Orientation in Objects

Einführung in Groovy

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTOR

Falk Sippach
Orientation in Objects GmbH

Veröffentlicht am: 8.11.2007

WARUM GROOVY?

Dynamische Skriptsprachen gibt es viele. Es existieren allein über 200 für die Java Virtual Machine. Im Jahre 2003 hat sich auch Groovy aufgemacht, einen Platz in diesem hart umkämpften Markt zu erobern. Was ist das Besondere an dieser noch recht jungen dynamic language und warum sollte man sich damit beschäftigen? Zwei Merkmale sind hier auf jeden Fall zu nennen und insbesondere die Kombination aus beiden macht Groovy im Moment so einzigartig:

- ausdrucksstarke, prägnante Syntax und dynamische Eigenschaften
- perfekte Integration mit Java

Orientation in Objects GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.de info@oio.de

WOHER KOMMT GROOVY?

Javas Erfolg der letzten Jahre ist unbestritten. Er beruht allerdings weniger auf der eigentlichen Sprache und deren Syntax, sondern vielmehr auf der hohen Qualität und der großen Verbreitung seiner Plattform, der Java Virtual Machine (JVM). Die gute Abstraktion der darunter liegenden Maschinen-Architektur ermöglicht die Entwicklung und den stabilen, effizienten Einsatz von Komponenten, Frameworks und Services auf den unterschiedlichsten Betriebssystemen.

Java ist verglichen zu Sprachen wie Ruby, Python, Lisp oder Smalltalk weniger elegant und ausdrucksstark. Groovy versucht diese Lücke zu schließen, in dem es beide Welten zusammenbringt. Mit Groovy wurde übrigens erstmals eine weitere Sprache nach Java für die JVM standardisiert (JSR 241).

Das Groovy-Projekt wurde 2003 von James Strachan und Bob McWriter mit der Intention gegründet, die Eleganz von Ruby auf die Java Plattform zu bringen. Im Januar 2007 erschien das finale Release 1.0 und mittlerweile steht bereits Version 1.1 in den Startlöchern. Das Open Source Projekt ist bei Codehaus gehostet (http://groovy.codehaus.org) und wird von Guillaume Laforge geleitet. Es gibt mittlerweile mehrere Bücher, wobei 'Groovy in Action' von Dierk König et al. als das Standardwerk bezeichnet werden kann. Zu besonderen Ehren kam Groovy im Mai 2007, als ihm der JAX Innovation Award verliehen wurde. Zum Vergleich, im Jahr zuvor ging diese Auszeichnung an das Spring Framework.

WIE SIEHT GROOVY AUS?

Vorher sollten zwei Begriffe geklärt werden, mit denen Groovy immer in Verbindung gebracht wird: dynamisch und Skriptsprache. Unter einem Skript versteht man ein Programm, das in einem Interpreter läuft, und zwar gewöhnlicherweise ohne separaten Kompilierungsschritt. Wenn man es genau nimmt, ist Groovy nach dieser Definition gar keine richtige Skriptsprache, da der Code vor dem Ausführen immer erst in Java Bytecode kompiliert wird. Groovy fühlt sich aber wie eine Skriptsprache an, da die Kompilierung unbemerkt hinter den Kulissen erfolgt. Man hat aber auch explizit die Möglichkeit, die Groovy Quellen in Bytecode zu übersetzen. Da sich Java und Groovy Bytecode nicht unterscheiden, läßt sich bereits die enge Integration zwischen den beiden Sprachen erahnen. Es können sowohl gegenseitig Objekte aufgerufen (Groovy nutzt und erweitert Javas mächtige Klassenbibliothek), als auch Tools wie Profiler, Debugger, etc. ohne Kompromisse in Groovy verwendet werden.

Dynamisch bedeutet, daß Typen von Variablen sowie Verhalten von Objekten erst zur Laufzeit ermittelt werden und bereits kompilierter Code im Stil der aspektorientierten Programmierung (AOP) um neue Funktionalitäten erweitert werden kann. Für den letzten Punkt ist das Meta Object Protocol (MOP) zuständig. Dadurch grenzt sich Groovy von vielen anderen sogenannten dynamischen Skriptsprachen ab, die nämlich nur das dynamische Typisieren unterstützen.

Doch wie sieht die Syntax nun eigentlich aus? Im Beispiel 1 ist das übliche Hello-World Beispiel als Groovy Skript abgedruckt. Dem gegenüber steht in Beispiel 2 die Java Variante dieser simplen Anwendung.

```
println 'Hello Groovy-World!'
```

Beispiel 1: Hello World in Groovy

```
public class SayHello {
  public static void main(String[] args) {
    System.out.println("Hello Java-World!");
  }
}
```

Beispiel 2: Hello World in Java

Zunächst fällt auf, daß der Quellcode von Groovy statt fünf nur eine Zeile umfaßt. Die Syntax scheint also kürzer zu sein, doch ist sie auch ausdrucksstärker und prägnanter wie weiter oben versprochen? Da es sich um eine Art Skriptsprache handelt, muß man keine Klassen oder Programmeinstiegspunkte (`main()` -Methode) schreiben, wie man dies von Java gewöhnt ist. Allerdings ist Groovy genau wie Java eine objektorientierte Sprache. Es geht sogar noch einen Schritt weiter, denn es kennt in Gegensatz zu Java keine primitiven Datentypen sondern nur Objekte. Doch dazu gleich mehr. Groovy-Skripte werden versteckt für den Entwickler durch den Compiler automatisch in Klassen eingepackt und mit einer `main()` -Methode versehen.

Beim nächsten Hinschauen fallen weitere Unterschiede zwischen der einen Zeile Groovy-Code und der entsprechenden Java-Zeile auf. Die Methode 'println' ist eine Art Alias für das längere Java-Konstrukt 'System.out.println' und tut genau das, was man erwarten würde. Des weiteren sollten die fehlenden Klammern und das nicht existierende Semikolon ins Auge springen. Und damit kommen wir gleich zu einem Grundsatz:

Kurze und leserliche Syntax!

Daraus ergeben sich einige Änderungen zu Java. Zum einen importiert Groovy automatisch eine Vielzahl von Packages und oft verwendeten Klassen (`groovy.lang.*, groovy.util.*, java.lang.*, java.util.*, java.net.*, java.io.*, java.math.BigInteger, java.math.BigDecimal`). Java tut dies nur für `java.lang.*`. Des weiteren können Zeichen weggelassen werden, die für den Compiler zur Interpretation des Codes nicht unbedingt nötig sind. Es mag für erfahrene Java Entwickler anfänglich etwas eigenartig aussehen, wenn Klammern, Semikolons, return-Anweisungen, Exception-Deklarationen und public Schlüsselwörter fehlen. Aber nach einer Eingewöhnungsphase wird man die erhöhte Lesbarkeit des Codes schätzen. Dieser Fakt ist übrigens eine Säule für Groovys gute Unterstützung für Domain Specific Languages (DSL).

Nochmals auf unser erstes Beispiel zurückkommend soll ein weiterer Punkt zur Syntax erwähnt werden. Das Java Hello World Beispiel ist nämlich auch ein korrektes Groovy Programm. Die vielen Gemeinsamkeiten zwischen den beiden Sprachen lassen den Gedanken aufkommen, daß Java-Code kompatibel zu Groovy ist. Dies ist allerdings nur bedingt richtig. Vielmehr ist die Syntax von Groovy lediglich an die von Java angelehnt. Es fehlt zum Beispiel das Konstrukt der inneren Klassen. Des weiteren gibt es Javas klassische for-Schleife (`for (int i = 0, i < 10; i++) {[..]}`) erst seit Version 1.1, wurde also bis vor kurzem ebenfalls nicht unterstützt. Nicht zu vergessen ist auch die geänderte Bedeutung gewisser Sprachkonstrukte. So sind Attribute, die in Java als package private deklariert sind (kein Access modifier), in Groovy Properties (getter und setter). Diese wenigen Unterschiede verhindern die direkte Portierung von Java Code. Trotzdem überwiegen die grundsätzlichen Gemeinsamkeiten zwischen den beiden Sprachen und erleichtern Java-Entwicklern den Einstieg in Groovy deutlich.

In den grundlegenden Konzepten folgt Groovy nämlich dem Vorbild und bietet zusätzlich vereinfachte Schreibweisen und nette Zusätze an (syntactic sugar). Kommentare, Pakete, Imports, Klassen, Methoden, Felder und Initialisierungsblöcke sind in ihrer einfachsten Erscheinungsform identisch mit Java. Für Imports gibt es zusätzlich ein Type Aliasing in der Form:

```
import java.awt.BorderLayout as BL
// [..]
def borderLayout = new BL()
```

Beispiel 3: Type Aliasing

Statische Typisierung von Attributen, Methodenparametern und Rückgabewerten kann entfallen, daß heißt, es wird erst zur Laufzeit geprüft, ob ein übergebener Parameter gültig ist. Groovy stellt dafür das Schlüsselwort `def` zur Verfügung. Übrigens wird die Referenz unter der Haube sehr wohl typisiert und zwar als `java.lang.Object`.

Zur Laufzeit verwendet Groovy dann das sogenannte Ducktyping, um Methoden auf untypisierten Referenzen aufzurufen. Das Prinzip dahinter ist einfach erklärt: Alles was wie eine Ente aussieht, wie eine Ente watschelt und wie eine Ente quakt, muß eine Ente sein. Damit Groovy eine bestimmte Methode aufrufen kann, muß diese also nur an dem entsprechenden Objekt vorhanden sein, andernfalls würde ein Laufzeitfehler auftreten. Im untenstehenden Beispiel interessiert Groovy zur Compile-Zeit noch nicht, ob Auto und Bobbycar eine Methode fahren() besitzen. Erst zur Laufzeit wird festgestellt, ob beide Objekte fahren können.

```
class Auto {
 def fahren() {println 'Auto fährt'}
}
class BobbyCar {
 def fahren() {println 'BobbyCar fährt'}
}
def auto = new Auto()
def bobbyCar = new BobbyCar()
[auto, bobbyCar].each {it.fahren}
```

Beispiel 4: Ducktyping

Für Methoden gibt es im Gegensatz zu Java weitere, viel flexiblere Möglichkeiten der Argument-Übergabe. Dazu zählen Parameter mit Vorgabe-Werten, variable Parameter (ähnliches Prinzip wie bei den Varargs in Java 5), benannte Argumente mittels Maps und das Übergeben von Listen bzw. Arrays, die auf die entsprechenden Methoden-Argumente aufgesplittet werden (Spread Operator `*`). Für Konstruktoren gibt es neben den eben aufgezählten noch zwei zusätzliche Varianten. Existiert ein Default-Konstruktor (argumentlos), kann man ähnlich den benannten Argumenten bei Methoden eine Map übergeben. Groovy füllt dann automatisch den Inhalt der Map in die entsprechenden Instanz-Properties, die natürlich existieren und genauso heißen müssen. Die zweite Möglichkeit für Konstruktoren besteht in der Zuweisung einer Liste, welche die Konstruktor-Argumente in der richtigen Reihenfolge enthält.

```
class GroovyExample {
 def a
 def b
 String s
 int i
 GroovyExample() {
 }
 GroovyExample(String s, int i, def a) {
  this.a = a
  this.s = s
  this.i = i
 }
 String toString() {
  "GroovyExample: $a, $b, $s, $i"
 }
 def methodDefaultArgs(int a, def b = new Date(),
  String c = 'abc') {
  "default args: $a, $b, $c"
 }
 def methodVarArgs(int[] a) {
  "variable args: $a"
 }
 def methodNamedArgs(Map args) {
  "named args: $args"
 }
}
def ge1 = new GroovyExample()
assert 'GroovyExample: null, null, null, 0' == ge1.toString()
def ge2 = new GroovyExample('irgendwas', 4, new Date())
try {
 new GroovyExample(new Date(), 'irgendwas, 4')
} catch (Exception e) {
 println 'Constructor doesn\'t exist.'
}
def ge3 = new GroovyExample(a: 4, b: 3, i:5)
GroovyExample ge4 = ['asdf', 3, new Date()]
assert 'default args: 3, http://grails.org, abc' == ge1.
 methodDefaultArgs(3, new URL('http://grails.org'))
assert 'variable args: [1, 2, 5, 6]' == ge1.
 methodVarArgs(1, 2, 5, 6)
assert 'named args: ["alter":35, "name":"Otto"]' == ge1.
 methodNamedArgs(name:'Otto', alter:35)
def args = [4, 'string1', 'string2']
assert 'default args: 4, string1, string2' == ge1.
 methodDefaultArgs(*args) // Spread-Operator
```

Beispiel 5: Argumente in Methoden und Konstruktoren

In einem Punkt grenzt sich Groovy jedoch deutlich von Java ab:

Alles ist ein Objekt!

Groovy verzichtet gänzlich auf primitive Datentypen und verwendet ausschließlich Referenzen auf Objekte. Statt `int` wird so immer das Objekt `Integer` verwendet. Im Code darf man allerdings auch weiterhin `int` schreiben.

In Java ist der Umgang mit Objekten oftmals sehr umständlich und wenig intuitiv. Groovy bietet an vielen Stellen direkte Sprachunterstützung, um die Verwendung oft genutzter Datentypen zu vereinfachen. Zu diesen Datentypen erster Klasse (First Class Citizens) gehören unter anderem Zahlen, Strings, reguläre Ausdrücke, Collections und Closures. Im nachfolgenden Beispiel sieht man, wie dadurch sehr einfach Listen mit der `[]` -Syntax erzeugt werden und wie anschließend auf einzelne Elemente zugegriffen werden kann.

```
def list = [1, 2, 3] // ArrayList mit 3 Werten
def emptyList = [] // leere ArrayList
assert 3 == list[1] + list[2]
```

Beispiel 6: Listen

Ein Grundpfeiler für die First Class Citizens ist das Überladen von Operatoren. In Java gab es damals gute Gründe, dieses mächtige Feature nicht zu implementieren. Aber seit jeher wird über die Vor- und Nachteile diskutiert. In Groovy wurde das Überladen sehr schön gelöst. Zu jedem Operator gibt es eine entsprechende Methode, die man einfach implementieren muß.

Um also zwei Objekte zu addieren, muß es beim Typ des ersten Summanden eine "plus()"-Methode geben, die als Parameter den Typ des zweiten Summanden aufnehmen kann. Auf diese Art können Objekte sehr einfach mittels Operatoren verbunden werden, wie man es von Javas primitiven Datentypen (`int i = 1 + 1`) bzw. Zeichenketten (`"foo" + "bar"`) kennt.

Operation	Operator	Operator-Methode
Addition	a + b	a.plus(b)
Multiplikation	a * b	a.multiply(b)
Inkrementierung	a++	a.next()
Indexierung	a[b]	a.getAt(b)
...		

Tabelle 1: Überladen von Operatoren

Das nächste Beispiel zeigt die Anwendung der Operatoren. Alle vier Zeilen sind inhaltlich gleich. Zu beachten ist, daß das Literal 1 in Groovy einer Referenz auf ein Integer-Objekt mit dem Wert 1 entspricht.

```
assert 2 == new Integer(1).plus(new Integer(1))
assert 2 == new Integer(1) + new Integer(1)
assert 2 == 1 + new Integer(1)
assert 2 == 1 + 1
```

Beispiel 7: Überladen von Operatoren

Bei der Zuweisung von Zahlen verhält sich Groovy übrigens auch etwas anders, als man es von Java gewöhnt ist. Definiert man eine Ganzzahl, wird daraus ein Integer-Objekt erzeugt, so wie man es erwarten würde. Bei Gleitkommazahlen verwendet Groovy allerdings nicht Double sondern BigDecimal. Dadurch wird gerade bei der Entwicklung von finanzorientierter Software die Verwendung von gebrochen rationalen Werten stark vereinfacht. Eine weitere Änderung ergibt sich bei der Division mit Rest. 1 geteilt durch 2 ergibt nun 0.5 statt 0, d. h. Groovy sucht automatisch den sinnvolleren Datentyp heraus. Will man bei der Definition von Zahlen spezielle Typen verwenden, stehen genau wie in Java die entsprechenden Präfixe zur Verfügung (1L .. Long, 1f .. Float, 1G .. BigInteger, ...).

Groovy kennt verschiedene Möglichkeiten, Zeichenketten literal zu definieren. Java am ähnlichsten sind Strings in einfachen und doppelten Hochkommata. In der zweiten Variante, den sogenannten GStrings, können zusätzlich Platzhalter verwendet werden. Allerdings werden die Platzhalter erst ausgewertet, wenn der GString als String verwendet wird. Das ermöglicht die Erstellung flexibler Vorlagen (Templates) oder auch die Erzeugung von echten Prepared Statements ohne die lästige Fragezeichen-Syntax (Groovy-SQL). Groovy's Zeichenkettenunterstützung geht noch weiter. Mehrzeilige Strings erhält man, indem die entsprechenden Hochkommata verdreifacht werden. Und last but not least erleichtert Groovy die Arbeit mit regulären Ausdrücken durch die "slashy syntax". Wird ein String durch Slashes (Schrägstriche) begrenzt, entfällt das Escaping von Sonderzeichen, insbesondere dem bei regulären Ausdrücken sehr oft verwendeten Backslash.

```
def s = 'ein einfacher String'
def gs = "$person.name ist ${person.getAlter} Jahre alt"
sql.execute "select from Person where name=$name"
def zweiZeilen = """erste Zeile
zweite Zeile"""
"\\d\\d:\\d\\d" == /\d\d:\d\d/
```

Beispiel 8: Zeichenketten

Groovy's Unterstützung bei regulären Ausdrücken geht aber noch viel weiter. Es bietet mit dem Negationsoperator (~ -> `negate()`) die Möglichkeit aus einem String ein `java.util.regex.Pattern`-Objekt zu erstellen. Darüber hinaus stellt es einen find-Operator (=~) und einen match-Operator (==~) zu Verfügung. Ersterer erzeugt ein Matcher-Objekt (`java.util.regex.Matcher`) mit allen gefundenen Passagen, während der zweite abprüft, ob die ganze Zeichenkette dem gegebenen Muster entspricht.

```
if ('eine Uhrzeit: 12:23' =~ /\d\d:\d\d/)
  println "enthält mindestens eine Uhrzeit"
if ('zwei Worte' ==~ /\w+ \w+/)
  println "enthält genau zwei Worte"
```

Beispiel 9: Reguläre Ausdrücke

Ein weiterer Stern am Groovy-Himmel ist die viel einfachere Verwendung der Collections. Ein Beispiel für die Listen wurde eben schon gezeigt. Für die Schlüssel-Werte-Paar Container (`java.util.Map`) gibt es ähnliche Deklarationsvarianten:

```
def map = [:] // leere HashMap
map = ["eins":1, "zwei":2]
assert 1 == map["eins"]
map["drei"] = 3
assert 2 == map.zwei
map.vier = 4
```

Beispiel 10: Map

Um auf die Werte zuzugreifen gibt es wieder den `getAt()`-Operator (`[]`). Wenn für die Keys Strings verwendet werden, kann der Zugriff wie auf ein Attribut erfolgen. Zusätzlich wurde mit dem Range ein ganz neuer Collection-Typ eingeführt. Hierbei handelt es sich um Intervalle (Wertebereiche), die durch eine linke und rechte Grenze definiert werden. Die Verwendungen für Ranges sind vielfältig. Sie bieten einerseits eine intuitivere Art des Iterierens (z. B. mit der each-Methode). Des weiteren kann man mit ihrer Hilfe sehr einfach auf Bereiche einer Liste zugreifen.

```
// von 1 bis 5
def range = 1..5
// Reverse Range: von 5 bis 3
def range = 5..3
// Schleife von 1 bis 3
for (i in 1..3) {println i}
// gibt 'abcde' aus
('a'..'e').each {print it}
// -> 3456789, halbexklusiv
(3..<10).each {print it}
// Ergebnis: 'Kleiner'
def substr = 'Kleiner Test'[0..6]
list = [1, 2, 3, 4]
assert [1, 2] == list[0..1]
// Ersetzen der letzten 3 Werte (-3..-1)
list[-3..-1] = ['a', 'b']
assert [1, 'a', 'b'] == list
```

Beispiel 11: Range

Die Kontrollstrukturen funktionieren in Groovy nicht viel anders als in Java, können aber teilweise flexibler eingesetzt werden. Allerdings muß man vor ihrer Anwendung verstehen, wie in Groovy boolesche Ausdrücke verarbeitet werden. Die "Groovy Truth" geht nämlich über simples true und false hinaus. So evaluieren zum Beispiel Collections, Strings und Matcher (reguläre Ausdrücke) zu true, wenn sie nicht leer sind und zu false andernfalls. Zahlen werden als wahr angesehen, wenn ihr Wert ungleich 0 ist. Alle anderen Objekt-Referenzen sind wahr, wenn sie nicht auf null zeigen. Das folgende Beispiel zeigt einige Beispiele.

```
assert true
assert !false
assert [1, 2, 3] // wahr, da nicht leer
assert false == [] // leere Collection
assert 5 // Zahlen sind "wahr", wenn sie ungleich 0 sind
assert !(0)
assert new Date() // true, da ungleich null
assert !(null) // false, da gleich null
if (kunde) {
 println kunde.toString()
}
def i = 5 - 5
if (i) {assert false}
```

Beispiel 12: Groovy Truth

Die Verzweigung if-else funktioniert analog zu Java. Weitaus flexibler gibt sich dagegen die switch-Anweisung. Ihre case-Zweige arbeiten nicht mit primitiven ganzzahligen Datentypen und Enumerationen, wie man es von Java gewöhnt ist, statt dessen kann jedes Objekt verwendet werden. Möglich wird dies durch die Methode isCase(), die Groovy bei der Klasse Object zur Verfügung stellt (und für manche Typen überladt, wie bei Class oder Collection).

```
def x = new Date()
switch(x) {
 case "Groovy" : println "in x steht der String Groovy"
  break
 case Date : println "x ist ein Datum"
  break
 case 1..10 : println "x ist Zahl zwischen 1 und 10"
  break
 default : println "x ist etwas anderes"
}
```

Beispiel 13: Switch

Groovy ergänzt viele Klassen der Java-Bibliothek automatisch mit einer Vielzahl von nützlichen Methoden. Zu diesem Groovy Development Kit (GDK) zählen Methoden für Collections, Datei Ein-/Ausgabe und Literale. Die Beschreibung all dieser Funktionen kann in der Groovy Dokumentation nachgelesen werden. Einige ausgewählte Beispiele werden aber gleich noch vorgestellt.

Die Verwendung von Schleifen in Groovy unterscheidet sich nur im Detail zu Java. Die klassische for-Schleife gibt es zum Beispiel erst seit Version 1.1. Das aus Java 5 bekannte foreach-Konstrukt sieht in Groovy ebenfalls etwas anders aus (Schlüsselwort `in` statt `:`). Hinzu kommt, daß in Groovy mittels GDK-Methoden und Closures sehr einfach Schleifen simuliert werden können, die zudem noch besser lesbar sind. Im folgenden Beispiel sind vier Variationen aufgeführt, die alle zum gleichen Ergebnis führen. Das es in Groovy keine primitiven Datentypen gibt, kann man sehr gut in der dritten Zeile sehen. Dort wird an dem Integer-Objekt mit dem Wert 10 die GDK-Methode `times()` aufgerufen. Diese Methode ruft dann 10mal die als ersten und einzigsten Parameter übergebene Closure auf. Zur Erinnerung, Klammern bei Methodenaufrufen können weggelassen werden, die geschweiften Klammern gehören an dieser Stelle zur Definition der Closure.

```
for (int i = 1; i <= 10; i++) {print i}
for (i in 1..10) {print i}
10.times {print it}
(1..10).each {print it}
```

Beispiel 14: Schleifen

Der Begriff Closure ist in diesem Artikel schon mehrfach gefallen. Java unterstützt dieses Konstrukt bislang nicht direkt, man kann es dort aber mit anonymen inneren Klassen nachbauen. Im Moment ist außerdem eine große Diskussion im Gange, wie man Closures in die Sprache Java direkt integrieren kann (vermutlich ab Java 7). In Groovy gehören Closures bereits zu einem der wertvollsten Features. Ihre Verwendung hat nachhaltigen Einfluß auf den gesamten Programmierstil. Aus Java bekannte Design Patterns werden dadurch entweder gar nicht benötigt oder lassen sich viel einfacher umsetzen.

Eine Closure ist ein wiederverwendbares Stück Code, das irgendwann definiert und erst zu einem späteren Zeitpunkt ausgeführt wird. Da es sich letzten Endes um ein Objekt handelt, kann es referenziert und zum Beispiel als Argument einer Methode übergeben werden. Eine Closure hat allerdings im Gegensatz zu normalen Objekten keinen Zustand, sondern nur Verhalten. Es erinnert dadurch stark an die aus C bekannten Funktionszeiger. Definiert werden Closures als Codeblöcke, die von geschweiften Klammern umschlossen sind. Sie können beim Aufruf Parameter erhalten und natürlich ein Ergebnis zurückliefern. Aufpassen muß man bei der Sichtbarkeit von Variablen, die außerhalb der Closure definiert sind. So verweist das Schlüsselwort `this` in der Closure auf die umschließende Klasse und nicht auf die Closure selbst (eine genauere Betrachtung würde an dieser Stelle allerdings zu weit führen). Wenn man eine Closure definiert hat, kann man sie mit der Methode `call()` aufrufen und ggf. Parameter übergeben. Argumente werden direkt hinter der öffnenden Klammer deklariert und mit einem Pfeil (`->`) vom Body abgetrennt. Wurden keine Argumente deklariert, dann gibt es zumindest den Standardparameter `it`.

```
def c = { println 'I am a closure ' + it }
c.call('test')
('a'..'g').eachWithIndex {item, index ->
   println "$index: $item"
}
```

Beispiel 15: Closures

Im GDK wird sehr viel Gebrauch von Closures gemacht. Um Schleifen zu simulieren, kann, wie weiter oben beschrieben, auf einer Collection die `each()` -Methode aufgerufen und ihr den Schleifenkörper in Form der Closure übergeben werden. Des weiteren kommen Closures beim Eventhandling in der GUI-Programmierung zum Einsatz. Die Groovy-Klasse Closure, die hinter diesen wiederverwendbaren Codeblöcken steht, implementiert außerdem das Interface Runnable, so daß Closures als eigene Threads gestartet werden können.

Eine weitere nette Erweiterung erwartet Groovy-Entwickler bei der Verwendung von GroovyBeans. Sie stellen das Pendant zu den JavaBeans dar. Das sind Klassen, die nach bestimmten Konventionen programmiert sind. Eines der Hauptmerkmale der JavaBeans sind die privaten Membervariablen mit zugeordneten `getter`- und `setter` -Methoden. Je nachdem, wie viele Attribute eine Klasse besitzt, wird es durch den Wildwuchs der viele, meist generierten `getter`- und `setter` sehr schnell unübersichtlich und lenkt von der eigentlich fachlichen Logik der Bean ab.

Groovy geht einen denkbar einfachen Weg. Attribute ohne Angaben zur Sichtbarkeit werden automatisch als Properties, also als private Member behandelt. Zusätzlich werden die entsprechenden `get`- und `set` -Methoden in den Bytecode generiert. Wenn es benötigt wird, kann man natürlich eigene Zugriffsmethoden implementieren, die dann beim Kompilieren nicht von Groovy überschrieben werden.

```
class Person {
 long id;
 String firstName;
 String lastName;
 String getEmail() {
  'info@oio.de'
 }
 // Geschäftsmethoden
}
```

Beispiel 16: GroovyBean

Der Zugriff auf die Felder muß nicht zwangsläufig über die `getter` und `setter` erfolgen. Groovy unterstützt den direkten Zugriff auf die Eigenschaft. Dabei wird aber nicht das Attribut abgefragt (das ist ja `private`). Der Zugriff erfolgt vielmehr automatisch über `getEigenschaft()` bzw. `setEigenschaft()`. Das ermöglicht eine sehr kurze und intuitive Schreibweise. Es geht sogar soweit, daß man eigene `setter` oder `getter` schreiben kann, ohne daß ein entsprechendes Attribut angelegt sein muß. Wie am Beispiel der E-Mail zu sehen ist, kann dann ebenfalls mit `p.email` darauf zugegriffen werden.

```
def p = new Person()
p.setFirstName('Otto')
p.lastName = 'Mustermann'
println p.getFirstName()
println "$p.firstName $p.lastName"
println p.email
```

Beispiel 17: GroovyBean in Aktion

Neben dem GDK, welches die Java Standard-Bibliothek erweitert, liefert Groovy noch jede Menge eigene Bibliotheken mit. Ohne ins Detail zu gehen, sei an dieser Stelle der hervorragende Support für den Zugriff auf Datenbanken, die einfache Verarbeitung von XML und die Erstellung grafischer Benutzeroberflächen (Swing, SWT) genannt. Mit den Groovlets (Groovy Servlet) und dem einfachen Template-Mechanismus (GString, GSP - Groovy Server Pages) kann man sehr leicht kleine Webanwendungen erstellen. Es gibt außerdem eine Handvoll externe Module, wobei hier stellvertretend die gute und vor allem einfache Unterstützung für Webservices erwähnt werden soll.

Ein Teil des XML-Supports und die Erzeugung graphischer Benutzeroberflächen geht auf das Builder-Konzept zurück, welches in Groovy an vielen Stellen zur Anwendung kommt. Builder ermöglichen es auf einfache Art und Weise hierarchische Baumstrukturen (wie XML oder die Anordnung von Komponenten in GUIs) in einer deklarativen Weise zu erzeugen. Im folgenden Beispiel werden einige Zeilen HTML erzeugt. Das Ergebnis ist direkt dahinter abgedruckt.

```
def builder = new groovy.xml.MarkupBuilder()
builder.html {
 head {
  title 'My Homepage'
 }
 body {
  h1 'Welcome'
  p 'Welcome to my Homepage'
 }
}
```

Beispiel 18: MarkupBuilder

```
<html>
  <head>
    <title>My Homepage</title>
  </head>
  <body>
    <h1>Welcome</h1>
    <p>Welcome to my Homepage</p>
  </body>
</html>
```

Beispiel 19: Ergebnis MarkupBuilder

Wie übersichtlich die Entwicklung von graphischen Benutzeroberflächen aussehen kann, zeigt das folgende Beispiel.

```
def swing = new groovy.swing.SwingBuilder()
def frame = swing.frame(title:'Groovy Swing Demo',
size:[300,300]) {
 borderLayout()
 panel(constraints:java.awt.BorderLayout.CENTER) {
  label(text:"Hello World")
  button(text:'Click Me', actionPerformed: {
   println "clicked"
  })
 }
 panel(constraints:java.awt.BorderLayout.SOUTH) {
  label(text:"How are you?")
 }
}
frame.pack()
frame.show()
```

Beispiel 20: SwingBuilder

Abbildung 1: Ergebnis SwingBuilder

Um mit Baumstrukturen arbeiten zu können, muß man über die Knoten navigieren. Auch hier bietet Groovy ein "Goody" an, daß in Anlehnung an XPath GPath heißt. Als Beispiel soll folgender Objektgraph dienen: eine Bank, die mehrere Kunden hat, die wiederum einen Namen und eine E-Mail-Adresse haben. Mit Hilfe von GPath kann man dann sehr einfach alle Kundennamen abfragen:

```
// -> [Otto, Fritz, Heinrich]
println bank.kunden.name
// -> ['otto@mail.de', null, 'h@abc.de']
println bank.kunden.email?.toLowerCase()
```

Beispiel 21: GPath

Wenn man alle E-Mail-Adressen der Kunden umgewandelt in Kleinbuchstaben abfragen möchte, würde man eine `NullpointerException` erhalten, wenn ein Kunde keine E-Mail-Adresse hat. Aber auch hier bietet Groovy ein nettes Gimmick, nämlich das sichere Dereferenzieren mit `?.`. Wenn der Teil des Ausdrucks bereits `null` ist, wertet ihn Groovy nicht mehr weiter aus.

WAS MACHT GROOVY DYNAMISCH?

Kommen wir nun zu den dynamischen Highlights von Groovy. Wenn man in Java eine Methode an einem bestimmten Objekt eines gewissen Typs aufruft, dann wird dieser Aufruf bei der Kompilierung fest eingespeichert. In Groovy ist dies ganz anders. Erst zur Laufzeit wird herausgefunden, welche Methode aufgerufen werden soll. Dies gelingt, weil ein Methodenaufruf durch eine Kette von Framework-Klassen geleitet wird. Dadurch kann jederzeit Einfluß darauf genommen werden, ob der Aufruf überhaupt erfolgen soll oder ggf. an eine ganz andere Stelle umgeleitet wird. Methoden können so auch noch zur Laufzeit in den Aufrufmechanismus eingeklinkt werden, obwohl sie zur Compile-Zeit noch gar nicht existiert haben. Um das dynamische Programmieren mit Groovy richtig zu verstehen, muß man sich zwangsläufig mit den internen Abläufen beschäftigen. Das würde an dieser Stelle zu weit führen. Trotzdem sollen einige Möglichkeiten vorgestellt werden, wie man dynamisch programmieren kann.

Eine Möglichkeit stellen die Expandos dar. Es handelt sich dabei um einen Container für Daten (Zustand) und Methoden (Verhalten), den man quasi zur Laufzeit "befüllen" kann.

```
def duck = new Expando()
duck.name = 'Duffy'
duck.fly = {length -> println "$name flies about $length
km"}
duck.fly(5)
```

Beispiel 22: Expando

In dem Beispiel sieht man, wie das Objekt `duck` nach der Instanziierung ein Attribut `name` mit dem Wert Duffy zugewiesen bekommt. Methoden ergänzt man durch das Setzen einer Closure. Durch seine Eigenschaften eignet sich Expandos sehr gut als Dummy-Objekte für Testzwecke (Stubs, Mocks). Sie haben aber auch einige Einschränkungen. So können sie keine bereits existierenden Klassen erweitern, keine Interfaces implementieren und auch nicht von anderen Klassen erben.

Wie bereits angedeutet erweitert Groovy mit dem GDK bereits bestehende Java-Klassen um neue Funktionen. Diesen Mechanismus kann man in Groovy in ähnlicher Weise mit Hilfe des Schlüsselwort `use` selbst implementieren. Um einer bestehenden Klasse eine Methode hinzuzufügen, muß diese Methode zuerst in einer Kategorieklasse als statische Methode definiert werden. Dabei muß der erste Parameter immer vom Typ des Objekt sein, an welchem später der Aufruf erfolgen soll.

```
class StringCategory {
 static String swapCase(String self) {
  def sb = new StringBuffer()
  self.each {
   sb << (Character.isUpperCase(it as char) ?
    Character.toLowerCase(it as char) :
    Character.toUpperCase(it as char))
  }
  sb.toString()
 }
}
use (StringCategory) {
 assert 'AlITTLEtEST' == 'aLittleTest'.swapCase()
}
```

Beispiel 23: Kategorie

Durch `use` kann man für einen Codeblock das zusätzliche Verhalten an bestimmte Klassen anhängen. Allerdings macht diese Syntax die Verwendung der Kategorien sehr umständlich. Seit Groovy 1.1 gibt es mit der ExpandoMetaClass eine einfachere Variante. Der Unterschied zwischen den beiden Varianten ist, daß Kategorien immer nur in einem bestimmten Thread zusätzliche Funktionalität an Objekte anhängen, während dies die ExpandoMetaClass global für alle Objekte einer bestimmten Klasse tut. Im nachfolgenden Beispiel erfolgt die Implementierung der `swapCase()` -Methode ein zweites Mal, diesmal für alle Objekte der String-Klasse.

```
String.metaClass.swapCase = {->
    def sb = new StringBuffer()
    delegate.each {
        sb << (Character.isUpperCase(it as char) ?
             Character.toLowerCase(it as char) :
             Character.toUpperCase(it as char))
    }
    sb.toString()
}
assert tEsT == 'TeSt'.swapCase()
```

Beispiel 24: ExpandoMetaClass

Die Magie der ExpandoMetaClass endet noch nicht an dieser Stelle. Man kann zum Beispiel auch Konstruktoren, statische Methoden usw. definieren. Durch Groovy's Methoden Pointer Syntax kann eine Klasse sogar Methoden einer anderen Klasse borgen, wie das folgende Beispiel zeigt.

```
class Person {
 String name
}
class Dog {
 def bark() {
  'Bark'
}
}
def lender = new Dog()
Person.metaClass.imitateDog = lender.&bark
def p = new Person()
assert 'Bark' == p.imitateDog()
```

Beispiel 25: Leihen von Methoden

Das Geheimnis hinter all diesen netten Mechanismen nennt sich Meta Object Protocol (MOP). Dieses Protokoll stellt Möglichkeiten zur externen Modifikation des Verhaltens von Objekten zur Verfügung. In Groovy hat jede Klasse eine Metaklasse, durch die man Informationen zu der Klasse erfragen kann. So kann man zum Beispiel herausfinden, ob die Klasse eine bestimmte Methode implementiert bzw. eine Property anbietet. Weiterhin gibt es sogenannte Hook-Methoden:

- `invokeMethod`- Abfangen aller Methoden, die an der Klasse aufgerufen werden
- `methodMissing`- Abfangen aller fehlgeschlagenen Methodenaufrufe (wenn Methode nicht existiert)
- `get/setProperty`- Abfangen von Zugriff auf Properties
- `propertyMissing`- Abfangen von fehlgeschlagenen Property-Zugriffen

Mit Hilfe dieser Hook-Methoden kann man in den Aufrufzyklus eingreifen, Interceptoren schreiben (wie in AOP), oder Aufrufe komplett auf dynamische Implementierungen umleiten.

```
def someObject = 'Test'
// print all the methods
someObject.metaClass.methods.each { println it.name }
// print all the properties
someObject.metaClass.properties.each { println it.name }
```

Beispiel 26: Meta-Informationen zu Klassen

Es geht sogar soweit, daß man aus Java heraus dynamisch Attribute oder Methoden zu Groovy Objekten (abgeleitet von GroovyObject) hinzufügen und aufrufen kann. Allerdings ist diese Arbeitsweise im Gegensatz zum internen Einsatz in Groovy recht umständlich.

MOP stellt auch die Grundlage für das Erstellen von Domain Specific Languages (DSL) dar. Eine DSL ist eine Sprache, die eine knappe, leicht verständliche Syntax zu einer speziellen Thematik (Domäne) anbietet und damit eine höhere Ausdruckskraft erreicht. Sie ist außerdem für die fachlichen Experten leichter erlernbar, als zum Beispiel eine komplexe Programmiersprache. In Groovy kann man sogenannte interne DSLs erzeugen. Dabei handelt es sich um eingebettete Sprachen, welche Sprachelemente der Muttersprache übernehmen.

Eine in Groovy implementierte DSL profitiert von nachfolgenden Punkten:

- ausdrucksstarke und prägnante Syntax
- benannte Parameter bei Methodenaufrufen
- Meta Object Protocol (dynamisch Methoden hinzufügen)
- Überladen von Operatoren
- Closures
- Builder

Ein Beispiel für eine einfache DSL ist im nachfolgenden Beispiel zu sehen. Das Addieren physikalischer Größen mit unterschiedlichen Einheiten basiert hauptsächlich auf dem dynamischen Hinzufügen von Methoden zu der Klasse Number (`getKm()` und `getM()`), die das Umrechnen der Einheiten übernehmen, und der Überladung von Operatoren. Zur Erinnerung, beim Zugriff auf Properties kann man statt `.getKm()` auch direkt `.km` schreiben.

```
println 4.km + 3 * 1000.m + 5.mile // -> 15.05 km
```

Beispiel 27: Verwendung einer einfachen DSL

WOZU IST GROOVY NUN GUT?

Nun soll diskutiert werden, wann und wofür sich der Einsatz von Groovy lohnt. Grundsätzlich läßt sich Groovy überall dort einsetzen, wo im Moment mit Java programmiert wird. Als großen Vorteil kann Groovy die ausdrucksstarke und vor allem prägnante Syntax und seine dynamischen Fähigkeiten in die Waagschale werfen. Dadurch muß nicht nur weniger Quellcode geschrieben werden, der Code wird auch lesbarer und läßt sich einfacher warten. Weniger Code bedeutet übrigens auch weniger Bugs, da laut Untersuchungen die Anzahl der Fehler im Verhältnis der Codezeilen ansteigt.

Natürlich gehen insbesondere die dynamischen Fähigkeiten zu Lasten der Performance. Für kritische Anwendungen ist Groovy dementsprechend nicht empfohlen. Zur Erinnerung, Java galt in seinen Anfangszeiten im Vergleich zu C/C++ als sehr inperformant. Mittlerweile ist der Unterschied nur noch geringfügig und auch bei Groovy wird von Release zu Release an der Performance gearbeitet. Und auch wenn Groovy sicher nie so performant wie eine statische Programmiersprache sein wird, gibt es durch die enge Integration mit Java (gleicher Bytecode) immer die Möglichkeit, performancekritische Teile der Anwendung in Java oder sogar nativ als DLL (Java Native Interface) zu implementieren.

In Vorträgen der Groovy-Macher wird immer wieder von erfolgreichen Einsätzen von Groovy berichtet. So hat eine große amerikanische Versicherungsgesellschaft mittlerweile etwa 50.000 Codezeilen in ein Modul zur Risikoberechnung investiert. Andere große Anwender von Groovy sind das französische Justizministerium oder das Europäische Raumfahrt Konsortium (EADS).

Groovys Anwendungsmöglichkeiten reichen von punktuellen Einsätzen in Java-Programmen, über Scripting zur Laufzeit, Testunterstützung, prototypischen Umsetzungen bis hin zu kompletten Anwendungen und der Integration in Java EE Projekten. Durch die gute Integration zwischen Groovy und Java bietet sich der Einsatz von Groovy speziell für kleine, spezifische Problemstellungen in Java-Anwendungen an. Seine Stärken spielt Groovy zum Beispiel bei der Verarbeitung von Daten aus. Das Parsen von XML mit anschließender GPath-Navigation läßt sich viel kürzer, schneller und intuitiver lösen als mit jeder Java-API. Bei Datei-Zugriffen nimmt Groovy dem Entwickler das Schreiben von Boiler Plate Code ab (z. B. Dateien schließen) und unterstützt ihn außerdem durch viele nützliche GDK-Methoden. Weitere Anwendungsmöglichkeiten sind der Zugriff auf Datenbanken, die Entwicklung von Benutzeroberflächen oder der Einsatz von Webservices.

Ein weiteres Anwendungsgebiet ist das Scripting. Unterpunkte davon sind das Abarbeiten von wiederkehrenden administrativen Aufgaben, die Terminierung von Tasks oder die Verwaltung von Projekt-Infrastrukturen mittels Build-Skripten (Ant). Für Windows Benutzer gibt es eine Bibliothek names Scriptom, die aufbauend auf der Java COM Bridge (Jacob) zum Beispiel das Fernsteuern von Microsoft Office erlaubt. An dieser Stelle nicht zu vergessen sind die sogenannten Laufzeit-Makros. In OpenOffice kann Groovy als Makrosprache eingesetzt werden. Denkbar wäre auch, dem Benutzer in einer laufenden Java-Anwendung die Möglichkeit für das Ausführen von Skripten zu geben, z. B. die Eingabe von Funktionen in einer Mathematik-Software (f(x) = sin(x)). Der Groovy-Code könnte aber auch genauso gut aus Datenbanken kommen oder über das Internet nachgeladen werden. So wäre die Codebasis zentralisiert und der Anwender würde immer den aktuellsten Stand ausführen, ohne explizit lokale Updates durchführen zu müssen. Um Groovy Skripte zur Laufzeit auszuführen, gibt es übrigens mehrere Möglichkeiten. Seit Java 6 wird zum einen eine standardisierte Schnittstelle für die Einbettung von Skriptsprachen mitgeliefert (JSR 223). Groovy selbst bietet mit der GroovyShell, der GroovyScriptingEngine und dem GroovyClassLoader drei weitere Möglichkeiten an.

Testen wird in Groovy ebenfalls zum Kinderspiel und qualifiziert Groovy damit auch für die testgetriebene Entwicklung (TDD). Neben den in den Sprachumfang eingebauten "assert"-Anweisungen (im Gegensatz zu Java sind sie nicht abschaltbar), liefert Groovy einen speziellen JUnit TestCase mit erweiterten "assert"- und "fail"-Methoden. Und allein durch die schon mehrfach angesprochene prägnante Syntax wird der Testcode verkürzt und im Hinblick auf Testdokumentation im wahrsten Sinne des Worte lesbarer. Durch seine dynamischen Fähigkeiten kann es zusätzlich bei der Erzeugung von Stub- und Mock-Objekten glänzen. Ohne weitere Bibliotheken kann man in Groovy dank des MOP Stellvertreter erzeugen und damit kleine Einheiten entkoppelt von ihren Abhängigkeiten in echter Unit Manier testen.

Last but not least können in Groovy natürlich ganze Anwendungen entwickelt werden. Groovy erlaubt eine schnelle und effiziente Entwicklung und eignet sich daher hervorragend zum Erstellen von Prototypen. Aufgrund der engen Integration in Java muß man diese Prototypen dann aber nicht komplett neu implementieren. Vielmehr kann man sehr einfach darauf aufbauen und neue Module in Java oder Groovy entwickeln. Gerade performancekritischere Teile wird man eher in Java entwickeln. Bei der Integration in Java EE Projekte bietet sich Groovy besonders auf zwei Arten an. Einerseits kann man es als eine Art Klebeschicht zwischen den Schichten sehen. Mit Groovy kann ein System programmatisch konfiguriert werden (weniger XML), es kann die Oberfläche bzw. die komplette Anwendung angepasst werden (Customizing) und es können sehr leicht Änderungen am laufenden System durchgeführt werden. Der letzte Punkt umfasst administrative Eingriffe ala JMX, wie Fehlersuche am Live-System (Admin-Konsole) und das Einspielen von Hotfixes zur Laufzeit. Dies funktioniert alles ohne lästiges Kompilieren und Deployen. Jeder, der schon EJB-Anwendungen entwickelt hat, kann ein Lied davon singen, wieviel Zeit bei den Compile-Deploy-Zyklen verloren geht. Andererseits kann Groovy aber auch für die Geschäftslogik eingesetzt werden und zum Beispiel über eine DSL (Domain Specific Language) Geschäftsregeln auslagern. DSLs sind ohnehin sehr vielseitig einsetzbar und können als Fundament für die Benutzeroberfläche, für Tests und die eigentliche Geschäftslogik dienen.

FAZIT

Dieser Artikel sollte eine Einführung in Groovy geben. Die Mächtigkeit dieser noch relativ jungen Sprache läßt sich vermutlich nur erahnen, da in diesem Rahmen bestenfalls an der Oberfläche der vielen nützlichen Features gekratzt werden konnte. Die Einsatzmöglichkeiten sind vielfältig und die Einstiegshürde liegt insbesondere für Java-Entwickler sehr niedrig. Wer Groovy ausprobiert, wird sich vermutlich schnell verlieben.

Ein weiteres spannendes Thema ist das Webframework Grails, welches technisch auf Groovy und vielen bekannten Java APIs (Hibernate, Spring, ...) aufbaut. Zu den Grundsätzen von Grails gehören DRY (Don't repeat yourself) und die leichte Einarbeitung durch einheitliche Vorgaben (Convention over Configuration). In wenigen Schritten erlaubt es die Erstellung einer kompletten Webanwendung. Ohne lästige Deploy-Zyklen wird ein produktives, zügiges Entwickeln gefördert. Grails eignet sich daher unter anderem für Rapid Prototyping, aber auch für professionelle Webanwendungen.

REFERENZEN

- [1] Groovy Homepage
  (http://groovy.codehaus.org)
- [2] Grails Homepage
  (http://grails.org)
- [3] Groovy in Action
  König, Dierk et al.
  (http://www.manning.com/koenig)
- [4] Groovy für Java-Entwickler
  Staudemeyer, Jörg
  (http://www.oreilly.de/catalog/groovyger)
- [5] groovy.blogs
  (http://groovyblogs.org)
- [6] AboutGroovy
  (http://aboutgroovy.com)