exensio
intelligente informationssysteme

Eine Groovy-Einführung

1. September 2009

Für die Java Virtual Machine (JVM) existieren eine Vielzahl von Komplementärsprachen. Unter den objektorientierten Programmiersprachen ist besonders Groovy durch seine Anlehnung an Python, Ruby, Perl und insbesondere Java aufgefallen. Diese Einführung gibt einen kurzen Überblick über den Funktionsumfang, die Eigenschaften und die Nutzung von Groovy.

Inhaltsverzeichnis

1. Einführung

Ist die Programmiersprache Groovy einen genauen Blick wert? Allein das gute Zusammenspiel mit *Java* macht neugierig. Doch welche weiteren Qualitäten und welche Reife besitzt diese junge Sprache, die viele Ideen von *Perl* [1], *Ruby* [2], *Python* [3] und *Java* [4] in sich vereint? Neben dynamischen Typen, einer mächtigen Syntax und der Möglichkeit vorhandene Java-Klassenbibliotheken zu verwenden, gibt es viel zu entdecken. Tauchen wir also ein in die Welt von Groovy!

2. Ein Teil von Java?

Obwohl von vielen als Skriptsprache bezeichnet, ist Groovy mehr. Groovy-Code läuft innerhalb der *JVM* ab und wird daher vor der Ausführung in *Java-Bytecode* übersetzt. Dies geschieht jedoch meist vom Nutzer unbemerkt, direkt vor dessen Verwendung. Dadurch fühlt sich Groovy wie eine Skriptsprache an und bereits einzeilige Programme können an den Interpreter übergeben werden. Bei Bedarf kann jedoch auch zu den aus Java bekannten *Klassen-Dateien* (engl.: „Class Files") kompiliert werden. Da die *JVM* Java- und Groovy-Bytecode gleich behandelt, können die für Java zur Verfügung stehenden APIs (Programmierschnittstellen, engl.: „Application Programming Interface") wie die *JDBC* (Java Database Connectivity) [5] sofort mit Groovy genutzt werden.
Bereits vorhandener und bewährter Java-Code fügt sich fast nahtlos in neue Programme ein. Zu beachten ist, dass Groovy bei der Auswertung von booleschen Werten (Wahrheitswerten) und bestimmten Kontrollflussstrukturen anders als Java vorgeht, doch dazu später mehr. Trotz allem ist es möglich, von Javas weiter Verbreitung, mächtigen Klassenbibliotheken und der API-Unterstützung zu profitieren. Auch syntaktisch lehnt sich Groovy stark an Java an. Dadurch sind bestehende Vorkenntnisse hilfreich, aber aufgrund der einfachen Erlernbarkeit von Groovy nicht zwingend nötig.

3. Keine primären Datentypen

Im Gegensatz zu Java verzichtet Groovy auf die Verwendung von primären Datentypen und erzeugt direkt Instanzen der Klasse *java.lang.Object*. Werden dennoch primäre Datentypen verwendet, entstehen automatisch Objekte des entsprechenden Wrapper-Typs. Dies erleichtert den direkten Umgang mit den verschiedenen Objektbehältern wie Maps und Listen.
Die Verwendung von arithmetischen Operatoren ist weiterhin möglich. Daher sind folgende äquivalente Schreibweisen gültig:

```
view plain  copy to clipboard  print  ?
01.  // Alles ist ein Objekt.
02.  result = 2 + 4
03.  result1 = 2.plus(4)
04.  assert result == result1
```

4. Automatische Importe und dynamische Datentypen

Neben den aus *Java* bekannten standardmäßigen Import des Funktionsumfangs aus *java.lang*, geht Groovy einen Schritt weiter und fügt außerdem jedem Programm die folgenden Pakete hinzu:

```
view plain  copy to clipboard  print  ?
01.  * java.io.*
02.  * java.lang.*
03.  * java.math.BigDecimal
04.  * java.math.BigInteger
05.  * java.net.*
06.  * java.util.*
07.  * groovy.lang.*
08.  * groovy.util.*
```

Die Einführung des neuen Schlüsselworts *def* ermöglicht die Definition von Variablen, deren Typ erst zur Laufzeit ermittelt wird. Dieser kann sich, auch innerhalb eines Programms, durch die erneute Zuweisung eines anderen Typen ändern.

```
view plain  copy to clipboard  print  ?
01.  groovy> // dynamische Datentypen
02.  groovy> def v = "Ich bin ein Text"
03.  groovy> println v.getClass()
04.  groovy> v = 3
05.  groovy> println v.getClass()
06.  groovy> v = 1.34
07.  groovy> println v.getClass()
08.  class java.lang.String
09.  class java.lang.Integer
10.  class java.math.BigDecimal
```

An der letzten Zeile des Beispiels ist zu erkennen, dass Groovy für Gleitkommazahlen nicht wie erwartet Doubles sondern BigDecimals verwendet. Da Groovy nur Objekte kennt, ist es möglich, auf die jeweils für diesen Typ vorhandenen Methoden direkt ohne Typecast zuzuzgreifen.

5. Zeichenketten

Der Umgang mit Zeichenketten (engl.: „Strings“) wird in Groovy ggü. Java erheblich erleichtert. Gewöhnliche Zeichenketten werden durch Hochkommata maskiert. Neu sind die so genannten *Gstrings*. Sie werden durch Anführungszeichen gekennzeichnet und bieten die Möglichkeit, Variablen und Ausdrücke in sich aufzunehmen. Außerdem kann durch drei aufeinanderfolgende Hochkommata bzw. Anführungszeichen ein String bzw. GString über mehrere Zeilen definiert werden, wie in der folgenden Abbildung dargestellt:

```
view plain  copy to clipboard  print  ?
01.  def s1 = 'Ein normaler String'
02.
03.  // s2 enthält einen GString
04.  def name = 'Herr Kroger'
05.  def s2 = "Hallo $name"
06.  def s3 = '''Dieser String
07.  geht über
08.  mehere Zeilen'''
```

Eine weitere Besonderheit unter den Zeichenketten bilden die regulären Ausdrücke (engl.: „Regular Expressions“). Sie werden durch Schrägstriche begrenzt. Innerhalb eines solchen regulären Ausdrucks ist es nicht mehr notwendig, einzelne Sonderzeichen durch einen Backslash darzustellen. Dies trägt erheblich zur Lesbarkeit und zum Verständnis bei.

```
view plain  copy to clipboard  print  ?
01.  def rege = /\w+-\w+-\d+/
02.  assert "\\w+-\\w+-\\d+" == rege
03.  def match = "KA-SI-12" =~ rege
04.  if(match.matches())
05.  println "Es handelt sich um ein Kennzeichen."
```

6. Vereinfachter Umgang mit Datentypen

Die Nutzung einiger Datentypen wird für Programmierer einfacher und intuitiver gestaltet. Zum Einen übernehmen Listen die Aufgaben der aus Java bekannten Arrays. Sie werden auf folgende Art und Weise deklariert.

```
view plain  copy to clipboard  print  ?
```

```
01. // Deklaration einer Liste
02. List list1 = ['a', 'b', 'c']
```

Einzelne Elemente können mit dem „+"-Operator hinzugefügt werden.

```
view plain  copy to clipboard  print  ?
01. // Einfügen von neuen Elementen mit dem + Operator
02. list1 = list1 + 'd'
```

Der „*"-Operator resultiert in einer Vervielfältigung der Liste.

```
view plain  copy to clipboard  print  ?
01. // Vervielfältigung von Listen durch den * Operator
02. list1 = list1*2
03. assert list1 == ['a', 'b', 'c', 'd', 'a', 'b', 'c', 'd']
```

Als zweiter Stellvertreter verdienen die verfügbaren *Maps* eine besondere Erwähnung, die den aus Java bekannten *Hashmaps* ähneln. Sie ermöglichen das Bündeln von verschiedenen Werten und Datentypen, die jeweils durch einen Schlüssel ausgezeichnet werden.

```
view plain  copy to clipboard  print  ?
01. // Deklaration einer Map
02. Map person = ['name':"Heinz Kroger", 'age': 31, 'married':true]
```

Der Zugriff ist zum Einen über das Schlüsselwort selbst möglich. Sollte es sich dabei jedoch um einen String handeln, kann auf die einzelnen Elemente wie auf Attribute eines Objekts zugegriffen werden.

```
view plain  copy to clipboard  print  ?
01. // 1. Zugriff über Schlüsselwort, 2. Zugriff als Objekteigenschaft
02. person['hometown'] = 'Karlsruhe'
03. assert person.age == 33
```

Für beide Datentypen stehen besondere Funktionen, wie z.B. die *each()*-Methode zur Verfügung, die das Iterieren über alle Listenelemente ermöglicht.

```
view plain  copy to clipboard  print  ?
01. // Iterieren über eine Liste mit der each() Methode
02. person.each { property, value ->
03.    println "Die Eigenschaft $property besitzt den Wert $value."
04. }
```

Eine Neuerung für Java-Programmierer stellen die *Bereiche* (engl.: „Ranges") dar. Sie können für alle Objekte verwendet werden, die die Methode *compareTo()* aus dem Paket java.lang.Comparable bereitstellen. Anwendung finden Sie auch innerhalb von Kontrollstrukturen. Sie können zum Beispiel bei der Falleinteilung von Switch-Anweisung genutzt werden.

```
view plain  copy to clipboard  print  ?
01. // Deklaration eines Bereichs
02. def range1 = 1..5
03. // Rückwerts verlaufender Bereich
04. def range2 = 5..1
05. // Falleinteilung in einer Switch-Anweisung mit Bereichen
06. switch (age) {
07.    case 1..6: println "Noch nicht geschäftsfähig!"; break
08.    case 7..17: println "Beschränkt geschäftsfähig!"; break
09.    case 18..150: println "Geschäftsfähig!"; break
10.    default: println "Bitte ein korrektes Alter angeben!"
11.
12. }
```

Eine weitere Erleichterung beim Umgang mit Datentypen wird durch neue Methoden ermöglicht, die das *GDK* (engl.: „Groovy Developement Kit") zur Verfügung stellt. So werden neue Funktionen ergänzt oder bestimmt Funktionen syntaktisch vereinheitlich. Abbildung 1 stellt dies am Beispiel der Groovy-Funktion *size()* dar.

Objekt	**Java**	**Groovy**
Array	length field	size()
Array	java.lang.reflect.Array.getLength(Array)	size()
String	length()	size()

StringBuffer	length()	size()
Collection	size()	size()
Map	size()	size()
File	length()	size()
Matcher	groupCount()	size()

Abbildung 1: size() in Groovy

7. Closures

Häufige Anwendung finden auch die so genannten *Closures*. Dabei handelt es sich um wiederverwendbare Codestücke, mit deren Hilfe Anweisungen, ähnlich wie Funktionen, zur späteren Wiederverwendung definiert werden können. Da es sich bei ihnen um einzelne Objekte handelt, ist es auch möglich, Closures als Funktionsparameter zu übergeben. Sie können zum Beispiel in Verbindung mit der vorgestellten *each()*-Methode verwendet werden. Eine Closure selbst kann auch Parameter erhalten oder zurückgeben. Mit dem Schlüsselwort *it* wird allgemein auf den aktuell übergebenen Wert verwiesen.

```
view plain  copy to clipboard  print  ?
01. // Definition einer Closure
02. def adder = { x, y -> return x + y}
03. List list1 = [1, 2, 3]
04. // Das Schlüsselwort it greift auf den aktuellen Parameter zu
05. def printChangedSigns = { println "Der neue Wert ist ${-it}." }
06. // Closure als Methodenparameter übergeben.
07. list1.each(printChangedSigns)
```

8. Currying

Ein weitere besondere Verwendungsmöglichkeit für Closures stellt das so genannte *Currying* dar. Mit dessen Hilfe ist es schnell möglich, Parameter der Closure mit Werten vorzubelegen. Auch wenn die Vorteile dieses Prinzips nicht sofort zu erkennen sind, wird nach wenigen praktischen Anwendungen schnell deutlich, dass es sich dabei um eine flexible, schnelle und übersichtliche Methode der Codewiederverwendung handelt.

9. Kontrollstrukturen

Im Gegensatz zu Java finden in Groovy einige neue Kontrollfluss-Strukturen wie der „?"-Operator Anwendung oder werden auf einer andere Art und Weise ausgewertet.
Der „?"-Operator ermöglicht die kurze und übersichtliche Überprüfung der Existenz von Parametern. Dadurch bleibt besonders bei der Arbeit mit vielen möglichen Null-Referenzen der Programmcode übersichtlich.

```
view plain  copy to clipboard  print  ?
01. // Ist "name" belegt, wird der Teil vor dem Doppelpunkt ausgeführt,
02. // ansonsten der Teil dahinter
03. def greeting = name ? "Hallo $name!" : "Willkommen!"
```

10. Leistung

Die komfortable Unterstützung, die Groovy dem Programmierer bietet, wird an einigen Stellen mit Leistungseinbußen bezahlt. Daher sollte, falls nötig, die richtige Art der Umsetzung gewählt werden. So ist beispielsweise, im Gegensatz zu normalen Strings, die Auswertung der zuvor behandelten Gstrings aufwändiger. In bestimmten Bereichen liegt die Leistung von Groovyprogrammen auch auf Grund von neu gebotenen Möglichkeiten noch deutlich hinter den Java-Gegenstücken. Dazu gehört zum Beispiel auch die Fähigkeit von Objekten, zur Laufzeit Informationen über sich selbst zu erhalten (engl.: „Reflections").
Zu beachten ist jedoch, dass sich Groovy in der momentanen Version stetig weiterentwickelt und dabei konstante Leistungsverbesserungen erfährt. Des Weiteren besteht, für besonderes leistungsbeanspruchende Programmteile, noch stets die Möglichkeit, sie als Javacode zu importieren.

11. IDE-Unterstützung

Die Unterstützung durch *integrierte Entwicklungsumgebungen* (IDE, engl.: „Integrated Development Environment") für Groovy bietet leider noch nicht den von *Java* bekannten Komfort und Umfang. Zum Erlernen der Sprache bietet sich die in der Groovy-Distribution erhaltene Groovy Konsole an. Als Entwickler wird man jedoch bald nicht auf die Unterstützung einer Entwicklungsumgebung verzichten möchten. Den mächtigsten Funktionsumfang bietet dabei momentan die kostenpflichtige Entwicklungsumgebung *IntelliJ IDEA*. Unter den kostenlosen Vertretern befinden sich *Eclipse* und *Netbeans*, wobei sich die für Sie erhältlichen Plugins in einer konsequenten und schnellen Weiterentwicklung befinden. Während sich Eclipse auf die Unterstützung von Groovy beschränkt, bietet Netbeans gleichzeitig einen erweiterten Funktionsumfang für die komfortable Nutzung des Webframeworks *Grails*.

12. Fazit

Wie dargestellt, schafft es Groovy eine Vielzahl von Vorteilen in sich zu vereinen. So resultiert die große Ausdrucksstärke von Groovy, in einem oft um 30% kürzeren Programmcode als sein Java Gegenstück. Die Entwicklungsdauer wird maßgeblich verkürzt.
Das nachträglich Auffinden von Fehlern gehört zu einem der zeitaufwendigsten und kostenintensivsten Faktoren in der Softwareentwicklung. Besonders bei erfahrenen Softwareentwicklern neigt die Fehleranzahl pro Programm-Zeile (LOC, engl.: „Lines of Code") dazu, sich auf ein konstantes Niveau einzupendeln. Dies führt dazu, dass eine Implementierung der gleichen Funktionalität in Groovy häufig eine geringere Fehleranzahl aufweist.
Die Komprimierung des Quellcodes macht Programme zudem übersichtlicher, was ihre spätere Wartung und Pflege deutlich erleichtert.
Die exzellente Integration mit Java, erlaubt zudem vorhandenes Wissen weiter zu verwenden und falls gewünscht bestimmte Programmteile, mit beispielweise leistungskritischen Aspekt, in Java zu verfassen.
All diese und viele weitere Eigenschaften, führten zu einer im größeren Nutzung und Verbreitung von Groovy und zum Gewinn des ersten Platzes des *JAX Innovation Award 2007*.

Referenzen

[1] Perl Programmiersprache:
http://www.perl.org

[2] Ruby Programmiersprache:
http://www.ruby-lang.org

[3] Python Programmiersprache:
http://www.python.org/

[4] Java Programmiersprache:
http://java.sun.com/

[5] JDBC:
http://java.sun.com/products/jdbc/overview.html

Jörn Kuhlenkamp