Orientation in Objects

Groovy Closures

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTOR

Falk Sippach
Orientation in Objects GmbH

Veröffentlicht am: 31.5.2010

EINLEITUNG

Mit der wachsenden Popularität der alternativen Sprachen für die Java Plattform sind Closures wieder verstärkt in den Fokus gerückt. Dabei sind sie keine Erfindung der letzten Jahre, sondern bereits in den 1960er Jahren in dem Lisp-Dialekt Scheme in Erscheinung getreten. Ursprünglich stammt das Konzept aus der funktionalen Programmierung, findet sich aber mittlerweile auch in vielen anderen Sprachen wieder. Typische Vertreter neben Lisp sind u.a. Haskell oder Erlang, auch Smalltalk, (J)Ruby, Scala, Javascript und Clojure. Und auch in Java sollen nach langem Hin und Her in Kürze Closures Einzug halten. Dieser Artikel soll die Funktionsweise von Closures und verschiedene Anwendungsfälle am Beispiel von Groovy näher betrachten.

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

EINFÜHRUNG

WAS IST EIGENTLICH EINE CLOSURE GENAU?

Auf den ersten Blick erinnern Closures stark an die anonymen inneren Klassen von Java. Sie sind allerdings einfacher zu nutzen und zudem viel flexibler einsetzbar. Als Codeblöcke definiert, kapseln sie Verhalten und machen Funktionen somit zu sogenannten First Class Citizens. Das heißt, man kann sie referenzieren (Stichwort Function Pointer) und als Parameter bzw. Rückgabewerte in anderen Funktionen verwenden. Methoden wie man sie bisher aus Java kennt, können hingegen nur im Kontext einer Klasse existieren.

Laut Definition werden sie als Funktionsabschluß bezeichnet. Das bedeutet, es handelt sich um Funktionen, die den zu ihrem Erzeugungszeitpunkt existierenden Kontext konservieren (in sich einschließen). Wird eine Closure ausgeführt, kann sie sowohl auf die ihr übergebenen Parameter als auch auf alle Instanzvariablen und -methoden sowie lokale Variablen zugreifen, die zum Erzeugungszeitpunkt existent waren. Interessanterweise sind aber gerade lokale Variablen zum Ausführungszeitpunkt der Closure eigentlich längst aus dem Speicher entfernt. Closures können natürlich beliebig oft aufgerufen und damit wiederverwendet werden. Dadurch bieten sie vor allem die Möglichkeit, kürzeren, prägnanteren und damit lesbareren Code zu schreiben.

JAVA UND CLOSURES

James Gosling über Closures in Java

Closures were left out of Java initially more because of time pressures than anything else. In the early days of Java the lack of closures was pretty painful, and so inner classes were born: an uncomfortable compromise that attempted to avoid a number of hard issues. But as is normal in so many design issues, the simplifications didn't really solve any problems, they just moved them.

Seit Version 1.1 besitzt Java also in Form der (anonymen) inneren Klassen eine Art einfachen Closure-Support. Sehr oft werden dabei Klassen oder Interfaces mit einer abstrakten Methode (SAM - Single Abstract Method) überschrieben. Solche Typen sind zum Beispiel Runnable, Callable, Comparator, ActionListener und TimerTask. Einsatz finden sie einerseits im Collection Framework (Comparator) oder für den Aufruf konkurrierender Programmabläufe (Multithreading). Sie haben aber im Vergleich zu Closures zwei entscheidende Nachteile. Einerseits ist da die umständliche und sehr wortreiche Syntax, zum anderen sind anonyme innere Klassen sehr eingeschränkt, was die Zugriffsmöglichkeiten auf ihren Aufrufkontext betrifft.

Für das JDK 7 war die Einführung von Closures lange Zeit Gesprächsthema und es gab drei ernsthafte Vorschläge für die Umsetzung. Letztendlich wurde das Thema aber sehr früh wieder aus der Agenda gestrichen. Umso überraschender kam Ende 2009 die Ankündigung, daß Closures im JDK 7 nun doch umgesetzt werden sollen (siehe [6]). Als Grund nannte Projektleiter Mark Reinhold die Vermeidung von Boilerplate-Code beim Schreiben von skalierbaren parallelisierten Programmen. Durch diese Ankündigung verschob sich zwar die Fertigstellung des JDK 7 um weitere Monate, aber seit Juni 2010 gibt es mit dem Milestone 8 eine fertige Implementierung, die sich allerdings nochmal erheblich von den drei ursprünglichen Vorschlägen unterscheidet. Man darf also gespannt sein, wenn hoffentlich ab Herbst 2010 das JDK 7 final geht. Um bis dahin mit den Konzepten vertraut zu sein, sollen im Folgenden die Grundlagen anhand der Closures in Groovy erläutert werden.

CLOSURES IN GROOVY

ERZEUGUNG VON CLOSURES

Closures werden in Groovy durch die Definition eines Codeblocks erzeugt. Dabei handelt es sich um eine oder mehrere Anweisungen, die durch geschweifte Klammern umschlossen sind. Im Gegensatz zu Codeblöcken in Java (statische, Instanz-, synchronized-Blöcke usw.) werden die Anweisungen in den Closures in Groovy allerdings nicht sofort ausgeführt, sondern erst zu einem späteren Zeitpunkt, wenn sie explizit aufgerufen werden. Ähnlich wie normale Funktionen dürfen Closures Argumente enthalten, die direkt hinter der öffnenden geschweiften Klammer beginnen und durch `->` von den eigentlichen Anweisungen abgetrennt sind. Die Argumente können typisiert sein (z. B. `int x`) oder nicht (z. B. `x` oder auch `def x`). Natürlich können Closures auch einen Wert zurückgeben, wobei wie in Groovy üblich, das Schlüsselwort `return` entfallen darf. Dann wird automatisch die letzte Anweisung zurückgeliefert.

```
{[argumente ->] anweisungen }
```

Beispiel 1: Allgemeine Syntax einer Closure Definiton.

Nachfolgend sollen einige Beispiele die verschiedenen Varianten von gültigen Closure-Definitionen zeigen.

```
{println 'Hello World'}
```

Beispiel 2: Einfachste Form einer Closure ohne expliziten Parameter und Rückgabewert.

Diese Closure enthält auf den ersten Blick kein Argument und gibt auch keinen sinnvollen Wert zurück. Wie schon angesprochen, es handelt sich hier nur um die Erzeugung. Der Inhalt (die Anweisungen innerhalb der geschweiften Klammern) wird ohne expliziten Aufruf nie ausgeführt. In diesem ersten Beispiel würde bei der Ausführung der Closure einfach "Hello World" auf der Konsole ausgegeben. Im Gegensatz dazu enthält die nächste Definition sowohl zwei Parameter (`x` und `y`) , als auch einen Rückgabewert. Bei Ausführung werden die beiden Argumente `x` und `y` auf der Konsole ausgegeben und anschließend das Produkt daraus zurückgegeben.

```
{int x, int y ->
 println "x=$x, y=$y"
 return x*y
}
```

Beispiel 3: Closure mit zwei Parametern und explizitem Rückgabewert.

Selbst wenn wie im ersten Beispiel gar kein Closure-Argument deklariert wurde, gibt es trotzdem immer den impliziten Parameter `it`. Um diesen impliziten Parameter zu unterdrücken, kann man auch explizit kein Aufrufargument deklarieren. Dazu muß man den Pfeil ohne vorherige Parameter anwenden, wie in der zweiten Zeile des folgenden Beispiels zu sehen ist.

```
// impliziter Parameter it
{it * 2}
// explizit kein Closure-Parameter
{-> println 'foo'}
```

Beispiel 4: Closure mit impliziten Parameter it und impliziter Rückgabe.

Auf einer Closure-Referenz kann man neben der eigentlichen Ausführung noch weitere nützliche Funktionen aufrufen. Die folgende Tabelle gibt eine Übersicht, einige werden in späteren Kapiteln noch ausführlicher besprochen.

Methode	Beschreibung
`getParameterTypes()`	Informationen über die akzeptierten Parameter, um Closures dynamisch aufrufen zu können.
`getMaximumNumber\ OfParameters()`	Liefert die maximale Anzahl an Parametern zurück, die die Closure akzeptiert.
`isCase()`	Diese Methode ruft die Closure auf und wertet das Ergebnis als `boolean` aus. Verwendet wird sie in `switch` und `grep()`.
`curry()`	Erzeugung einer neuen Closure, wobei Teile der Parameter der Ausgangsclosure mit Werten vorbelegt werden.
`asWritable()`	Liefert neue Closure zurück, die gleichzeitig das Interface `Writable` implementiert. Dadurch wird die Methode `writeTo(Writer)` angeboten, mit der das Ergebnis der Closure effizient in einen `Writer` geschrieben werden kann.

Tabelle 1: Methoden des Typs Closure

CLOSURES AUSFÜHREN

Um eine Closure wiederzuverwenden, kann man sie einer Variablen zuweisen. Diese Variable wird dann analog zu einer Methode aufgerufen, wobei die Parameter in runden Klammern angehängt werden. Alternativ besitzt jede Closure mit der Methode `call` eine Langform. Außerdem kann man Closures theoretisch auch direkt definieren und gleich ausführen, wenn dem Codeblock die runden Klammern und ggf. die Parameter angehängt werden. Im folgenden Beispiel sind die verschiedenen Varianten abgebildet.

```
def c = {println 'Hello World'}
c()
c.call()
{println 'Hello World'}()
```

Beispiel 5: Closures ausführen

Mit der Erzeugung einer Closure erhalten wir also eine Referenz auf eine sonst namenlose Folge von Anweisungen. Diese Referenz kann aber nicht nur einer Variablen zugewiesen werden, um sie später auszuführen. Vielmehr wird man sie auch als Parameter in Methodenaufrufen (oder sogar Closure-Aufrufen) verwenden. Im folgenden Beispiel wird eine Closure erzeugt, die einen übergebenen Parameter quadriert. Diese Closure wird beim Aufruf der calculate()-Methode als Operator wiederverwendet. Weiterhin wird calculate() ein zweites Mal aufgerufen, diesmal mit einer adhoc definierten Closure, die die Kubikzahl für den übergebenen Wert berechnet.

```
def square = {it * it}
println square(4)
void calculate(def x, def operator) {
 println operator(x)
}
calculate(2, square)
calculate(2, {value -> value * value * value}
```

Beispiel 6: Zuweisen zu Variable bzw. Übergabe als Methodenparameter

ANWENDUNGSFÄLLE

Closures erlauben komplexe Programmieraufgaben sehr elegant und lesbar zu lösen. Dadurch ergeben sich jede Menge Anwendungsgebiete. In erster Linie sollte man sie als ein Stück Code sehen, welches andere Anweisungen erweitert und vor allem bereichert. Weil sie Verhalten kapseln und wiederverwendbar machen, vermeiden Closures jede Menge Codeduplikate. Immer wenn man in Java mit Schleifen arbeiten müsste, kann man den Schleifeninhalt in Groovy durch eine Closure abbilden. Ein anderes Beispiel dafür wäre die Definition eines Prädikats bzw. einer Bedingung, um aus einer Menge von Elementen einzelne zu selektieren. Das Groovy Development Kit (GDK) erweitert das JDK für viele dieser Anwendungsfälle um nützliche Methoden, die sehr oft Closures als Parameter enthalten. Durch Groovys flexible Syntax können beim Aufruf von Methoden Closures aus den Aufruf-Klammern herausgezogen werden, wenn es sich um das letzte Argument handelt. Im nachfolgenden Beispiel kann man dieses Verhalten beobachten. Die each()-Methode kann übrigens auf jedem Objekt aufgerufen werden, am sinnvollsten anwendbar ist sie natürlich auf Containern (Collections oder Strings). Die als Parameter übergebene Closure wird dabei nacheinander mit jedem Element des Containers aufgerufen. Falls man zusätzlich den Schleifen-Index benötigt, kann man stattdessen mit der eachWithIndex()-Methode arbeiten.

```
[1, 2, 3].each({print it})
['a', 4, new Date()].each {
   println it
}
['a', 4, new Date()].eachWithIndex {value, index ->
   println "$index: $value"
}
println ([3, 4, 5].findAll {it % 2 == 1})
println (1..10).collect {it * it}
```

Beispiel 7: Closures in GDK-Methoden

Neben der reinen Iterator-Funktion `each()` gibt es eine Vielzahl weiterer Möglichkeiten. So wendet `collect()` die Closure ebenfalls auf jedes Element an, gibt die Ergebnisse aber dann in einer neuen Collection zurück. Mit `findAll()` wird für jedes Element ein Prädikat (Closure mit booleschen Rückgabewert) ausgewertet. Dabei wird bei `true` das jeweilige Element der Result-Collection hinzugefügt. Weitere nützliche GDK-Methoden sind in der folgenden Tabelle aufgelistet. Eine ausführliche Dokumentation findet sich online unter [5].

Methode	Beschreibung
`each()`, `eachWithIndex()`	Iteration und Aufruf der übergebenen Closure für jedes Element (mit und ohne Index)
`collect()`	Ähnlich wie `each()`, allerdings Rückgabe der transformierten Elemente.
`find()`, `findAll()`, `findIndexOf()`, `find...`	Suche von Elementen anhand eines übergebenen Musters in Form einer Closure.
`any()`, `every()`	Überprüfung, ob irgendein oder jedes Element einem bestimmten, mit Closure übergebenen Muster übereinstimmt.
`use()`	Simulation eines Codeblocks mit besonderen Eigenschaften durch Anwendung von Metaprogrammierung-Magie auf die Anweisungen in der übergebenen Closure.
`with()`	Aufbau einer einfachen DSL bzw. Fluent-API durch Delegation aller Aufrufe innerhalb der Closure an das with-Objekt.

Tabelle 2: GDK-Methoden mit Closure-Argumenten

Mit Closures kann man relativ einfach beliebige Ein-Methoden-Interfaces (SAM) implementieren. Man findet diese SAM-Interfaces unter anderem im Collection-Framework, beim Event-Handling von Oberflächenbibliotheken und bei der nebenläufigen Programmierung mit Threads. In Java müßte man dafür mit anonymen inneren Klassen arbeiten. Wie kompakt die Implementierung des Strategie- bzw. Befehl-Musters in Groovy aussehen kann, zeigen die nächsten Beispiele. Statt die konkrete Strategie bzw. den konkreten Befehl in einer eigenen Klasse zu implementieren, muß man die Anweisungen nur in einer Closure definieren. Groovy versucht übrigens die Closures automatisch an die erwartete Schnittstelle anzupassen. Alternativ hat man auch immer die Möglichkeit mit dem Schlüsselwort `as` die Closure explizit in einen Typ zu zwingen.

```
def descComparator = { x, y -> y <=> x } as Comparator
def liste = [5, 2, 8]
println liste.sort(descComparator) // -> [8, 5, 2]
// button.setActionPerformed(new ActionListener() {...});
button.actionPerformed = { label.text = 'gedrückt' }
// Erzeugung eines neuen Threads
Thread.start {('A'..'Z').each { sleep 100; println it }}
Thread.start {(1..26).each { sleep 110; println it }} as
Runnable
```

Beispiel 8: Implementieren beliebiger (Ein-Methoden-) Interfaces

Für die Implementierung der SAM-Interfaces bieten sich Closures geradezu an. Aber auch die Simulation von Interfaces mit mehr als einer abstrakten Methode läßt sich mit Closures relativ einfach umsetzen. Und zwar wird hierfür eine Map definiert. Dabei werden den Schlüsseln Closures zugeordnet, welche die zu überschreibenden Methoden simulieren. Diese Map wird dann mit `as` als das zu implementierende Interface gecastet. Das folgende Beispiel zeigt die Erzeugung eines MouseListeners.

```
def x = [mouseClicked:{println "click"},
   mousePressed:{println pressed}, ...] as MouseListener
```

Beispiel 9: Implementierung von Interfaces mit mehr als einer Methode

Ein anderes Anwendungsbeispiel ist das Erzeugen von Dummy-Objekten mittels einer Map oder eines Expandos. Durch das Hinzufügen von Closures werden die Methoden des Dummys simuliert.

```
m = [ f: { println 'f called' } ]
m.f()
m = new Expando( f: { println 'f called' } )
m.f()
```

Beispiel 10: Zuweisen zu Variable bzw. Übergabe als Methodenparameter und Aufruf einer Closure

Die Liste der Anwendungsszenarios von Closures in Groovy läßt sich noch beliebig fortsetzen. So kann man das Method Around Pattern mittels Closures implementieren und so sehr einfach fachliche von nicht-fachlicher Logik, dem sogenannten Boilerplate Code (z. B. Exception-Handling, Aufräumen von Resourcen usw.), trennen. Durch die weiter oben in der Tabelle kurz vorgestellten Methoden with() und use() kann der Kontext von Anweisungen, die sich innerhalb eines Codeblocks (einer Closure) befinden, umgebogen werden. Somit können diese Anweisungen beispielsweise um neue Funktionen erweitert werden (Teil von Groovys Metaprogrammierung). Closures in Groovy ermöglichen zu guter Letzt eine äußerst effektive Art, das Builder Pattern umzusetzen, um zum Beispiel eigene Domain Specific Languages (DSL) zu entwerfen.

ERZEUGUNGSKONTEXT UND GÜLTIGKEITSBEREICH

Closures konservieren ihren bei der Erzeugung umgebenden Zustand. Beim Aufrufen hat die Closure somit Zugriff auf alle bei der Erzeugung erreichbaren Member und lokalen Variablen, obwohl sie scheinbar außerhalb des Sichtbarkeitsbereichs liegen oder im Falle von lokalen Variablen gar nicht mehr im Speicher existieren. Das folgende Beispiel soll diese Mechanismen verdeutlichen. Es gibt eine `TestKlasse`, die ein Property (privates Attribut mit getter/setter) namens `i` und eine öffentliche Instanz-Methode `tuWas(s)` deklariert. In einer zweiten öffentlichen Methode `erzeugeClosure()` wird wiederum eine lokale Variable `lokal` deklariert und anschließend eine Closure erzeugt und sofort zurückgegeben. Ganz wichtig ist hier wieder, daß die Closure an dieser Stelle nur definiert und nicht aufgerufen wird. In der Closure wird sowohl auf die lokale Variable der erzeugenden Methode, als auch auf die Member-Variable und die Instanz-Methode der `TestKlasse` zugegriffen.

```
class TestKlasse {
  def i = 10
  def tuWas(p1, p2) {
    "p1=$p1, p2=$p2, i=$i"
  }
  def erzeugeClosure() {
    def lokal = 1
    return {param ->
      lokal++;
      i = i + lokal*10;
      tuWas(lokal, param)
    }
  }
}
def tk = new TestKlasse()
def c = tk.erzeugeClosure()
println c(13) // => "p1=2, p2=13, i=30"
println c(17) // => "p1=3, p2=17, i=60"
println tk.erzeugeClosure()(33) // => "p1=2, p2=33, i=80"
```

Beispiel 11: Zugriff auf Erzeugungskontext

Außerhalb der Klassendefinition wird eine neue Instanz der `TestKlasse` erzeugt und darauf die Methode `erzeugeClosure()` aufgerufen. Die zurückgegebene Closure wird anschließend zweimal mit unterschiedlichen Parametern ausgeführt und das Ergebnis aufgelistet. Man sieht sehr schön, daß beim Aufruf der Closure sowohl Zugriff auf `i` und `tuWas(s)` besteht und außerdem auch die lokale Variable `lokal` les- und schreibbar ist, obwohl sie nach der Rückgabe aus der Methode `erzeugeClosure()` eigentlich längst gelöscht sein sollte. Dieser Zugriff funktioniert nur deshalb, weil der Groovy-Compiler für alle zur Erzeugungszeit bekannten lokalen Variablen in der Closure jeweils eine Instanzvariable mit der Kopie des Wertes anlegt. Bei Erzeugung einer weiteren Closure wird quasi mit einer frischen lokalen Variable gearbeitet (siehe letzte Zeile des Beispiels).

Der Zugriff auf die Member von Testklasse funktioniert, weil sich jede Closure eine Referenz auf das sie bei der Erzeugung umschließende Objekt merkt. Zur Ausführungszeit wird beim Auflösen von Anweisungen zuerst innerhalb der Closure (lokale Variable, Instanzvariable) und dann in eben diesen Referenzen (`owner` und `delegate`) gesucht. Die Auflösungsreihenfolge kann auch gesteuert werden, wie im nächsten Beispiel zu sehen ist. Die this-Referenz (zeigt auf die Closure selbst) und der Owner können nicht verändert, das Delegate kann allerdings manuell umgebogen werden. Dadurch ergeben sich Möglichkeiten, daß eine Closure nicht nur in ihrem Objekt (this), in ihrem Erzeugungskontext (owner) sondern auch in einem beliebigen, frei wählbaren Objekt der Anwendung nach unbekannten Variablen und Methoden sucht.

```
class TestKlasse {
  def i = 10
  def erzeugeClosure() {
    return { ->
      i++
      "i=$i, j=$j"
    }
  }
}
i = 20
j = 50
def c = new TestKlasse().erzeugeClosure()
c.setDelegate(this)
println c()
println c()
c.setResolveStrategy(Closure.DELEGATE_FIRST)
println c()
```

Beispiel 12: Zugriff auf Erzeugungskontext

BLICK HINTER DIE KULISSEN

Um die Eigenheiten von Closures zu verstehen, lohnt sich ein Blick auf die Ausgabe des Groovy-Compilers. Im folgenden Beispiel wird in einem Groovy-Skript (mittels der GroovyConsole) eine Closure erzeugt, die bei ihrem Aufruf das Quadrat des übergebenen Parameters zurückgibt.

```
def c = {it*it}
println this.class.name // => ConsoleScript
println c.class.name // => ConsoleScript$_run_closure1
println c(4) // => 16
```

Beispiel 13: Einfache Closure in Groovy

Der Groovy-Compiler übersetzt dieses Skript schematisch gesehen in den folgenden Java-Code. Kompilierter Groovy-Code unterscheidet sich übrigens nicht von Bytecode, der aus Java erzeugt wurde. Für die Closure wird eine eigene (innere) Klasse erzeugt (`_run_closure1`). Der Inhalt wird dabei in die Methode `doCall()` generiert. Hätte es lokale Variablen gegeben, wären jeweils Instanzvariablen mit einer Kopie des Wertes erzeugt worden. Da die vier Anweisungen ein Skript darstellen, erzeugt der Groovy-Compiler weiterhin die Klasse `ConsoleScript`, wobei der Skript-Code in die `run()` Methode generiert wird. Dabei kann man sehen, daß beim Erzeugen einer neuen Closure-Instanz der Kontext mit hineingegeben wird. Überprüfen kann man diese Funktionalität über die Ausgabe des Klassennames der this-Referenz (Skriptklasse) und der Closure-Referenz. Der Aufruf der Closure (`c(4)`) triggert über `call()` die generierte Methode `doCall()` und stößt damit das Quadrieren des Parameters 4 an.

```
public class ConsoleScript extends Script {
 public class _run_closure1 extends Closure {
 public _run_closure1(Object owner) {
  super(owner);
 }
 public Object doCall(Object it) {
  return it * it;
 }
 }
 public void run() {
 Closure c = new ConsoleScript._run_closure1(this);
 // => ConsoleScript
 System.out.println(this.getClass().getName());
 // => ConsoleScript$_run_closure1
 System.out.println(c.getClass().getName());
 System.out.println(c.call(4)); // => 16
 }
 public static void main(String[] args) {
 new ConsoleScript().run();
 }
}
```

Beispiel 14: Java-Äquivalent einer einfachen Closure in einem Groovy-Skript

"FUNKTIONALERES" PROGRAMMIEREN DANK CLOSURES

Funktionales Programmieren ist ein Programmierstil, der Probleme durch die Anwendung von Funktionen löst. Eine Grundvoraussetzung ist, daß diese Funktionen keine Seiteneffekte, wie das Ändern eines globalen Zustands, Ein- und Ausgaben bzw. Datenbank-Updates, haben. Die Funktionen müssen immer das gleiche Ergebnis zurückgeben, wenn sie mit den gleichen Eingabeparameter aufgerufen werden. Der Zustand in der funktionalen Progammierung wird in den Funktionsparametern auf dem Stack anstatt auf dem Heap gehalten. Dadurch sind solche Funktionen einfacher zu verstehen, zu testen und zu debuggen. Imperative Programmierung (z. B. mit Java) konzentriert sich im Gegensatz dazu auf Zustandsänderungen und die Ausführung von fortlaufenden Befehlen. Typische Vertreter der funktionalen Programmiersprachen sind Lisp, Erlang, Haskell, Scala und Clojure. Ihre Merkmale sind:

- Closure Support
- Funktionen höherer Ordnung (Funktionen, die andere Funktionen als Argumente erhalten oder Funktionen zurückgeben)
- Verwendung von Rekursion zur Steuerung des Kontrollflusses
- Keine (wenige) Seiten-Effekte, keine referenzielle Transparenz

Natürlich ist Groovy keine funktionale Programmiersprache, man kann damit trotzdem in funktionaler Weise Software entwickeln. Im Mittelpunkt der funktionalen Programmierung steht gegenüber der Objektorientierung die Zerlegung von Algorithmen in einzelne Funktionen ähnlich wie man es aus der Mathematik kennt. Funktionen werden dann mit anderen Funktionen aufgerufen. Und das funktioniert nur, wenn sie First Class Citizens sind. Und genau das bieten Closures in Groovy.

Theoretisch könnte man jede Methode in Groovy auch als Closure erzeugen. Durch die Kurzform (`c()` anstatt `c.call()`) merkt man beim Aufruf von Closures gegenüber dem Aufruf von Methoden keinen Unterschied. Im folgenden Beispiel wird zuerst eine Funktion `facFunction` erzeugt, welche die Fakultät für einen bestimmten Wert berechnet. Die gleiche Funktionalität kann man erreichen, wenn man eine Closure (`facClosure`) definiert, die ebenfalls ein Aufrufargument besitzt. Der Aufruf von `facFunction(5)` und `facClosure(5)` ist vom Ergebnis her identisch. Es ist übrigens auch möglich, Methoden (statische und Instanz) in Closures zu konvertieren. Durch den & Operator kann man eine Referenz (ähnlich eines Function Pointers) auf `facFunction` erzeugen. Die daraus entstandene Closure `facClosure2` verhält sich wie `facClosure` und kann genauso als Parameter herumgereicht und aufgerufen werden.

```
// Methode/Funktion facFunction
def facFunction(value) {
   if (value < 2) {
      1
   } else {
      value * facFunction(value - 1)
   }
}
println facFunction(5) // => 120
// Closure facClosure
def facClosure = {value ->
   if (value < 2) {
      1
   } else {
      value * call(value - 1)
   }
}
println facClosure(5) // => 120
// Referenz auf Methode facFunction()
def facClosure2 = this.&facFunction
println facClosure2(4) // => 24
```

Beispiel 15: Funktionen vs. Closures

Ein typischer Anwendungsfall der funktionalen Programmierung ist das Currying. Damit sind allerdings keine scharf gewürzten Methoden gemeint. Der Name stammt vielmehr von Haskell Curry, einem Mathematiker, der sich viel mit kombinatorischer Logik beschäftigt hat. Seine Arbeiten bauen auf dem Wissen eines russischen Wissenschaftlers - Moses Schönfinkel - auf, deswegen spricht man gelegentlich auch vom Schönfinkeln. Durch Currying kann man einen oder mehrere Parameter einer Closure fixieren und erhält eine neue Closure mit entsprechend weniger Aufrufargumenten. Im folgenden Beispiel wird eine Closure `log` erzeugt, die aus dem übergebenen Log-Level und der Log-Nachricht einen Log-String zusammenbaut und zurückgibt. Diese Closure kann man ganz normal wie einen Funktionsaufruf verwenden. Als Alternativen werden durch `curry()` neue Closures erzeugt, wobei das Log-Level jeweils auf einen festen Wert eingestellt wird. Dadurch entstehen die Closures `warn` und `info`, die nur noch mit einer Nachricht aufgerufen werden müssen und das entsprechende Log-Level automatisch ausgeben.

```
def log = {level, msg -> "${level.toUpperCase()}: $msg"}
println log('debug', 'Debugmeldung') // => DEBUG:
Debugmeldung
println log('warn', 'Warnhinweis') // => WARN: Warnhinweis
def warn = log.curry('warn')
def info = log.curry('info')
println info('Infomeldung') // => INFO: Infomeldung
println warn('Warnhinweis') // => WARN: Warnhinweis
```

Beispiel 16: Funktionen vs. Closures

Neben dem Fixieren von einfachen Datentypen kann man mittels Currying auch Funktionen verketten, wie man es bereits von der Mathematik her kennt: `f(g(x))`. Dafür muß eine Funktion als Parameter einer anderen Funktion übergeben werden, was man als Funktion höherer Ordnung bezeichnet. Im folgenden Beispiel verkettet die Closure `composition` eine Funktion, die Werte verdoppelt, mit einer Funktion, die Werte verfünfacht. Heraus kommt eine neue Funktion, die übergebene Werte mit 10 multipliziert. Das Besondere ist hier, daß ein größeres Problem in mehrere Berechnungsfunktionen unterteilt wurde, welche dann beliebig zusammengesteckt und wiederverwendet werden können.

```
def multiply = { x, y -> return x * y }
def double = multiply.curry(2)
def fiveTimes = multiply.curry(5)
def composition = { f, g, x -> return f(g(x)) }
def tenTimes = composition.curry(double, fiveTimes)
println tenTimes(4) // => 40
```

Beispiel 17: Komposition von Funktionen

ZUSAMMENFASSUNG

Closures sind eines der Highlights in der Syntax von Groovy. Es gibt dutzende Anwendungsfälle und es ist schwer, beim Programmieren in Groovy ohne Closures auszukommen. Allein beim GDK, welches die Java Standard Klassen (JDK) um viele nützliche Methoden erweitert, trifft man auf Schritt und Tritt auf Closures. Beispielhaft sei hier nochmal das Iterieren, Filtern oder Anwenden von Funktionen auf Datencontainern genannt. Wenn man dank Groovys Metaprogrammierungsfähigkeiten eigene DSLs erstellen möchte, muß man sich mit den Eigenschaften von Closures und den verschiedenen Möglichkeiten, unbekannte Werte aufzulösen, beschäftigen. Außerdem lassen sich durch Closures viele Design Patterns sehr elegant und effizient verbauen. Dadurch profitiert vor allem der geschriebene Quellcode, der einerseits kürzer, trotzdem ausdrucksstärker und prägnanter wird und somit die Lesbarkeit deutlich erhöht.

REFERENZEN

- [1] Groovy in Action
  König, Dierk et al.
  (http://www.manning.com/koenig)
- [2] Groovy: Grundlagen und fortgeschrittene Techniken
  Baumann, Joachim
  (http://www.dpunkt.de/buecher/2610.html)
- [3] Programming Groovy
  Subramaniam, Venkat
  (http://www.pragprog.com/titles/vslg/programming-groovy)
- [4] Wikipedia
  (http://de.wikipedia.org/wiki/Closure)
- [5] Groovy Development Kit (GDK)
  (http://groovy.codehaus.org/groovy-jdk/)
- [6] Closures for Java
  (http://blogs.sun.com/mr/entry/closures)