei der Vorbelegung von Parametern in Closures

Komposition für Closures

Verkettung mit dem Shift-Operator

- Komposition wird jetzt deutlich einfacher
- Groovy bietet jetzt einen überladenen Shift-Operator für Closures an
- Beide Shift-Operatoren sind überladen

- Shift-Operator funktioniert für beliebig viele Parameter

- Parameter werden der ersten Closure zur Verfügung gestellt
- Ergebnisse der ersten Closure werden als Parameter für die zweite Closure verwendet
  - Ergebnisliste wird umgewandelt in Einzelparameter

```
comp = { f, g, Object[] params -> g( f(*params) ) }

mul3 = {x-> x * 3 }                                   Vor 1.8
add5 = {x -> x + 5 }
mul3add5 = comp.curry(mul3, add5)
add5mul3 = comp.curry(add5, mul3)

println mul3add5(2)                                   11
println add5mul3(2)                                   21
```

```
mul3 = {x-> x * 3 }                                   Shift-
add5 = {x -> x + 5 }                                  Operator
mul3add5 = add5 << mul3
add5mul3 = mul3 << add5
println mul3add5(2)                                   11
println add5mul3(2)                                   21

add5mul3 = add5 >> mul3
println add5mul3(2)                                   21
```

Closure Trampolines

Erster Schritt in Richtung Continuations

- Rekursive Definitionen belasten den Aufruf-Stack sehr stark

- Idee
  - Ein rekursiver Aufruf wird nicht durchgeführt, sondern als Objekt gekapselt und zurückgeliefert.
  - Auf oberster Ebene werden die zurückgelieferten Objekte „ausgeführt“
  - Damit geringere Stack-Belastung
  - „Stack Unwinding“

- Notwendige Vorbedingung: Umformung in endrekursive Form

Rekursive Closure

```
def gauss
gauss = {  n ->
    if (n <= 1) return 1
    else return gauss(n - 1) + n
}
println gauss(100)
```

Endrekursive Closure

```
def gaussER
gaussER = { n, accu ->
    if (n <= 1) return accu + 1
    else return gaussER(n - 1, accu + n)
}
gauss = {n -> gaussER(n, 0) }
println gauss(100)
```

Closure mit Trampolines

```
def gausstr
gausstr = { n, accu ->
    if (n <= 1) return accu + 1
    else return gausstr.trampoline(n - 1, accu + n)
}
gauss = {n -> gausstr.trampoline()(n, 0) }
println gauss(1000)
```

Aufrufcaches für Closures

Closure Memoization

- Erzeugt eine Variante eine Closure mit einem Aufruf-Cache
  - Dieser speichert die Ergebnisse in Abhängigkeit von den Parametern
- Sorgt für sehr schnelle Beantwortung mehrerer Aufrufe mit den gleichen Parametern
- Der Gültigkeit des Caches hängt an der Closure
- Parallele Verwendung in verschiedenen Threads ist möglich

```
def fib
fib = { n ->
    if(n<2) return 1G
    fib(n-1) + fib(n-2)
}

res = fib(30)                    Aufrufzeit 6,188 s

fib = fib.memoize()
res = fib(300)                   Aufrufzeit 0,016 s
```

- Folgende Varianten existieren
  - *memoize()* — unbeschränkte Anzahl Einträge
  - *memoizeAtLeast(n)* — mindestens n Einträge
  - *memoizeAtMost(n)* — maximal n Einträge
  - *memoizeBetween(n, m)* — zwischen n und m Einträge

Verbesserungen bei der Vorbelegung von Parametern in Closures

Neue Methoden für Currying

- Bisher konnten Werte nur von links beginnend belegt werden

```
modulo =  { x, y -> x % y }
mod2 = modulo.rcurry(2)
println mod2(5)                    1
```

- Dies sorgt für eine unnatürliche Folge der Parameter, wenn Vorbelegung von Parametern geplant ist

```
pbtc = { a, b, c -> (a + b) * c }
p3t5 = pbtc.ncurry(1, 3, 5)
println p3t5(2)                    25
```

- Deshalb gibt es zwei neue Methoden
  - *rcurry(val)* — Von rechts beginnend
  - *ncurry(n, val…)* — Mit expliziter Angabe (bei 0 beginnend)

Neue AST-Transformationen

- Allgemein
  - @Log
  - @ListenerList
  - @Field
- Ausführungsbezogen
  - @ThreadInterrupt
  - @TimedInterrupt
  - @ConditionalInterrupt
- Paralleler Zugriff
  - @WithReadLock
  - @WithWriteLock
- Klassenbezogen
  - @Canonical
    - @ToString,
    - @EqualsAndHashCode,
    - @TupleConstructor
  - @InheritConstructor
  - @AutoClone
  - @AutoExternalizable

Allgemeine AST-Transformationen

- *@Log*
  - Injiziert einen Logger (log) und schützt Aufrufe mit Loglevel-Prüfung
  - Log-Level: **info**, **fine**, **finer**, **finest**, **warning**, **severe**
  - Varianten für verschiedene Logger *@Log*, *@Commons*, *@Log4j*, *@Slf4j*
- *@ListenerList*
  - Fügt einer Collection-Property *addListener()*, *removeListener()*, *getListeners()*-Methoden hinzu sowie fire…()-Methoden für jede Methode im beobachteten Objekt
- *@Field*
  - Erlaubt die Definition eines Feldes in einem Groovy-Skript

```
@Log
class LogTest {
    def ausgabe() { log.info "Log-Entry" }
}
t1 = new LogTest()
t1.ausgabe()
```

```
interface Klingel { void glocke(benutzer) }
class ListenerTest {
    @ListenerList(name="Bewohner")
    List<Klingel> listeners
}

t2= new ListenerTest()
t2.addBewohner({println it} as Klingel)
t2.fireGlocke("Postbote")
```

```
@Field
Logger log
```

Ausführungsbezogene AST-Transformationen

- *@ThreadInterrupt*
  - Fügt Abbruchprüfungen ein bei Schleifen (for, while, do), am Anfang von Closures und Methoden.

```
if(java.lang.Thread.currentThread().isInterrupted()) {
    throw new java.lang.InterruptedException(
                                'Execution Interrupted')
}
```

- *@TimedInterrupt (long time)*
  - Fügt Abbruchprüfungen ein mit einem Timeout *time*

```
@TimedInterrupt(10l)
def ewig() {
    def a = 0
    while (true) { a= a+1 }
}
println "Interrupted“
```

- *@ConditionalInterrupt*
  - Verwendet eine mitgelieferte Closure für die Abbruchprüfung

```
@Field counter = 0
@ConditionalInterrupt({ this.counter > 50 })
def e() {
    while(true) println counter++
}
e()
```

- Prüfung kann auf die aktuelle Klasse beschränkt werden (Annotations-parameter *applyToAllClasses*)

AST-Transformationen für kontrollierten parallelen Zugriff

- Schlüsselwort synchronized ist in Fällen mit vielen Lese- und wenigen Schreibzugriffen zu restriktiv
- Dafür gibt es in Java seit 1.5 die Klasse ReentrantReadWriteLock

- Groovy bietet zwei Transformationen für Read- bzw. Write-Locks für Methoden
  - @WithReadLock
  - @WithWriteLock

- Lock-Downgrading nicht ohne Weiteres möglich
- Geht aber „von Hand“ (siehe update()-Methode)

```
class Locked {
    def rwLock = new ReentrantReadWriteLock()
    private final Map data = new HashMap()
    @WithReadLock(value = "rwLock")
    String read(String key) { data.get(key) }
    @WithWriteLock(value = "rwLock")
    void write(key, value) { data[key] = value }
    String update(key, value) {
        rwLock.writeLock().lock();
        try {
                write(key, value)
                rwLock.readLock().lock();
                rwLock.writeLock().unlock();
                return read(key)
        } finally {
                if(rwLock.writeLocked)
                    rwLock.writeLock().unlock();
                rwLock.readLock().unlock();
}}}
```

Klassenbezogene AST-Transformationen (1)

- *@ToString*
  - Fügt eine *toString()*-Methode für alle Felder ein
- *@EqualsAndHashCode*
  - Fügt Methoden *equals()* und *hashCode()* ein
- *@TupleConstructor*
  - Fügt einen Konstruktor hinzu, der entweder mit Parameterliste oder über Tupel-Notation aufgerufen werden kann
- *@Canonical*
  - Fasst *@EqualsAndHashCode*, *@TupleConstructor* und *@ToString* zusammen

```
@Canonical
@TupleConstructor
class Adresse {
    String strasse
    int nummer
    String stadt
}

a1 = new Adresse("Düsseldorfer Straße", 1,
                 "Düsseldorf")
a2 = new Adresse(strasse:"Düsseldorfer Straße",
                 nummer:1,
                 stadt:"Düsseldorf")

assert a1 == a2
println a1
```

```
Adresse(Düsseldorfer Straße, 1, Düsseldorf)
```

Klassenbezogene AST-Transformationen (2)

- *@InheritConstructor*
  - Erzeugt für alle Konstruktoren der Superklasse einen Konstruktor, der diesen aufruft
  - Existierende Konstruktoren werden nicht überschrieben
- *@AutoClone*
  - Aktuell experimentell
  - Implementiert eine *clone()*-Methode, die eine vollständige Kopie des Objekts erzeugt
  - Drei Strategien: Clone, Copy-Constructor und Serializable
- @AutoExternalizable
  - Aktuell experimentell
  - Implementiert Interface *Externalizable* und die Methoden *writeExternal()* und *readExternal()*

```
@InheritConstructors
class CustomException extends Exception {
}
```

```
@AutoExternalize
@AutoClone
class Person {
    String vorname
    String nachname
}

p1 = new Person( vorname:"Dierk",
                 nachname:"Koenig")
p2 = p1.clone()
println p2.vorname
```

Dierk

Groovy++

- Statische Prüfung des Quelltextes
- Bei statischer Übersetzung ähnlich schnell wie Java
- Einfache Mischung statischer und dynamischer Programmteile
- Beinhaltet
  - Typ-Inferenz
  - Implizite Casts
  - Endrekursion
  - Traits (Mixins zur Übersetzungszeit)
  - Extension methods (Categories zur Übersetzungszeit)

Statischer Aufruf

```
@Typed package mypackage

["Hello, ", "World!"].each { current ->
    print current.toLowerCase ()
}
println ()
```

Typinferenz

```
def a = 0, b = 1, c
c = a + b
println c.class
```

java.lang.Integer

Groovy++ - Dynamische und statische Programmteile

- Wenn der statische Compiler Methoden oder Zugriffe nicht auflösen kann, wird dynamischer Zugriff (über *Meta Object Protocol*) verwendet und als Rückgabewert *java.lang.Object* angenommen
- Achtung: Damit auch keine dynamische Manipulation der Aufrufe möglich
- Parameter der *@Typed*-Annotation können zur Unterstützung des Compilers verwendet werden
  - *@Typed(TypePolicy.STATIC)*
  - *@Typed(TypePolicy.DYNAMIC)*
  - *@Typed(TypePolicy.MIXED)*
- Im Normalfall mindestens 2-fache Geschwindigkeit

Statischer Aufruf

```
@Typed(TypePolicy.STATIC)
class GaussS {
    static final BigInteger calc( BigInteger n,
                                  BigInteger accu){
            if (n <= 1) return accu + 1G
            else return calc(n - 1, accu + n)
    }
}
```

Dynamischer Aufruf

```
@Typed(TypePolicy.DYNAMIC)
class GaussD {
    static final BigInteger calc( BigInteger n,
                                  BigInteger accu){
            if (n <= 1) return accu + 1G
            else return calc(n - 1, accu + n)
    }
}
```

Auflösung von Endrekursion mit Groovy++

- Funktioniert mit der neuesten Version von Groovy++ wieder
  - groovypp-0.4.301
  - groovypp-0.4.296 funktioniert nicht (Version im Windows-Installer für Groovy 1.8.1)
- Erreicht automatisch, was in Groovy aktuell über Trampolines manuell gemacht werden kann

```
gaussBench = { calc, val ->
    def start = System.currentTimeMillis()
    calc(val, 0G)
    def now = System.currentTimeMillis()
    (now - start)/val
}
```

Benchmark

```
println "Start"
time = gaussBench(GaussS.&calc, 2500000)
println "Stop: $time"
```

0.00366 / Aufruf

```
println "Start"
time = gaussBench(GaussD.&calc, 2500)
println "Stop: $time"
```

0.0128 / Aufruf

Groovy++ : Traits

- *Traits* in Groovy++ implementieren Interfaces mit Implementierungen von Methoden
- Wenn eine Klasse einen *Trait* implementiert, dann werden Methoden aus dem *Trait* verwendet, wenn die Klasse keine eigene Implementierung bereitstellt
- Ein *Trait* ist eine Mischung von Klasse und Interface
- Wird zur Laufzeit als Interface geführt

```
@Trait
class Persistence {
    def load() { println "Loaded" }
    def save() { println "Saved" }
}
```

```
class MyData implements Persistence {
    def data
    def moreData
    def load() { println "Not loaded" }
}
```

```
md = new MyData()
md.save()                                   Persistence
md.load()                   groovy.lang.GroovyObject
md.class.interfaces.each { println it }

println Persistence.interface                   true
```

Groovy++: Extensions

Categories zur Übersetzungszeit

- Globale Extensions
  - Groovy++ sucht spezifische Dateien im Klassenpfad und verwendet alle dort gelisteten Klassen als globale Extensions
- Extensions, die in der Klasse definiert oder von der Elternklasse geerbt sind
- Annotation der Klasse mit *@Use* stellt Extensions zur Verfügung (Extensions werden vererbt)
  - Wichtig hierbei: Die Klasse muss natürlich statisch übersetzt werden (Annotation *@Typed*)

```
@Typed class HS {
    static hallo(String contents) { "Hallo, $contents" }
    def method(String name) { name.hallo() }
}

println new HS().method("kleiner Prinz")
```

Hallo, kleiner Prinz

```
@Typed
class StringCategory {
    static hallo(String contents) { "Hallo, $contents" }
}
@Use(StringCategory)
@Typed class HSC {
    def method(String name) { name.hallo() }
}
println new HSC().method("kleiner Prinz")
```

Zusammenfassung

- Die neuen Möglichkeiten in Groovy 1.8 bieten noch elegantere Programmierung
- GPars ist jetzt Teil des Groovy-Kerns und erlaubt sehr gute Abstraktionen für parallele Programmierung
- Mehr Fokus auf Performance
  - Deutlich erweiterte Unterstützung der primitiven Datentypen
- Groovy++ bietet die Möglichkeit der statischen Übersetzung und damit Java-ähnliche Geschwindigkeit
  - Verzicht auf Dynamik sehr genau steuerbar
  - Damit alle Vorteile und keine Nachteile
  - Gerade in performance-kritischen Systemen gut einsetzbar

Fazit Groovy ist erwachsen geworden

Gibt es Fragen?