Groovy Syntax und Beispiele

Nebst den hier am häufigsten genutzten Elementen gibt es noch viele weitere Funktionen und Helfer, siehe dazu auch die offizielle Doku (auf die Version achten).

Syntax

Klammern um Parameter sowie das Semikolon am Ende der Zeile können weggelassen werden. Empfehlung: Klammern verwenden, Semikolon weglassen.

```
println "Hello, World"
println("Hello, World") // Empfehlung
println("Hello, World");
```

Typen müssen nicht explizit angegeben werden (`def` entspricht quasi `Object` in Java bzw. `var` in JavaScript). Empfehlung: Typen explizit angeben, falls die IDE sie nicht automatisch erkennt und demtentsprechend keine sinnvollen Vorschläge für Syntax-Completion macht.

```
def x = 11.8 // erst Zahl
println(x)

x = "Test" // dann String
println(x.toUpperCase())
```

Typen werden, soweit möglich automatisch umgewandelt, es müssen keine expliziten Casts verwendet werden (hier kann aber das Schlüsselwort `as` verwendet werden). Es erfolgt auch ein automatisches Boxing / Unboxing (int <--> Integer etc.).

```
def i = "5" as int
eineListe.add(12)
```

Der Gleichheits-Operator in Groovy (`==`) wird im Hintergrund zu einem `.equals()` Aufruf umgewandelt.

```
String a = "test"
String b = "te" + "st"
println(a == b) // true, in Java: a.equals(b)
```

Strings können von Anführungszeichen (`"`), Apostrophen (`'`) oder Slashes (`/`) umschlossen sein. Empfehlung: Das verwenden, was im String nicht vorkommt, um nicht escapen zu müssen (Slashes, falls Anführungszeichen und Apostrophe vorkommen bzw. für Reguläre Ausdrücke).

```
"str" == 'str' == /str/
```

Es gibt auch Mehrzeilige Strings. Ein Backslash (`\`) bewirkt, dass der Zeilenumbruch vor der ersten Zeile im Code nicht mit ausgegeben wird.

```
println("""
Zeile 1
Zeile 2
""")

println("""\
einzige Zeile""")
```

In Strings mit Anführungszeichen (einfach oder dreifach für mehrzeilige Strings) können Code-Platzhalter verwendet werden, die durch ihren Wert ersetzt werden.

```
def user = "alice"
println("Hi $user")
println("aktueller Timestamp: ${new Date().time}")
```

Closures

Closures entsprechen in etwa den `function`s in JavaScript oder Lambda-Funktionen in Java 8. Es muss dabei nicht explizit `return` angegeben werden, der Rückgabewert des letzten Statements wird als Rückgabewert der Closure benutzt. Parameter müssen nicht typisiert werden.

```
// keine Parameter
def c1 = {
  "Hello"
}
c1()

def c2 = { name ->
  "Hello, $name!"
}
c2("World")
```

Methoden, die eine Closure als einzigen oder letzten Parameter akzeptieren, kann die Closure direkt übergeben werden (keine runden Klammern nötig).

```
"...".eachLine { line ->
  println(line.reverse())
}
```

Falls die Closure genau ein Parameter übergeben bekommt, so muss dieser nicht benannt werden, der Standardname ist "`it`".

```
"...".eachLine {
  println(it.reverse())
}
```

Spaceship-Operator

Vergleiche (`compareTo`) erledigt der Spaceship-Operator, das vereinfacht die Implementation eines `Comparator`s.

```
println 5 <=> 10         // Java: 5.compareTo(10)
println "def" <=> "abc"  // Java: "def".compareTo("abc")
println null <=> 10      // gibt keine Exception
```

Beispiel: System Properties filtern

```
System.properties
  .findAll { it.key.startsWith("user") }
  .sort { x, y -> x.key <=> y.key }
  .each { println it }
```

Get / Set

Es werden automatisch JavaBeans-Accessors erzeugt bzw. angesprochen, d.h. man muss nicht `getProp()` oder `setProp()` aufrufen, sondern kann einfach `prop` schreiben.

```
// Lesen
println(obj.thing)   // entspricht obj.getThing()
println("...".empty) // entspricht "...".isEmpty()
```

```
// Schreiben
obj.thing = null     // entspricht obj.setThing(null)
```

Arrays / Lists / Maps / Ranges / Collections

Für Listen und Maps gibt es eine native Syntax.

Liste

```
def a = [1, 2, 3]  // eine ArrayList im Hintergrund
println(a[1])      // einfach auf Elemente zugreifen

println([].size()) // eine leere Liste
```

Map

```
def m = [language: "Groovy", version: 2.1] // eine LinkedHashMap
println m["language"]  // einfach auf Elemente zugreifen
println m.version      // einfach auf Elemente zugreifen

println([:].size())    // eine leere Map
```

Mit Ranges kann man einfach Zahlenbereiche generieren.

```
(1..10).each { println(it) }  // 1 bis 10
(1..<10).each { println(it) } // 1 bis 9
(5..-5).each { println(it) }  // absteigend
```

Außerdem können mittels Ranges teile von Strings und Collections extrahiert werden.

```
def a = [1, 2, 3, "a", "b", "c"]
println a[-2]     // b
println a[2..4]   // [3, a, b]
println a[2..1]   // [3, 2]
println a[-2..2]  // [b, a, 3]

def s = "the quick brown fox"
```

```
println s[4..9, -3..-1]   // quick fox
```

Um zu überprüfen, ob eine Collection ein Element enthält, kann der `in` Operator verwendet werden.

```
println(5 in [1, 5, 10]) // true
```

Diesen gibt es auch in der verkürzten Variante der `for`-Schleife.

```
def a = [1, 5, 10]
for (i in a) {
  println(i)
}
```

Groovy bringt viele Hilfsmethoden für Collections mit.

```
[5, 4, 8, 1, 4, 5, 2].sort().unique().reverse()   // Sortieren, Doppler entfernen, Reihenfolge
invertieren
(1..10).take(3).drop(1)                           // nur die ersten 3 Elemente nehmen und
davon dann das erste wegschmeißen
[4, 7, 1].min()                                   // Minimum finden
[cents: 5, dime: 2, quarter: 3].max { it.value }  // Maximum finden und dabei festlegen, was
verglichen werden soll pro Element
(1..100).sum()                                    // Summe bilden
["abc", "def", "abcdef"].findAll { it.startsWith("a") }  // alle Elemente finden, die einer
Bedingung entsprechen
["abc", "abcdef"].collect { it.length() } == [3, 6]      // Elemente transformieren

['a', 'b'] << [1, 2]    // [a, b, [1, 2]]   Element anhängen
['a', 'b'] + [1, 2]     // [a, b, 1, 2]     Listen zusammenführen
[1, 2, 3].join("~")   // 1~2~3     Elemente mittels Trenner zusammenführen

def a = [1, 2, 3, 4, 5]
println(a.any { it > 3 })    // true, erfüllt mindestens ein Element die Bedingung?
println(a.every { it < 5 })  // false, erfüllen alle Elemente die Bedingung?
```

Wenn man nicht immer prüfen will, ob ein Key in einer Map ist, bevor man ihn benutzt, kann man ein Default-Wert festlegen. Dieser wird dann unter dem Key automatisch in der Map angelegt, sobald das erste Mal auf ihn zugegriffen wird.

```
def map = [:].withDefault {
  new Date()
}

println map.test  // Zeit 1
sleep(2000);
println map.test  // immer noch Zeit 1
println map.test2 // Zeit 2
```

RegEx

Reguläre Ausdrücke haben auch eine verkürzte Syntax. Slashes werden hier nur der Lesbarkeit / Identifizierbarkeit verwendet, es sind immer noch normale Strings (nicht wie in JavaScript, wo mit Slashes direkt reguläre Ausdrücke erzeugt werden).

```
// Pattern erzeugen
Pattern p = ~/\d+/
println(p.matcher("123").matches())

// Matcher erzeugen
Matcher m = "123" =~ /\d(\d)\d/
println(m.matches())
println(m.group(1))

// Direkt matchen
println("123" ==~ /\d+/) // true
```

Null-Dereferenzierung

Um NullPointerExceptions vorzubeugen, ohne dauernd mit `if` auf `null` prüfen zu müssen, empfiehlt sich der `?.` Operator. Dieser stoppt, sobald der Ausdruck vor dem Fragezeichen `null` ist und gibt `null` zurück.

```
class Person {
  def name
}
```

```
def p
println(p?.name?.length())  // null

p = new Person()
println(p?.name?.length())  // null

p.name = "Tester"
println(p?.name?.length())  // 6
```

Elvis-Operator

Default Werte können mit dem Elvis-Operator (`?:`) vereinfacht werden. Dies entspricht dem Ternary-If, wobei die Bedingung und der True-Teil identisch sind.

```
null ?: 5        // 5         -->  null ? null  :      5
false ?: "test"  // test      --> false ? false : "test"
4 ?: 8           // 4         -->     4 ?     4 :      8
```

Switch

Das switch Statement wurde auch erweitert.

```
def testSwitch(val) {
  switch (val) {
    case ~/^Switch.*Groovy$/: return 'Pattern match'
    case BigInteger: return 'Class isInstance'
    case 60..90: return 'Range contains'
    case [21, 'test', 9.12]: return 'List contains'
    case 42.056: return 'Object equals'
    case { it instanceof Integer && it < 50 }:
      return 'Closure boolean'
    default: return 'Default'
  }
}

println testSwitch("Switch to Groovy")
println testSwitch(42G)
```

```
println testSwitch(70)
println testSwitch('test')
println testSwitch(42.056)
println testSwitch(20)
println testSwitch('default')
```

JSON

Mit JSON kann sehr einfach gearbeitet werden.

JSON lesen

```
def slurper = new groovy.json.JsonSlurper()
def json = slurper.parseText("""
{
  "name": "Mustermann",
  "ids": [5, 10, 11]
}
""")
println(json.name)   // Mustermann
println(json.ids[1]) // 10
```

> In neueren Groovy-Versionen liefert der `JsonSlurper` eine `LazyMap` zurück, die nicht serialisiert werden kann (falls etwas in der Session gespeichert werden soll). Die Variante mit `HashMap` ist noch als `JsonSlurperClassic` verfübar.

Ein JSON-String ist auch schnell aus normalen Objekten (Strings, Lists, Maps, ...) erzeugt:

JSON schreiben

```
def builder = new groovy.json.JsonBuilder()
def obj = [
  name: "Mustermann",
  ids: [5, 10, 11]
]
builder(obj)
println(builder.toPrettyString()) // JSON von oben
```

Als Abkürzung kann die Hilfsklasse `JsonOutput` verwendet werden.

JsonOutput

```
import groovy.json.JsonOutput

def obj = [
  name: "Mustermann",
  ids: [5, 10, 11]
]
def jsonStr = JsonOutput.toJson(obj)
println(jsonStr) // alles auf einer Zeile, spart Bytes
println(JsonOutput.prettyPrint(jsonStr)) // lesbar
```

XML

Für XML-Input gibt es 2 Verarbeitungsmöglichkeiten: `XMLParser` und `XMLSlurper` (siehe dazu auch die Groovy Doku).

`XmlParser` parst das komplette XML und baut eine Struktur im RAM auf, was es erlaubt Teile der XML oft und dabei schnell abzufragen (falls dem XML-Dokument Nodes aktualisiert oder hinzugefügt werden sollen, wird auch `XmlParser` empfohlen):

XML lesen (mit XmlParser)

```
def parser = new XmlParser()
def root = parser.parseText("""
<person>
  <access read='true' write='false' />
  <data>
    <name>Mustermann</name>
  </data>
  <data>
    <name>Musterfrau</name>
  </data>
</person>
""")
println(root.access.@write[0])     // false
println(root.data[1].name.text())  // Musterfrau
```

`XmlSlurper` hingegen wertet das XML nur partiell und on-demand aus - somit wird RAM gespart, häufige Zugriffe hingegen dauern ggf. länger (dafür können Abfragen aber etwas kürzer

geschrieben werden):

XML lesen (mit XmlSlurper)

```
def slurper = new XmlSlurper()
def root = slurper.parseText("""
<person>
  <access read='true' write='false' />
  <data>
    <name>Mustermann</name>
  </data>
  <data>
    <name>Musterfrau</name>
  </data>
</person>
""")
println(root.access.@write)     // false
println(root.data[1].name)      // Musterfrau
```

Um XML-Dokumente programmatisch zusammenzubauen, bietet such `MarkupBuilder` an:

XML schreiben

```
def writer = new StringWriter()
def builder = new groovy.xml.MarkupBuilder(writer)

builder.person {
  access(read: true, write: false)
  data {
    name("Mustermann")
  }
  data {
    name("Musterfrau")
  }
}

println(writer.toString()) // XML von oben
```

Um schnell einzelne Teile einer XML zu ändern und wieder in einen String zu überführen, kann der `XmlParser` und die Hilfsklasse `XmlUtil` verwendet werden. Hier ein Beispiel, um Werte zum Validation-Feld hinzuzufügen:

XML parsen, bearbeiten, zurückschreiben

```
import groovy.xml.XmlUtil

// XML parsen
def rootNode = new XmlParser().parseText("""
<result size="1">
  <field name="Kontakt_Email" resultcode="1" bin="00000001" hex="0x01" />
</result>
""")

// Attribut ändern
rootNode.@size++

// Node hinzufügen (Parameter sind Name der Node sowie Attribute als Map)
rootNode.appendNode("field", [
  "name": "Kontakt_Name",
  "resultcode": "1",
  "bin": "00000001",
  "hex":"0x01"
])

// wieder XML-String erzeugen
println(XmlUtil.serialize(rootNode))
/* Ergebnis:
<?xml version="1.0" encoding="UTF-8"?><result size="2">
  <field name="Kontakt_Email" resultcode="1" bin="00000001" hex="0x01"/>
  <field name="Kontakt_Name" resultcode="1" bin="00000001" hex="0x01"/>
</result>
 */
```

SQL

Auch mit SQL kann einfacher gearbeitet werden.

```
java.sql.Connection conn = ...
def sql = new groovy.sql.Sql(conn)

def foo = "fruit"
```

```
sql.eachRow("SELECT * FROM food WHERE type=${foo}") {  // Parameter werden automatisch SQL-
Encoded
  println("I like ${it.name}")                         // gibt Spalte "name" aus
}
```

> Um ein automatisches SQL-Encoden zu gewährleisten, darf der Query-String nicht aus mehreren Strings zusammengebaut werden, sondern darf nur ein String mit Platzhaltern sein.

Überladene Methoden und Typen

Da Groovy Methoden dynamisch aufruft, zur Laufzeit aber nicht mehr die genauen Typen aus dem Quellcode kennt (wegen Type Erasure etc.), kann es vorkommen, dass mehrere Methoden gefunden werden, die zum Aufruf passen.

Das ist meistens der Fall, wenn `null` übergeben wird, hier weiß Groovy nur, dass es ein `NullObject` ist, kennt aber nicht den genauen Typ:

```
String str = null
println(str.getClass().simpleName) // NullObject
```

Beispielsweise gibt es diese zwei Methoden:

```
void setDate(String str) { ... }
void setDate(Date date) { ... }
```

Erfolgt nun in Groovy der Aufruf folgendermaßen:

```
// null als Literal
obj.setDate(null)
obj.date = null

// oder als Variable
String str = null
obj.date = str
```

Dann weiß Groovy nicht, welche Methode es genau aufrufen soll, das sowohl `String` als auch `Date` zu `NullObject` passen (`null` bzw. `NullObject` kann in beides umgewandelt werden).

Frühere Groovy-Versionen haben einfach irgendeine Methode genommen, neue Versionen werfen eine Exception. Da dies nicht rückwärts-kompatibel ist, bringen CMS-Versionen, welche neuere

Groovy-Versionen enthalten, stattdessen einen Fehler im Log, funktionieren aber weiterhin mit altem Code (es wird irgendeine Methode genommen).

Falls so ein Szenario auftritt, kann man Groovy aber helfen, in dem man beim Aufruf den Typ explizit als Cast hinzufügt (es muss hier die Methode aufgerufen werden, beim Property-Setter geht die Typ-Info wieder verloren):

```
obj.setDate(str as String) // OK
obj.setDate((String)str)   // OK

// obj.date = str as String // geht nicht
// obj.date = (String)str   // geht nicht
```