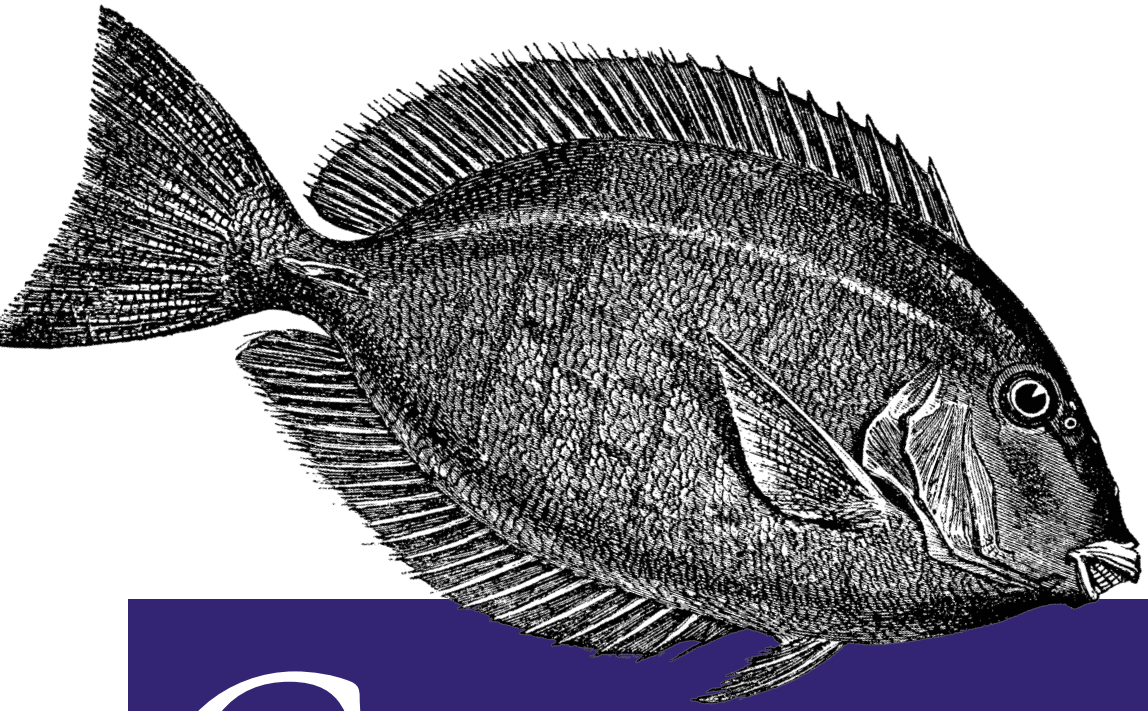
Dynamik im Java-Universum

Deutsche Originalausgabe



Groovy

für Java-Entwickler

O'REILLY®

Jörg Staudemeyer

Groovy für Java-Entwickler

Jörg Staudemeyer

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag
Balthasarstr. 81
50670 Köln
Tel.: 0221/9731600
Fax: 0221/9731608
E-Mail: kommentar@oreilly.de

1. Auflage 2007

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.ddb.de* abrufbar.

Lektorat: Christine Haite, Köln
Fachgutachter: Karsten Loesing, Bamberg, Dirk Gomez & Daniel Lehmann, Bonn, Christof Vollrath, Berlin
Korrektorat: Sibylle Feldmann, Düsseldorf
Satz: G&U Language & Publishing Services GmbH, Flensburg; www.GundU.com
Umschlaggestaltung: Edie Freedman, Boston & Michael Oreal, Köln
Produktion: Andrea Miß, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN 978-3-89721-483-5

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Vorwort

Groovy ist eine neue Open Source-Programmiersprache, die mit dem Ziel erschaffen wurde, produktivitätssteigernde Features moderner Interpreter-Sprachen wie Python, Ruby und Smalltalk nahtlos in die Java-Welt zu integrieren.

Groovy wird häufig als »Skriptsprache« bezeichnet. Das wird ihr nicht ganz gerecht. Denn Groovy kann zwar als Skriptsprache verwendet werden, steht auch in der Tradition von Skriptsprachen und hat typische Eigenschaften von Skriptsprachen – geht aber in vielen Aspekten über Skriptsprachen hinaus. Vielleicht sollte man deshalb besser von einer »dynamischen Sprache« sprechen. Die Sprache selbst lehnt sich stark an Java an, enthält aber eine Reihe wesentlicher Erweiterungen und Besonderheiten, die sie um einiges mächtiger und sehr viel flexibler machen.

1. Skriptfähigkeit. Sie können – aber müssen nicht – Groovy-Programme direkt aus dem Quellprogramm heraus starten, ohne sie explizit zu Klassendateien kompilieren zu müssen.
2. Eine mächtigere Sprache. Groovy umfasst einige Erweiterungen wie Closures, überladbare Operatoren und die Unterstützung von Standardtypen, durch die sie viel mächtiger wird als Java. In Verbindung mit den Klassen der Groovy-Laufzeitbibliothek kann der Programmcode gegenüber Java oft auf ein Bruchteil reduziert werden.
3. Der dynamische Charakter der Sprache Groovy eröffnet eine Fülle neuer Möglichkeiten, die weit über das hinausgehen, was konventionelle statische Programmierung erlaubt.

Diese neue Sprache würde es vielleicht nicht geben, hätten nicht in letzter Zeit eine neue Skriptsprache namens *Ruby* und das darauf basierende Web-Entwicklungsframework Ruby on Rails enormes Aufsehen wegen der Geschwindigkeit erregt, die sie bei der Anwendungsentwicklung erlauben. Groovy ist aber keine Portierung von Ruby auf die

Java-Plattform,[1] sondern ein völlig eigenständiges Konzept. Allerdings übernimmt Groovy einige wichtige Ideen von Ruby (und anderen Sprachen), darunter deren dynamischen Charakter und die Anlehnung an funktionale Programmierung. Entscheidend ist, dass Groovy im Unterschied zu einer bloßen Portierung einer fremden Sprache fest in der Java-Welt verwurzelt ist. Die Sprache ist konsequent als Weiterentwicklung von Java konzipiert, und die Integration mit konventionell programmierten Java-Programmen ist – in beiden Richtungen – völlig problemlos.

Wenn Sie sich also mit Java auskennen und sich mit Groovy beschäftigen möchten, brauchen Sie nicht etwas völlig Neues zu lernen. Sie fangen mit dem an, was Sie kennen: Das schlichte »Hello World« funktioniert ohne jede Änderung auch in Groovy. Binnen kürzester Zeit werden Sie aber feststellen, dass es anders geht, dass sich so manche Konstruktion, die in Java umständlich und undurchsichtig ist, mit Groovy auf wenige Zeilen verkürzen lässt, die das Wesentliche Ihres Programms ausdrücken. Und noch etwas später werden Sie erkennen, dass Ihnen Groovy Möglichkeiten eröffnet, die weit über das hinausgehen, was das klassische Java Ihnen bieten kann. Zu Groovys eigenständigen Konzepten gehören beispielsweise Builder der Features zum Entwerfen von Domain Specific Languages (DSL).

Sie müssen sich also nur ein wenig darauf einlassen.

Erwarten Sie aber nicht, dass Sie Ihr nächstes größeres Projekt komplett in Groovy implementieren werden. Die Sprache ist jung, vieles ist noch in Bewegung, und es gibt so manchen Bug und manche Ungereimtheit, die zu beseitigen noch einige Versionsnummern dauern wird. Darüber hinaus wirft auch der dynamische Charakter der Sprache Fragen nach der Stabilität und der Performance auf, die vor einem Einsatz in Projekten beantwortet werden müssen. Schon jetzt gibt es aber eine Reihe von Anwendungsfällen, für die Groovy sich als Mittel der Wahl darstellt:

- Mit seiner Scripting-Fähigkeit bietet sich Groovy besonders für die Anpassung von Softwaresystemen an lokale oder zeitliche Gegebenheiten an, insbesondere wenn die üblichen Mittel wie Properties- und XML-Dateien nicht mehr ausreichen.
- In Zusammenhang mit Grails ist Groovy eine hochinteressante Möglichkeit für den Aufbau von Webanwendungen oder Web-Frontends, die die bewährten Mittel der Java-Webentwicklung nutzen, aber gleichzeitig viel rascher und flexibler erstellt werden können.
- Als Hilfsmittel für kleine Ad-hoc-Tools, die von Java-Entwicklern aufgrund der vertrauten Syntax und Umgebung mit geringem Aufwand selbst geschrieben werden können, insbesondere wenn sie beliebige Java-Bibliotheken nutzen.
- Implementierung von Build-Prozessen und sonstigen automatisierten Vorgängen, in denen sich die deklarativen Gestaltungsmittel von Apache Ant mit den prozeduralen Sprachmitteln von Groovy kombinieren lassen

1 Eine solche Portierung bietet – mit einer völlig anderen Zielrichtung als Groovy – ein Open Source-Projekt mit dem Namen *JRuby*.

- Testskripten sowie Dummies und Mock-Objekte, die den isolierten Test einzelner Module erleichtern.
- Prototypen von Anwendungen oder Komponenten, mit denen Sie die Machbarkeit von Softwareprojekten nachweisen können und die als Platzhalter für noch in der Entwicklung befindliche Anwendungsteile dienen.
- Schließlich, und das ist nicht zu unterschätzen, ist Groovy eine hervorragende Möglichkeit für Java-Programmierer, sich mit neuen, agilen Programmierkonzepten auseinanderzusetzen, ohne sich erst einmal mühsam in die Spezifika einer unbekannten Plattform wie Python oder Ruby einarbeiten zu müssen.

Mit dem *Java Specification Request* (JSR) 241 ist sie in die Standardisierung durch den *Java Community Process* einbezogen und wird damit quasi die zweite »amtliche« Sprache auf der Java-Plattform neben Java selbst. Groovy versteht sich somit als *eine* Implementierung des JSR-241. Entwickelt wird sie in einem bei Codehaus (*www.codehaus.org*) gehosteten Open Source-Projekt, das von Bob McWhirter und James Strachan gegründet wurde; derzeitiger Project Manager ist Guillaume Laforge. Nach einer Reihe von Überarbeitungen erfolgte die Freigabe der Groovy-Version 1.0 Anfang 2007.

Dieses Buch hat die Groovy-Version 1.1-beta-2 zur Grundlage, die zwar die wichtigsten, aber sicher noch nicht alle Features der Version 1.1 enthält. Diese soll im dritten Quartal 2007 erscheinen, also etwa gleichzeitig mit diesem Buch. Daher ist mit geringfügigen Abweichungen zwischen dem zu rechnen, was Sie hier lesen, und dem, was Sie beim Arbeiten mit Groovy auf dem Bildschirm sehen. Das Projekt Groovy ist ebenso dynamisch wie die Programmiersprache, deswegen müssen Sie ohnehin mit einer flotten Weiterentwicklung rechnen; eine Version 1.2 und weitere sowie auch 2.0 werden nicht allzu lange auf sich warten lassen. Und so kann das Ziel dieses Buchs in erster Linie nur sein, Sie mit den grundlegenden Konzepten und den wichtigsten Möglichkeiten vertraut zu machen, die nicht allzu bald *deprecated* sein werden.

Zu diesem Buch

Groovy für Java-Entwickler ist eine Einführung in die Programmierung mit Groovy für Menschen, die sich mit Java schon etwas auskennen und mehr über diese neue Programmiersprache erfahren möchten. Das Buch soll Ihnen die wesentlichen Konzepte verdeutlichen, die hinter der Sprache und dem Web-Entwicklungsframework stehen, und Ihnen zeigen, welche neuen Möglichkeiten sich damit erschließen. Da Groovy syntaktisch auf Java aufsetzt (viele Java-Quellprogramme können ohne oder mit geringfügigen Änderungen auch mit Groovy genutzt werden) und auch die Java-Standardbibliotheken nutzt, werden wir nur die Unterschiede und die zusätzlichen Möglichkeiten gegenüber Java darstellen, nicht aber das, was man auch mit Java allein bereits machen kann.

Folglich ist dieses Buch keine Einführung in die Programmierung, sondern setzt zumindest Grundkenntnisse in der Entwicklung von Java-Programmen voraus; besser noch ist

ein gewisses Fundament an Erfahrungen mit der Technik, den Konventionen und diversen Möglichkeiten der Softwareentwicklung auf Java-Basis. Wir werden diese Grundlagen nicht immer erläutern können, wohl aber hier und da Hinweise zum Nachlesen geben.

Der Autor hat sich viel mit Groovy beschäftigt und im Laufe der Zeit immer mehr Geschmack an den neuen Möglichkeiten gefunden, die diese Sprache gegenüber der Programmierung mit dem konventionellen Java bietet. Er gehört aber nicht zum Groovy-Projekt und hält auch absichtlich eine gewisse kritische Distanz. Sie erleichtert es, auf Schwächen und problematische Aspekte hinzuweisen – ohne allerdings in kleinkarierte Krittelei zu verfallen, denn was das gemessen an der Aufgabe sehr kleine Team geleistet hat, ist bewundernswert. Dennoch sollen Sie als Anwender von Groovy nicht in Fallen laufen, die Ihnen leicht die Freude an der Arbeit mit diesem insgesamt wunderbaren Werkzeug verleiden könnten.

Das Buch gliedert sich in neun Kapitel und einige Anhänge, deren jeweilige Ziele hier kurz erläutert werden sollen.

Kapitel 1, Erste Schritte
: Als Erstes und als Grundlage für alles Weitere wollen wir Ihnen zeigen, wie Sie Groovy auf Ihrem Rechner installieren und wie Sie Ihr erstes kleines Programm schreiben und auf unterschiedliche Weise zum Laufen bringen können.

Kapitel 2, Die Sprache Groovy
: Dieses Kapitel macht Sie mit allen Dingen vertraut, die Groovy als Sprache von Java unterscheidet. Es zeigt Ihnen, wie Groovy insgesamt »funktioniert«; auf vieles, was hier nur kurz angedeutet werden kann, gehen spätere Kapitel ausführlicher ein.

Kapitel 3, Objekte in Groovy
: Wie in Java, so schreiben Sie auch in Groovy Programme immer in Form von Klassen. Dabei ist jedoch in Groovy manches anders als in Java, und es gibt eine Reihe zusätzlicher Möglichkeiten, die dieses Kapitel ausführlich erläutert.

Kapitel 4, Neue Konzepte
: In Groovy gibt es einige Dinge, die Sie von Java und dem JDK nicht kennen, z.B. Closures, GStrings und Builder. Da sie eine wichtige Grundlage für die Arbeit mit Groovy und das Verständnis der Groovy-Bibliothek sind, stellen wir sie hier ausführlich vor und zeigen, wie man mit ihnen arbeitet.

Kapitel 5, Wie Groovy das JDK erweitert
: Wenn Sie mit Groovy programmieren, können Sie an altbekannten Java-Standardklassen plötzlich Methoden aufrufen, die es dort gar nicht gibt. Wir nennen Sie *vordefinierte Methoden* und erklären hier, welche es gibt und wie sie funktionieren.

Kapitel 6, Highlights der Groovy-Standardbibliothek
: Zu Groovy gehört neben der Programmiersprache eine Bibliothek mit einer Fülle fertiger Klassen, mit denen sich häufige Programmieraufgaben leichter und effizienter erledigen lassen. Dieses Kapitel gibt einen Überblick sowie eine Einführung in die wichtigsten dieser Klassen.

Kapitel 7, Dynamisches Programmieren
: Eine Spezialität von Groovy ist, dass es Objekte zur Laufzeit ganz anders erscheinen lassen kann, als es in seiner Klasse ursprünglich definiert ist. Hier zeigen wir Ihnen die verschiedenen Möglichkeiten, Objekte mit zusätzlichen Methoden und Properties zu versehen, Methoden zu verändern und sogar dem Methodenaufruf an sich eine neue Bedeutung zu geben.

Kapitel 8, Groovy und Java integrieren
: Groovy und Java sind eng verbunden; trotzdem gilt es einiges zu beachten, wenn Sie Groovy-Klassen oder -Skripte in Ihre Java-Programme integrieren wollen. Dazu gehören Aspekte zur statischen und dynamischen Integration sowie Sicherheitsfragen, die bei der Laufzeitintegration von Skripten zu beachten sind.

Kapitel 9, Groovy im Entwicklungsprozess
: Mit Groovy kann man nicht nur Anwendungen entwickeln. Es eignet sich auch hervorragend für den Einsatz als Werkzeug im Entwicklungsprozess – auch von normalen Java-Anwendungen. Wir zeigen Ihnen den Einsatz von Groovy in Zusammenhang mit Ant-Builds und JUnit-Tests.

In den Anhängen finden Sie noch eine Reihe von Informationen zum Nachschlagen. Diese werden Ihnen sowohl beim Lesen des Textteils als auch später zum Nachschlagen hilfreiche Dienste leisten.

- Anhang A enthält eine Zusammenstellung aller vordefinierten Methoden, dies sind jene Methoden, die in einem Groovy-Programm für diverse Typen verfügbar sind, obwohl sie dort gar nicht definiert sind.
- Anhang B dokumentiert einen Teil der Klassen, die in der Groovy-Standardbibliothek enthalten sind und die für die alltägliche Groovy-Programmierung besonders wichtig sind.
- Anhang C listet eine Reihe von Oberflächenkomponenten auf, die vom `SwingBuilder`, einer Klasse zum Aufbau von Swing-Oberflächen in einem quasi-deklarativen Stil, zur Verfügung gestellt werden.
- Anhang D erläutert die zu Groovy gehörenden Tools mit ihren jeweiligen Aufrufoptionen.

Hinweise zur Lektüre

Als Java-Programmierer werden Sie mithilfe dieses Buchs rasch einen Zugang zur Arbeit mit Groovy finden. Lesen Sie auf jeden Fall die Kapitel 1 und 2, die Ihnen zeigen, wie Sie Groovy installieren, wie Sie Groovy-Skripte und -Programme zum Laufen bringen und worin die wesentlichsten sprachlichen (d.h. syntaktischen und semantischen) Unterschiede zwischen den Sprachen Groovy und Java bestehen.

Sinnvoll ist auch, die Kapitel 3 bis 5 anfangs zumindest »diagonal« zu lesen. Sie zeigen wichtige Grundlagen für das Programmieren in Groovy, die aber an manchen Stellen sicher etwas tiefer ins Detail gehen, als es am Anfang notwendig ist. Irgendwann sollten Sie die drei Kapitel allerdings auch etwas eingehender studieren, um sich sicher in der Groovy-Welt bewegen zu können.

Die restlichen Kapitel (6 bis 9) betreffen dann verschiedene Spezialfragen, die eher dann von Interesse sind, wenn Sie ein bestimmtes Problem lösen müssen, für das Groovy Ihnen eine spezielle Lösung bieten kann. Auch hier ist es sicher sinnvoll, die Seiten anfangs einmal zu überfliegen, damit Sie überhaupt wissen, was es so gibt und wo Ihnen Groovy helfen kann. Eingehendes Lesen lohnt sich aber in erster Linie dann, wenn es für Sie konkret wird.

Weitere Informationen

Und auch nach Freigabe der Version 1.1 wird Groovy *work in progess* sein, d.h., bis ein gewisser Reifegrad erreicht ist, wird sich manches ändern und vieles hinzukommen. Dieses Buch kann Ihnen nur einen ersten Einstieg bieten und zeigen, welche Möglichkeiten Ihnen mit Groovy und Grails prinzipiell zur Verfügung stehen. Spätestens wenn Sie tiefer gehende Informationen benötigen, wenn Zweifelsfälle auftreten oder wenn sich Dinge ändern, werden Sie sicher auf weitere Informationen zugreifen wollen. Dazu seien hier schon mal einige mögliche Quellen genannt. Zunächst jene im Internet (alle in Englisch):

- Die Projekt-Website *http://groovy.codehaus.org*. Hier finden Sie unter DOCUMENTATION eine Fülle von Informationen, allerdings – wie häufig in Open Source-Projekten mit knappem Budget – nicht immer lückenlos und nicht immer ganz aktuell. Auch das JavaDoc zu Groovy ist hier zu finden, aber auch darin fehlt so mancher Kommentar oder ist manche Erläuterung etwas knapp.
- Angesichts dessen kann es durchaus hilfreich sein, einen Blick in den Quellcode mancher Groovy-Standardklasse zu werfen. Laden Sie sich die Quellen von den Download-Seiten des Projekts oder betrachten Sie sie online unter *http://groovy.codehaus.org/xref/*.
- Bei der äußerst lebendigen Mailliste *user@groovy.codehaus.org* können Sie im Fall eines konkreten Problems mit Groovy auf Unterstützung rechnen. Scheuen Sie sich nicht, Ihre Fragen auf Englisch zu stellen; in der weltweiten Groovy-Community gibt es viele Teilnehmer mit einer anderen Muttersprache. Bisherige Beiträge lassen sich bei *Nabble* unter *http://www.nabble.com/groovy---user-f11867.html* nachlesen.
- Es gibt noch einige weitere hilfreiche Informationsquellen im Internet, und während Sie dieses hier lesen, sind vielleicht schon wieder einige hinzugekommen. Eine spezielle Suchmaschine ermöglicht Ihnen, alle zugleich gezielt zu durchsuchen. Sie ist erreichbar unter *http://searchgroovy.org/*.

Was die gedruckte Literatur angeht, ist das Angebot zurzeit noch recht dünn. Auf dem deutschsprachigen Markt ist derzeit außer ein paar Ankündigungen und einer Reihe von

Zeitschriftenartikeln kaum etwas zu finden. Wenn Sie mehr lesen möchten, müssen Sie also derzeit auf englischsprachige Bücher ausweichen.

- Das Buch der Hauptakteure des Groovy-Projekts: *Groovy in Action* von Dierk Koenig, Andrew Glover, Paul King und Guillaume Laforge (Manning Verlag, englisch). Obwohl allgemein als die definitive Dokumentation zu Groovy angesehen, kann angesichts der rasanten Weiterentwicklung auch dieses dicke Buch (knapp 700 Seiten) nicht alle Bereiche vollständig abdecken.
- Ein Band namens *Groovy Recipes: Greasing the Wheels of Java* von Scott Davis und Venkat Subramaniam (Pragmatic Programmers, im Vertrieb von O'Reilly) zeigt anhand von zahlreichen Beispielen, wie konkrete Programmierprobleme mit Groovy effektiv gelöst werden können. Dieses Buch soll Ende 2007 erscheinen.
- Das Buch *Groovy Programming: An Introduction for Java Developers* von Kenneth Barclay und John Savage (Morgan Kaufmann Publishers) hat eher den Charakter eines Lehrbuchs und behandelt auch manche Themen eingehend, die Java-Programmierern eigentlich bekannt sein sollten.

Wie oben bereits gesagt, setzen wir in diesem Buch ein solides Grundwissen über die Programmiersprache Java und die Java-Standardbibliotheken voraus. Das ist natürlich ein weites Feld, und es kann durchaus sein, dass Sie an der einen oder anderen Stelle noch einmal nachschauen müssen, wie es denn mit Java genau funktioniert. Es gibt diverse gute Bücher über Java; eines davon ist sehr verbreitet und kann online kostenlos gelesen oder heruntergeladen werden, daher möchten wir es an diese Stelle nennen:

- *Java ist auch eine Insel: Programmieren mit der Java Standard Edition Version 6* von Christian Ullenboom. Online unter: *http://www.galileocomputing.de/openbook/javainsel6/*.

Wenn Sie im Text Verweise wie »siehe *Java-Insel*, 12.1« sehen, dann bedeutet dies, dass Sie unter dem genannten Abschnitt Hintergrundwissen finden, das hilfreich zum Verständnis des betreffenden Themas in diesem Buch sein kann.

Typografische Konventionen

Dieses Buch wendet folgende typografische Konventionen an:

Kursiv
: Für E-Mail-Adressen, Dateinamen, URLs, um neue Begriffe hervorzuheben, wenn sie erstmalig benutzt werden, und für Kommentare innerhalb von Codeabschnitten.

KAPITÄLCHEN
: Für Buttons, Menüeinträge und sonstige GUI-Elemente

`Nichtproportionalschrift`
: Für den Code, um den Inhalt von Dateien oder die Ausgabe von Kommandos zu zeigen und um Module, Methoden, Anweisungen und Kommandos zu kennzeichnen.

`Nichtproportionalschrift, fett`
: In Codeabschnitten, um Kommandos oder Text zu kennzeichnen, der von Ihnen eingegeben werden soll.

Die Website zum Buch

Zu *Groovy für Java-Entwickler* gibt es eine eigene Website, auf die Sie unter *http://www.acanthurida.de/groovy/* zugreifen können. Sie enthält weitere aktuelle Informationen als Ergänzung zu diesem Buch. Sie finden dort unter anderem:

- Codebeispiele aus dem Buch zum Herunterladen
- Fehlerkorrekturen zum Buch
- Tipps und Tricks zur Groovy-Programmierung
- aktuelle Neuigkeiten aus der Groovy-Welt

Weitere Informationen zum Buch finden Sie bei der Buchbeschreibung des O'Reilly Verlags unter *http://www.oreilly.de/catalog/groovyger*. Viel Spaß beim Surfen.

Danksagung

Auch wenn nur ein Name auf dem Cover steht, so ist denn ein solches Buch nie das Produkt einer Person allein. Deswegen möchte ich hier einige derjenigen erwähnen, die einen Beitrag zu dem geleistet haben, was Sie nun zwischen zwei Buchdeckeln vorfinden.

Die höchst engagierte Gruppe des Groovy-Projekts muss sicher wegen ihrer beeindruckenden Leistungen an erster Stelle stehen – allen voran Projektmanager Guillaume Laforge und Technical Lead Jochen Theodorou, die außerdem noch die Zeit und die Geduld finden, nicht enden wollende Diskussionen mit einer wachsenden Zahl von Anhängern und Nutzern zu führen.

Meine Partnerin Brigitte hat mich ermuntert, dieses doch etwas sportliche Projekt, ein Buch neben der alltäglichen Arbeitstätigkeit zu schreiben, anzugehen. Sie hat mitgetragen und mitgelitten – und mich mehr als einmal daran erinnert, dass es auch noch schöne Dinge jenseits von Groovy gibt.

Manche Freunde habe ich nur noch selten gesehen in diesen Monaten des Schreibens, in der die Zeit für alle nicht lebenswichtigen Dinge knapp wurde. Ihnen habe ich ziemlich viel Verständnis abverlangt.

Ja, *last* und absolut *not least* muss ich Christine Haite vom O'Reilly Verlag nennen, die dieses Buchprojekt gewagt hat. Gemeinsam mit den Fachgutachtern hat sie viele Fehler gefunden und manche gute Idee eingebracht. Und ohne ihre unwiderstehlichen Motivationsoffensiven hätte das Buch wohl keine Chance gehabt.

KAPITEL 1

Erste Schritte

Wie fangen wir an? Genau wie eine Fremdsprache lernt man auch eine Computersprache am besten, indem man sie benutzt. Deshalb wollen wir in diesem Kapitel die grundlegenden Voraussetzungen dafür schaffen, dass Sie die neue Sprache »sprechen« lernen. Dazu sollte zunächst einmal Groovy auf Ihrem Rechner verfügbar sein, und Sie sollten prinzipiell schon einmal wissen, welche Werkzeuge Ihnen damit zur Verfügung stehen und wie man diese benutzt. Dazu benötigen wir zwangsläufig schon einmal ein paar simple Groovy-Programme. So richtig los mit der Sprache selbst geht es aber erst in Kapitel 2, wenn wir auf die Einzelheiten eingehen, die Groovy von Java unterscheiden.

Groovy installieren

Natürlich müssen wir, bevor wir Groovy benutzen können, die zugehörigen Werkzeuge und Bibliotheken auf unserem Rechner verfügbar machen. Das ist keine besonders schwierige Aufgabe und sollte, sofern die Voraussetzungen erfüllt sind, in wenigen Minuten erledigt sein.

Voraussetzungen prüfen

Die wichtigste Voraussetzung für die Installation von Groovy auf Ihrem Rechner ist ein aktuelles *Java Development Kit* (JDK) oder zumindest das *Java Runtime Environment* JRE). Gegenwärtig (Groovy 1.1) muss es die Versionsnummer 1.4 oder höher haben. Frühere Versionen funktionieren nicht; wenn Ihr Java also älter ist, können Sie dies gleich zum Anlass nehmen, die aktuelle JDK- (bzw. JRE-) Version einzuspielen. Die Java-Version 5.0 oder höher ist nur dann von besonderem Nutzen, wenn Sie deren Bibliotheken von Groovy aus nutzen möchten. Zwar sind auch einige der Sprachneuerungen von Java 5.0 in Groovy abgebildet, aber da Groovy auf Java 1.4 lauffähig bleiben muss, kön-

nen sie nur sehr begrenzt genutzt werden.[1] Einige der neuen Java-5-Konzepte wie Annotationen sind in Groovy ohnehin wenig sinnvoll, da Groovy eine dynamisch typisierte Sprache ist; sie sind in Groovy unter dem Gedanken aufgenommen worden, die Integration mit Java zu verbessern.

Eine Möglichkeit, die korrekte Java-Version zu überprüfen, ist die Eingabe des folgenden Befehls in einem Konsolenfenster (unter Windows »Eingabeaufforderung« genannt):

```
> java -version
```

Daraufhin sollte eine Meldung wie die folgende erscheinen, deren erster Zeile Sie die Versionsnummer Ihrer Java-Installation entnehmen.

```
java version "1.5.0_06"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_06-b05)
Java HotSpot(TM) Client VM (build 1.5.0_06-b05, mixed mode, sharing)
```

Stellen Sie weiterhin sicher, dass die Umgebungsvariable `JAVA_HOME` korrekt gesetzt ist und auf das Wurzelverzeichnis Ihrer JDK-Installation verweist. Wenn Sie andere Java-basierte Software verwenden, wird dies in der Regel ohnehin der Fall sein. Andernfalls benutzen Sie die Mittel Ihres Betriebssystems, um die Umgebungsvariable systemweit oder benutzerspezifisch zu setzen. In Windows sind die Umgebungsvariablen aus der Systemsteuerung unter SYSTEM→ERWEITERT→UMGEBUNGSVARIABLEN zu erreichen. Bei Unix-artigen Systemen werden die Umgebungsvariablen in Profildateien definiert; im Detail ist dies von der Unix-Variante und der konkreten Installation abhängig.

Groovy herunterladen und entpacken

Öffnen Sie Ihren Browser, lenken Sie ihn zu der Seite *http://dist.codehaus.org/groovy/distributions/* und suchen Sie die neueste Groovy-Version als *.zip*- oder *.tar.gz*-Datei. Laden Sie diese herunter und entpacken Sie sie in einem temporären Verzeichnis.

Neuerdings können Sie Groovy auch in Form eines Windows-Installers erhalten. Wenn Sie mit Windows arbeiten und sich für diesen Weg entscheiden, brauchen Sie den Installer nur herunterzuladen und zu starten. Alles Weitere wird für Sie erledigt. Trotzdem ist es sinnvoll, vorher die folgenden Absätze zu lesen, damit Ihnen die Bedeutung der Fragen, die Ihnen bei der Installation gestellt werden, klar genug ist.

Angenommen, die von Ihnen heruntergeladene Datei hat den Namen *groovy-1.1.zip* und Sie haben sie unter Windows in das Verzeichnis *C:\Programme* entpackt, dann haben Sie darin nun ein neues Unterverzeichnis namens *C:\Programme\groovy-1.1*, in dem sich wiederum verschiedene Verzeichnisse und Dateien befinden.

Damit sind Sie schon fast fertig. Um Groovy nutzen zu können, sollten Sie nur noch zwei Anpassungen bei den Umgebungsvariablen vornehmen. Definieren Sie zunächst eine

1 Ab Version 2.0 wird Groovy allem Anschein nach die Java-Version 5.0 voraussetzen und damit die Neuerungen dieser Version in größerem Umfang nutzen können.

neue Umgebungsvariable GROOVY_HOME, die auf das Installationsverzeichnis verweist. In unserem Beispiel ist das also *C:\Programme\groovy-1.1*.

Fügen Sie der PATH-Variablen ein neues Element hinzu, das auf das Verzeichnis *bin* unterhalb des Groovy-Installationsverzeichnisses verweist. Unter Windows fügen Sie dazu wieder über die Systemsteuerung ;%GROOVY_HOME%\bin (also durch Semikolon getrennt) der Umgebungsvariablen PATH hinzu, bei Unix-artigen Betriebssystemen ist es :$GROOVY_HOME/bin (hier durch Doppelpunkt getrennt). Dies ist ein Ausschnitt aus einem Unix-typischen Initialisierungsskript:

```
# Java initialisieren
JAVA_HOME=/opt/jdk1.5.0
PATH=$PATH:$JAVA_HOME/bin
export JAVA_HOME
export PATH
# Groovy initialisieren
GROOVY_HOME=/home/user/groovy-1.1
PATH=$PATH:$GROOVY_HOME/bin
export GROOVY_HOME
export PATH
```

Als schnellen Test können Sie nun ein neues Konsolenfenster aufmachen und Folgendes eingeben:

```
> groovy -v
```

Daraufhin sollte so etwas wie die folgende Zeile erscheinen, die Ihnen mitteilt, welche Groovy-Version Sie soeben installiert haben und auf welcher Java-Version diese läuft.

```
Groovy Version: 1.1 JVM: 1.5.0_06-b05
```

Wenn Sie stattdessen eine Fehlermeldung vorfinden, müssen Sie sich leider erst einmal auf die Suche nach der Ursache begeben. Kleiner Tipp: Sehen Sie sich noch einmal an, ob die Umgebungsvariablen richtig gesetzt sind (auch JAVA_HOME) und ob, wenn Sie mit Unix arbeiten, die Shell-Skripte in *$GROOVY_HOME/bin* auf ausführbar gesetzt sind und die Shell neu gestartet wurde.

Wenn sich Groovy aufrufen lässt, könnten Sie noch ein Weiteres tun und die Groovy-Bibliotheken fest in Ihre Klassenpfadvariable CLASSPATH aufnehmen, damit sie beim Aufruf des Java-Compilers oder der JVM automatisch referenziert werden. Dies wird von manchen Anleitungen so empfohlen, wir raten aber eher davon ab, denn die Shell-Skripte (groovy, groovyc, groovysh usw.) fügen ihrerseits auch noch einmal Groovy-spezifische Bibliotheken in den Klassenpfad ein, und es kann dann zu Konflikten kommen.

Damit Sie wissen, was Sie mit dieser Installation eigentlich auf Ihrem Rechner angerichtet haben, wollen wir einmal die verschiedenen Verzeichnisse unterhalb Ihres Groovy-Installationsverzeichnisses durchgehen und feststellen, was sich eigentlich genau darin befindet.

bin
: Hier liegen die verschiedenen Dienstprogramme von Groovy in Form von Shell-Skripten für Unix und Batch-Dateien für Windows. Weiter unten werden sie noch einmal einzeln erläutert.

conf
: In diesem Verzeichnis befindet sich gegenwärtig nur eine Konfigurationsdatei, die festlegt, welche Laufzeitbibliotheken standardmäßig beim Aufruf von Groovy geladen werden.

docs
: Anders als Sie vielleicht erwarten, liegt hier keine umfassende Groovy-Dokumentation. Vielmehr befinden sich in diesem Verzeichnis einige HTML-Dateien zum Projekt. Interessant ist die Datei *groovy-jdk.html*, die eine aktuelle Übersicht der von Groovy vordefinierten Methoden liefert. Außerdem befinden sich in den Unterverzeichnissen *xref* und *xref-test* navigierbare HTML-Versionen der Groovy-Quelldateien. Die Dokumentation hingegen können Sie von der Groovy-Website herunterladen.

embeddable
: Hier befindet sich eine Java-Archivdatei, die für die Integration von Groovy-Programmen in Java benötigt wird.

lib
: Eine lange Reihe von Java-Bibliotheken, die überwiegend bei der Entwicklung von Groovy selbst benötigt werden. Als Groovy-Anwender benutzen Sie lieber die JAR-Datei im Verzeichnis *embeddable*.

Tabelle 1-1 enthält eine Übersicht der verschiedenen Dienstprogramme, die sich im Unterverzeichnis *bin* befinden. Eine ausführlichere Beschreibung einschließlich der jeweils verfügbaren Optionen finden Sie in Anhang D.

Tabelle 1-1: Übersicht der Groovy-Dienstprogramme[a]

Befehl	Beschreibung
`groovy`	Aufruf des Groovy-»Interpreters«. Das anzugebende Groovy-Programm, das aus unterschiedlichen Quellen kommen kann, wird direkt ausgeführt.
`groovyc`	Aufruf des Groovy-Compilers. Er übersetzt analog zum Java-Compiler `javac` das übergebene Groovy-Programm in Java-Bytecode.
`groovysh`	Aufruf der befehlszeilenorientierten Groovy-Konsole.
`groovyConsole`	Aufruf der grafischen Groovy-Konsole.

[a] Das Dokumentationswerkzeug *grok*, das analog zu *javadoc* funktionieren soll, und das Konvertierungsprogramm *java2doc* sind in der Version 1.1 leider noch nicht einsatzfähig.

Und was ist mit meiner Lieblings-IDE?

Dies ist vielleicht der einzige wirklich wunde Punkt, wenn Sie sich von Java kommend an die Sprache Groovy gewöhnen möchten. Die Toolunterstützung ist bei Weitem noch

nicht so ausgereift, wie Sie es von der Java-Entwicklung mit einer integrierten Entwicklungsumgebung (IDE), wie Eclipse, NetBeans, JBuilder, IntelliJ und wie sie alle heißen, gewohnt sind.

Dies liegt zum einen daran, dass Groovy wegen seiner dynamischen Eigenschaften weniger Möglichkeiten der Unterstützung bietet. Wenn der Editor nicht wissen kann, was für ein Objekt ich in einer Variablen speichere, und wenn diesem Objekt womöglich auch noch dynamisch Methoden zugeordnet werden können, dann kann er beim besten Willen nicht wissen, welche Methoden an ihr aufgerufen werden können. Was die statische Typisierung bei Java ermöglicht, nämlich zum Beispiel auf Knopfdruck alle Felder und Methoden des Objekts anzuzeigen, das sich gerade unter dem Cursor befindet, ist dann nicht mehr so ohne Weiteres möglich.

Dazu kommt, dass Groovy noch eine verhältnismäßig junge Sprache ist und deshalb die Werkzeuglandschaft zwangsläufig bisher nicht so weit entwickelt sein kann wie bei anderen Sprachen mit einer längeren Geschichte.

Trotzdem gibt es einige Plugins für gängige Java-Entwicklungsumgebungen (hervorzuheben ist hier das Eclipse-Plugin, das schon einen sehr guten Eindruck macht) und Highlighter für Texteditoren, die Ihnen zumindest Syntax-Coloring, zum Teil aber auch weiter gehende Fähigkeiten wie die Unterstützung des Debuggers bieten. Sie stellen schon mal eine wertvolle Hilfe dar und machen auch stetig Fortschritte, aber erwarten Sie nicht zu viel. Auf der Groovy-Website finden Sie unter IDE SUPPORT Informationen zu den aktuellen Plugins und deren Download-Adressen.

In den Beispielen zu diesem Buch werden wir generell davon ausgehen, dass Sie mit einem ganz normalen Texteditor arbeiten, in dem Sie Skripte schreiben, die Sie dann mit *groovy* ausführen. Andernfalls müssten wir ständig die Besonderheiten dieser oder jener Entwicklungsumgebung berücksichtigen. Wenn Sie Ihre Java-IDE gut kennen, dürfte es aber auch kein Problem sein, wenn Sie sich das passende Groovy-Plugin besorgen und in Ihrer gewohnten Umgebung weiterarbeiten.

Häufig gibt es aber auch »Einzeiler«, die Sie einfacher in einer der interaktiven Konsolen (*groovysh* und *groovyConsole*) nachvollziehen können, die Sie gleich noch näher kennenlernen werden. Dieser letztere Fall wird in den Beispielen durch fetten Druck und den vorangestellten Groovy-Prompt gekennzeichnet, wie die folgende kurze Zeitanzeige beispielhaft zeigt:

```
groovy> println new Date()
Mon Mar 21 15:02:35 CET 2007
```

Mit Groovy programmieren

Nachdem wir alles, was wir zum Arbeiten mit Groovy benötigen, auf unserem Rechner haben, sollten wir uns einmal ansehen, welche Möglichkeiten uns zur Verfügung stehen. Groovy kann auf unterschiedliche Weise verwendet werden, und dafür stehen verschie-

dene Werkzeuge zur Verfügung. Worin sie bestehen und wie sie zu bedienen sind, soll in diesem Abschnitt einführend erläutert werden.

Groovy – Skriptsprache oder Compilersprache?

Groovy ist gemeinhin als Skriptsprache bekannt, obwohl dies eigentlich nicht korrekt ist. Jedes in Groovy geschriebene Programm wird zunächst in Java-Bytecode-Klassen übersetzt und dann ausgeführt. Darin unterscheidet es sich nicht besonders von einem normalen, in Java geschriebenen Programm. Der Unterschied zu Java besteht darin, dass die Groovy-Laufzeitumgebung die Fähigkeit hat, Klassen erst zur Laufzeit zu kompilieren, und dass dieser Vorgang vom Anwender nicht unbedingt wahrgenommen wird, da normalerweise keine sichtbaren *.class*-Dateien mit dem Bytecode entstehen.

Groovy steht nicht nur wegen der verdeckten Kompilierung in dem Ruf, eine Skriptsprache zu sein, sondern auch aufgrund bestimmter Spracheigenschaften. Die auffälligsten von ihnen sind sicher der fehlende Zwang zur Typisierung von Variablen und die Möglichkeit, Code außerhalb von Klassen und Methoden zu schreiben. Auch das ist eigentlich nicht ganz zutreffend, denn auch in Groovy befindet sich letztendlich – konsequenter als in Java selbst – jede Information in einem Objekt, und jeder Code wird im Kontext einer Klasse ausgeführt. Nur geschieht dies auf sehr viel flexiblere Weise als in Java, und manche Schreibarbeit wird einfach vom Compiler übernommen.

Wir wollen die Frage, ob Groovy eine Skript- oder eine Compilersprache ist, hier nicht weiter verfolgen. Wir bleiben uns aber immer bewusst, dass jedes Stück Programm in Groovy immer erst zu einer Klasse kompiliert werden muss, bevor es ausgeführt werden kann.

Den Begriff »Skript« werden wir in diesem Buch auf solche Programme anwenden, deren Quellcode nicht durch eine Klassendefinition eingeschlossen ist, egal ob es interaktiv eingegeben wird, als Datei vorliegt oder aus irgendeiner anderen Quelle kommt, und egal, ob es erst zur Laufzeit kompiliert wird oder schon im Voraus. In diesem Sinn ist die folgende Zeile ein Groovy-Skript:

```
println 'Hallo Welt'
```

Und dies ist eine Groovy-Klasse – auch wenn es im Endeffekt dasselbe tut:

```
class HalloWelt {
  def main(args) {
    println 'Hallo Welt'
  }
}
```

Unabhängig davon, ob es sich bei einem Programm um ein Groovy-Skript oder eine Groovy-Klasse handelt, können Sie damit in ganz unterschiedlicher Weise umgehen:

1. Wenn es kurz ist und nicht wiederholt benötigt wird, können Sie das Programm auch interaktiv eintippen und sofort ausführen lassen. Dies machen Sie mit `groovysh` oder der `GroovyConsole`.

2. Sie können es mit `groovy` direkt ausführen lassen; dabei kann das Programm als *.groovy*-Datei oder als Stream vorliegen, oder es kann sogar als Befehlszeilenparameter mitgegeben werden.
3. Sie können es auch zu einer normalen *.class*-Datei kompilieren und diese entweder selbst starten oder in Java-Anwendungen einbinden. Dazu benötigen Sie den Groovy-Compiler *groovyc*.
4. Schließlich können Sie es auch innerhalb eines anderen (Java- oder Groovy-)Programms direkt aus dem Quellcode ausführen lassen.

Bei all diesen Möglichkeiten, mit Ausnahme der dritten, gibt es keinen gesonderten Kompiliervorgang, bei dem eine gesonderte `.class`-Datei entsteht. Trotzdem findet bei allen ein Übersetzen von Quellcode in Bytecode statt, auch wenn dieser unmittelbar als Klasse in die virtuelle Maschine geladen wird und sofort zur Ausführung bereitsteht.

Die ersten drei der genannten vier Möglichkeiten wollen wir uns jetzt näher ansehen. Die vierte würde den Rahmen dieses Einstiegs sprengen, daher werden wir sie später in Kapitel 3 wieder aufnehmen.

Groovy interaktiv

Zunächst möchten wir Sie mit zwei Möglichkeiten bekannt machen, Groovy-Skripte und -Programme laufen zu lassen, die eher zum Ausprobieren und Experimentieren als für die eigentliche Softwareentwicklung geeignet sind: die beiden Dienstprogramme `groovysh` und `groovyConsole`. Beide ermöglichen es, Groovy-Code einzugeben, ihn auszuführen und das Ergebnis zu betrachten. Daneben gibt es noch die Möglichkeit, Groovy im Servermodus zu betreiben und Skripte über das Netzwerk ausführen zu lassen.

Eine vollständige Referenz der verschiedenen Tooloptionen finden Sie in Anhang D.

Die Groovy-Shell (groovysh)

Dieses Tool ähnelt einer interaktiven Shell, Sie können Anweisungen eingeben und sie unmittelbar ausführen lassen. Sie werden gleich sehen, dass diese Ähnlichkeit nicht unbegrenzt ist. Sie starten die Shell durch Eingabe des Befehls `groovysh` in der Konsole (bei Windows können Sie auch auf START→AUSFÜHREN klicken und dann den Namen des Befehls eingeben).

Das Programm meldet sich mit einer Begrüßung und fordert Sie auf, eine Skriptzeile einzugeben. Sie können nun ein sich über mehrere Zeilen erstreckendes Skript eintippen, die Sie jeweils mit der Return-Taste abschließen.

Ausgeführt wird das Skript erst, wenn Sie den Befehl `go` eingeben. Das liegt daran, dass Groovy immer ein ganzes Programm übersetzen muss und es nicht zeilenweise ausführen kann wie etwa eine Unix-Shell. Sie sehen dann möglicherweise in Ihrem Skript enthaltene

Ausgabetexte und zum Abschluss das Ergebnis des letzten Befehls bzw. `===> null`, wenn der letzte Befehl des Skripts kein Ergebnis hat.

```
> groovysh
Lets get Groovy!
================
Version: 1.0-JSR-06 JVM: 1.5.0_06-b05
Type 'exit' to terminate the shell
Type 'help' for command help
Type 'go' to execute the statements

groovy> for ( p in System.properties.sort {x,y->x.key<=>y.key} ) {
groovy>   if ( p.key.startsWith('user')) println p
groovy> }
groovy> go

user.country=DE
user.dir=D:\Programme\Groovy\bin
user.home=C:\users\jst
user.language=de
user.name=JST
user.timezone=
user.variant=

===> null
```

Sie können nun weitere Skripte eingeben, wobei die Binding-Variablen von einem Skript zum nächsten erhalten bleiben. Den aktuellen Inhalt des Bindings können Sie sich mit dem Befehl `binding` anzeigen lassen. Es gibt einige weitere Befehle, die Sie sich mit dem Befehl `help` ansehen können. Beenden Sie Ihre Groovy-Session mit `exit` oder schließen Sie einfach das Konsolenfenster.

```
groovy> binding
Available variables in the current binding
x = 2.5
groovy> exit
```

Die Groovy-Konsole (groovyConsole)

Während `groovysh` zumindest gegenwärtig doch ein sehr rudimentäres Werkzeug ist, bietet die `groovyConsole` mit ihrer grafischen Oberfläche zumindest ein Mindestmaß an Komfort. Nach der Eingabe dieses Befehls öffnet sich ein Anwendungsfenster mit zwei untereinander angeordneten Textbereichen (siehe Abbildung 1-1). Im obigen können Sie ein beliebig langes Groovy-Skript eingeben. Wenn Sie es mit der Menüauswahl ACTIONS →RUN oder Strg-R starten, erscheinen im unteren Textbereich die Skripteingabe, wie sie in `groovysh` aussehen würde, eventuelle Ausgaben des Skripts sowie zum Abschluss auch wieder das Ergebnis des zuletzt ausgeführten Skriptbefehls.

Die Groovy-Konsole verfügt noch über einige weitere Möglichkeiten, beispielsweise zum Laden und Speichern von Dateien, die sich leicht aus dem Menü erschließen lassen.

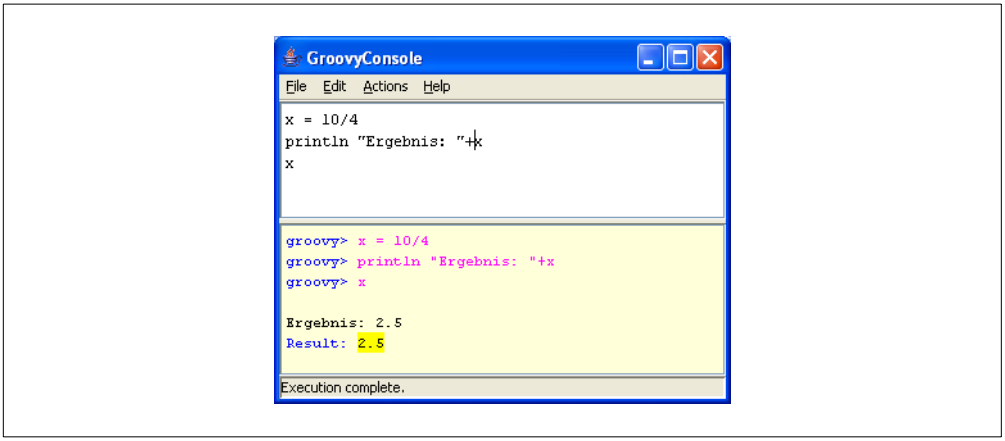


Abbildung 1-1: Die Groovy-Konsole

Das Skript als Aufrufoption (groovy -e)

Wenn Sie es ganz eilig haben und Ihr Programm nur aus einer oder wenigen Anweisungen besteht, was man in Groovy leicht hinbekommt, können Sie sich auch das Starten von `groovysh` oder `groovyConsole` sparen. Rufen Sie einfach im Befehlszeilenfenster den Befehl `groovy` so auf, dass Sie nicht den Namen einer auszuführenden Datei, sondern mit der Option `-e` die Anweisungen selbst als Aufrufargument angeben.

```
> groovy -e "for (p in System.properties) println p"
java.runtime.name=Java(TM) 2 Runtime Environment, Standard Edition
sun.boot.library.path=C:\Programme\Java\jdk1.5.0_06\jre\bin
...
```

Schneller kriegen Sie es wohl kaum hin, sich eben mal alle System-Properties der Java-Laufzeitumgebung ausgeben zu lassen. Aber warten Sie ab, wenn Sie erst etwas vertrauter mit den Stärken von Groovy sind, wird Ihnen vielleicht auch so mancher Einzeiler einfallen, der ihnen das Leben erleichtert.

Groovy-Quellprogramm erstellen und direkt ausführen

Von kleineren Experimenten abgesehen, wird man in der Praxis Groovy in der Regel nicht interaktiv mit `groovysh` und `groovyConsole`, sondern in der Form ausführbarer Dateien anwenden. Wie dies funktioniert, werden wir uns jetzt einmal ansehen.

Ein Beispielprogramm

Die verschiedenen Möglichkeiten, ein Groovy-Programm zu erstellen und in Gang zu setzen, demonstrieren wir am besten anhand eines einfachen Beispielprogramms. Wir wollen hier nicht noch einmal das alte »Hello World« strapazieren, sondern ein kleines Hilfsprogramm schreiben, das auch wirklich einen gewissen Nutzen hat. Und bei der Gelegenheit erhalten Sie gleich einen ersten Eindruck davon, was Groovy-Programme mit Java-Programmen gemein haben und was sie unterscheidet.

Unsere kleine Utility dient dazu, alle vorhandenen System-Properties der Java-Laufzeitumgebung nach Namen sortiert auszugeben. Wenn dem Programm ein Argument mitgegeben worden ist, werden jedoch nur diejenigen System-Properties angezeigt, deren Name mit diesem Argument beginnen. In Java könnte das etwa so aussehen, wie in Beispiel 1-1 dargestellt (wir verwenden dabei die Java 5-Syntax, aber keine Typparameter):

Beispiel 1-1: SystemProperties, Java-Version

```
import java.util.*;

public class SystemProperties {

    public static void main (String[] args) {
        String prefix = args.length > 0 ? args[0] : "";

        ArrayList entries = new ArrayList();
        for (Map.Entry e : System.getProperties().entrySet()) {
            if (e.getKey().toString().startsWith(prefix)) {
                entries.add(e);
            }
        }
        Collections.sort(entries,new MeinComparator());
        for (Object e : entries) {
            System.out.println(e);
        }
    }
}

class MeinComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        return ((Map.Entry)o1).getKey().toString()
                    .compareTo(((Map.Entry)o2).getKey().toString());
    }
}
```

Wenn Sie ein wenig Java verstehen (und davon gehen wir in diesem Buch aus), wird Ihnen schnell klar sein, was das Programm macht: Es baut eine `ArrayList` aus den Namen der `Map.Entry`-Elements auf, bei deren Schlüsselnamen das vorgegebene Präfix passt, sortiert die Liste mithilfe einer eigenen `Comparator`-Klasse und gibt sie schließlich der Reihe nach aus.

Sie kompilieren das Programm wie üblich:

```
> javac SystemProperties.java
```

Wenn Sie es starten, erhalten Sie alle System-Properties, und zwar fein säuberlich sortiert. Um die Selektion nach Präfix zu testen, geben wir ein Argument ein und definieren auch gleich eine passende zusätzliche System-Property namens `user.laune` in der Eingabezeile:

```
> java -Duser.laune=neugierig SystemProperties user
user.country=DE
user.dir=C:\workspace\groovy\Kap-01\src
user.home=C:\Users\jst
user.language=de
user.laune=neugierig
user.name=jst
user.timezone=
user.variant=
```

Denselben Programmcode könnten Sie auch – mit demselben Ergebnis – als Groovy-Programm verwenden. Das ist nicht besonders eindrucksvoll, denn dazu brauchen Sie keine neue Programmiersprache, aber es funktioniert. Groovy gibt uns aber die Möglichkeit, das Programm von allerlei Ballast zu befreien, der eigentlich unnötig ist. Beispiel 1-2 zeigt eine Variante von `SystemProperties`, die einige der arbeitssparenden Möglichkeiten der Sprache Groovy nutzt.

Beispiel 1-2: SystemProperties, Groovy-Version

```
prefix = args.length ? args[0] : ""

entries = []
for (e in System.properties) {
    if (e.key.startsWith(prefix)) {
        entries.add(e)
    }
}

entries.sort { e1,e2->e1.key.compareTo(e2.key) }

for (entry in entries) {
    println entry
}
```

Die Logik des Programms ist unverändert geblieben, wir haben aber allen Ballast entfernt, der in Java notwendig ist, damit Sie das Programm überhaupt kompilieren können, den Groovy aber nicht unbedingt benötigt, weil auch so zu erkennen ist, was das Programm tun soll. Dazu gehören einige Importanweisungen, die Klassendefinition mit der `main()`-Methode, die Deklaration der Variablen, Semikola zur Begrenzung der Anweisungen an Zeilenenden sowie Klammern um einfache Methodenparameter (wie bei `println()` in der vorletzten Zeile).

Außerdem haben wir einige der Möglichkeiten von Groovy angewendet, mit weniger Code mehr auszudrücken, zum Beispiel beim Instantiieren einer `ArrayList` mit zwei eckigen Klammern (`entries = []`). Schließlich haben wir ein neues Konzept verwendet, das es in Java überhaupt nicht gibt: die Closure. Wir benutzen sie zum Sortieren der Liste `entries` anstelle der Klasse `MeinComparator`.

Vermutlich werden Ihnen jetzt nicht alle Details dieses Programms verständlich sein – nach der Lektüre der beiden folgenden Kapitel werden Sie da klarer sehen –, aber Sie können sicher erkennen, wie der Code des Java-Programms zusammengeschrumpft ist. Dabei waren wir noch zurückhaltend bei der Komprimierung, damit Sie die Verwandtschaft mit Java deutlich erkennen können. Wenn man alle Möglichkeiten nutzt, ist es nämlich durchaus möglich, in Groovy das ganze Programm in eine einzige Anweisung zu schreiben, wie Beispiel 1-3 zeigt.

Beispiel 1-3: SystemProperties, komprimierte Groovy-Version

```
System.properties.entrySet()
    .findAll { it.key.startsWith(args.length?args[0]:"") }
    .sort { x,y -> x.key<=>y.key }
    .each { println it }
```

Wir speichern das Programm unter dem Namen *SystemPropertiesAnzeiger.groovy* ab. Welche der drei Versionen wir nehmen, ist dabei völlig egal, denn sie werden alle drei von Groovy verstanden und tun genau dasselbe.

Sie brauchen nicht unbedingt die Dateiendung *.groovy* zu benutzen. Stattdessen können Sie auch eine der Endungen *.gvy*, *.gy* und *.gsh* verwenden oder die Endung ganz weglassen. Es hat sich aber mehr oder weniger eingebürgert, die Endung *.groovy* zu verwenden oder – nur in Unix-artigen Betriebssystemen – ganz auf die Endung zu verzichten, wenn das Programm ähnlich wie ein Shell-Skript direkt aufgerufen werden soll.

Ein Groovy-Programm direkt ausführen

Gehen Sie mit der Konsole in das Verzeichnis, in dem sich Ihr neues Skript *SystemProperties.groovy* befindet, und geben Sie die folgende Anweisung ein:

```
> groovy SystemProperties.groovy
```

Nun sollten alle in Java gesetzten System-Properties ordentlich sortiert erscheinen. Experimentieren Sie etwas herum, indem Sie beim Aufruf verschiedene Parameter mitgeben, um zu sehen, dass die richtigen Properties herausgefiltert werden. Sie können übrigens auch die Endung *.groovy* beim Aufruf weglassen; wenn Ihre Programmdatei eine der Endungen *.groovy*, *.gvy* oder *.gy* hat, wird sie auch dann noch gefunden.

Natürlich stehen Ihnen die betriebssystemspezifischen Möglichkeiten zur Verfügung, mit deren Hilfe Sie den Skriptaufruf vereinfachen können. Unter Unix-ähnlichen Betriebssystemen etwa können Sie folgende Zeile an den Anfang Ihres Skripts setzen:

```
#!/usr/bin/env groovy
```

Sorgen Sie noch dafür, dass das Skript ausführbar ist, dann brauchen Sie jetzt lediglich den Namen des Skripts einzugeben, damit es startet:

```
> chmod +x SystemProperties.groovy
> ./SystemProperties.groovy
```

Wenn Sie noch die Dateiendung entfernen und das Skript in Ihr *bin*-Verzeichnis – oder in ein anderes Verzeichnis im Unix-Pfad – aufnehmen, können Sie es einfach mit `SystemProperties` aufrufen, wie jeden anderen Unix-Befehl.

Unter Windows könnte es sinnvoll sein, die Dateiendung *.groovy* mit dem Befehl `groovy` zu verknüpfen, damit Sie Groovy-Skripte per Mausklick aus dem Explorer starten können. Nun ist es aber unter Windows ziemlich mühselig, eine Verknüpfung mit einer Batchdatei wie dem Groovy-Startskript *groovy.bat* herzustellen. Im obigen Beispiel wäre das Ergebnis zwar nicht besonders beeindruckend, denn es würde nur kurz ein Fenster mit den ausgegebenen System-Properties erscheinen und sofort wieder verschwinden. Aber Sie können ja mit Groovy beispielsweise auch Programme mit grafischer Oberfläche erstellen, bei denen ein solcher Start durchaus angemessen ist.

Eine Lösung für das Problem der Verknüpfung zwischen Dateiendung und Programm stellt der Groovy Native Launcher dar, der für verschiedene Betriebssysteme, darunter auch Windows, verfügbar ist und von den Groovy-Seiten heruntergeladen werden kann. Sie können das Programm einfach in das *bin*-Verzeichnis unter dem Groovy-Wurzelverzeichnis ablegen und anstelle des Skripts verwenden. Weitere Informationen unter *http://docs.codehaus.org/display/GROOVY/Native+Launcher*.

Für den Skriptaufruf gibt es eine Reihe von Optionen, die Sie sich mit `groovy -h` auflisten lassen können. In Anhang D sind sie noch einmal einzeln aufgeführt.

Groovy-Programme kompilieren und starten

Eine der Besonderheiten von Groovy ist, dass Sie die in dieser Sprache erstellten Programme sowohl als Skript, also direkt aus dem Quellprogramm heraus, als auch in kompilierter Form verwenden können. Dies wollen wir mit dem obigen SystemProperties-Programm ausprobieren.

Ein kompiliertes Groovy-Skript

Jedes Groovy-Programm, das sich als Skript ausführen lässt, kann auch kompiliert werden. Wenn Sie mit der Konsole noch immer in dem Verzeichnis stehen, in dem sich das oben erstellte Skript *SystemProperties.groovy* befindet, rufen Sie doch einmal den Groovy-Compiler auf:

```
> groovyc SystemProperties.groovy
```

Es geschieht nichts weiter, als dass nach kurzer Zeit die Eingabeaufforderung wieder erscheint. Allerdings befindet sich nun in Ihrem Verzeichnis eine neue Datei namens *SystemProperties.class*, und dies ist tatsächlich ein mithilfe der Java-Laufzeitumgebung ausführbares Java-Bytecode-Programm. Wenn Sie die *Java Virtual Machine* (JVM) im Pfad haben, können Sie das Programm direkt mit Java starten. Allerdings muss sich die Groovy-Laufzeitumgebung im Klassenpfad befinden, daher ist der Aufruf etwas länglich:

```
> java -cp %GROOVY_HOME%\embeddable\groovy-all-1.1.jar;. SystemProperties
user.country=DE
user.dir=C:\workspace\groovy\Kap-01\src
...
```

Tatsächlich hat der Groovy-Compiler das Skript in eine Java-Klasse übersetzt, die denselben Namen hat wie die Skriptdatei, also `SystemProperties`. Außerdem hat er ihr noch eine `main()`-Methode hinzugefügt, damit sie direkt gestartet werden kann.

Da Groovy bei Skripten Klassennamen aus dem Dateinamen der Quelldatei generiert, müssen Sie etwas vorsichtig bei der Benennung der Datei sein. Der Dateiname muss ein in Groovy gültiger Bezeichner sein, darf nicht mit einem reservierten Wort wie `for` oder `class` übereinstimmen und sollte auch sonst keinen Anlass zu Namenskonflikten bieten.

Eine kompilierte Groovy-Klasse

Ändern Sie das Programm *SystemProperties.groovy* mit dem Texteditor wie in Beispiel 1-4 dargestellt, und speichern Sie es unter dem Namen *SystemPropertiesAnzeiger.groovy* ab.

Beispiel 1-4: Die Klasse SystemPropertiesAnzeiger

```
class SystemPropertiesAnzeiger {

    static void main (args) {
        def prefix = args.length ? args[0] : ""
        def entries = []
        for (e in System.properties) {
            if (e.key.startsWith(prefix)) {
                entries.add(e)
            }
        }
```

Beispiel 1-4: Die Klasse SystemPropertiesAnzeiger (Fortsetzung)

```
        entries.sort { e1,e2 -> e1.key.compareTo(e2.key) }

        for (entry in entries) {
            println entry
        }
    }
}
```

Der einzige Unterschied besteht darin, dass wir das Programm jetzt in der `main()`-Methode einer Klasse haben und dass die beiden Variablen `prefix` und `entries` jetzt mit `def` deklariert sind. Letzteres ist nötig, weil es jetzt lokale Variablen der Methode sind, die auch in Groovy deklariert sein müssen – allerdings bleiben sie mit der `def`-Deklaration untypisiert.

Kompilieren wir die Klasse wie gehabt:

```
> groovyc SystemPropertiesAnzeiger.groovy
```

Wiederum entsteht eine Java-Klassendatei namens *SystemPropertiesAnzeiger*, die wir auch gleich mithilfe der JVM starten:

```
> java -cp %GROOVY_HOME%\embeddable\groovy-all-1.1.jar;. SystemPropertiesAnzeiger
user.country=DE
user.dir=C:\workspace\groovy\Kap-01\src
...
```

Alles wie gehabt, nur dass inzwischen ein paar mehr Dateien in unserem Verzeichnis stehen. Der Groovy-Compiler hat es nun etwas leichter, denn er braucht jetzt keine Klasse und keine `main()`-Methode mehr zu generieren, da wir diese schon selbst geschrieben hatten.

Und lässt sich das Programm *SystemPropertiesAnzeiger.groovy* nun nicht mehr als Skript ausführen? Doch, auch das ist möglich, wie ein kurzer Versuch zeigt:

```
> groovy SystemPropertiesAnzeiger.groovy
```

Dies funktioniert genauso. Groovy kompiliert die Klasse einfach schnell und führt sie gleich aus, ohne sie irgendwo als Klassendatei zu speichern. Mit anderen Worten: Sie können durchaus eine Groovy-Klasse wie ein Skript und ein Groovy-Skript wie eine Klasse behandeln. Für Groovy macht dies keinen großen Unterschied, vielmehr ist es eher eine Frage der praktischen Erfordernisse, wie Sie die Sprache benutzen: Während für das schnelle kleine Hilfs- oder Testprogramm ein Skript ausreicht, ist in größeren Projekten, bei denen Wiederverwendung und Strukturierung an erster Stelle stehen, eher das Arbeiten mit ordentlichen Klassen angesagt.

Sie haben Groovy auf Ihrem Rechner installiert und wissen nun, wie Sie damit Programme schreiben und starten können. Das reicht fast schon aus, um mit Groovy arbeiten zu können, denn in vielen Fällen reichen Ihre Java-Kenntnisse aus, um Groovy-Programme zu schreiben. Es gibt aber auch ein paar Abweichungen zwischen der Java- und der Groovy-Sprachsyntax, die Sie kennen müssen. Diese kleinen Unterschiede und die vielen Vereinfachungen und zusätzlichen sprachlichen Mittel von Groovy sind das Thema des nächsten Kapitels.

KAPITEL 2

Die Sprache Groovy

Wenn Sie sich mit Java schon etwas auskennen, stellt es für Sie kein großes Problem dar, ein Groovy-Programm zu schreiben. Trotzdem gibt es aber einige kleine und feine Unterschiede in der Syntax und einige nicht ganz unerhebliche Differenzen in der Semantik der beiden Sprachen, die man kennen muss, um von Java kommend sinnvoll mit Groovy arbeiten zu können. Neben diesen Unterschieden zwischen Groovy und Java wollen wir einige wesentliche Sprachelemente vorstellen, die nicht von Java her bekannt sind, um das Verständnis der folgenden Kapitel zu erleichtern. Diese neuen Dinge können hier jedoch nur angerissen werden, um Ihnen einen ersten Eindruck zu vermitteln; spätere Kapitel gehen dann vollständiger und ausführlicher darauf ein.

Was Sie einfach weglassen können

Es gibt einige Sprachelemente, die Sie von Java her gewohnt sind, die Sie aber in Groovy-Programmen einfach weglassen können, wenn Sie wollen, ohne dass sich der Compiler beschwert und ohne dass dies zu einer inhaltlichen Veränderung des Programms führt.

In welchem Umfang Sie von dieser Möglichkeit Gebrauch machen möchten, bleibt am Ende Ihnen überlassen. Manche sehen darin eine Möglichkeit, den Code übersichtlicher (weil schlanker) zu gestalten, andere halten lieber an der gewohnten Syntax fest. Für beides gibt es Argumente wie Vertrautheit, Schnelligkeit beim Schreiben oder Lesbarkeit, die – je nach Geschmack – auf beide Schreibweisen anwendbar sind.

Warum ist Groovy anders?

Die Unterschiede zwischen den Sprachen Groovy und Java lassen sich durch drei Prinzipien kennzeichnen:

Weniger Code – weniger Arbeit. Groovy-Programme sind fast immer deutlich kürzer als Java-Programme mit derselben Funktionalität. Dies wird dadurch erreicht, dass in Groovy vieles einfach weggelassen werden kann, was ohnehin klar ist. Beispiele hierfür sind die Klassendeklaration und die `main()`-Methode in einem einfachen Skript oder die Getter und Setter in Bean-Klassen. Daneben gibt es in Groovy eine Reihe neuer, mächtigerer Sprachkonstrukte und raffinierter APIs, die es ermöglichen, mit weniger Code mehr zu erreichen als in Java.

Übersichtlichere Programme. Wenn Sie Teile des Programmcodes einfach weglassen können, weil sie der Compiler auch so versteht, hat dies die angenehme Nebenwirkung, dass die Programme wesentlich besser lesbar werden. Ein Beispiel hierfür sind die Getter- und Setter-Methoden, die Sie in Java der Konvention folgend für jedes von außen sichtbare Feld einer Klasse schreiben und die in Groovy entfallen können, wenn sie keine zusätzliche Funktionalität außer dem Setzen oder Auslesen eines korrespondierenden Felds haben. Das befreit den Code von Ballast und erleichtert es, den fachlichen Gehalt des Codes zu erkennen.

Größtmögliche Erwartungskonformität (Principle of least surprise). Diesem im Zusammenhang mit der Programmiersprache Ruby bekannt gewordenen Prinzip zufolge soll das Programm das tun, was Sie von ihm erwarten, und Sie nicht mit überraschenden Ergebnissen konfrontieren – was in der Regel Fehler sind. Hierzu zwei Beispiele:

- In Groovy hat der Ausdruck `"abc".toUpperCase()=="abc"` den Wert `true`, während er in Java `false` ist (da Java hier auf Objektidentität prüft und nicht auf Gleichheit).
- Groovy berechnet bei der Formel `1.3+0.4` den Wert `1.7`, während Java auf `1.7000000000000002` kommt (da Java mit Fließkommazahlen arbeitet, Groovy dagegen standardmäßig mit Dezimalzahlen der Klasse `java.lang.BigDecimal`).

Das Prinzip der größtmöglichen Erwartungskonformität hat allerdings einen wesentlichen Nachteil: Es ist vom Betrachter abhängig. Einen erfahrenen Java-Programmierer wird das Verhalten eines Java-Programms in den beiden genannten Fällen nicht überraschen; im Gegenteil, wird er sich wundern, dass man mit == die Gleichheit von String-Inhalten abfragen kann. Yukihiro Matsumoto, der Erfinder von Ruby, hat einmal davon gesprochen, dass die größtmögliche Erwartungskonformität bei Ruby sich auf *seine* Erwartungen bezieht, die Erwartungen aller möglicher Programmier kann er nicht kennen. Und so ist es sicher auch hier: Groovy-Programme verhalten sich so, wie die Entwickler dieser Sprache es für sinnvoll erachten, und nicht unbedingt, wie Sie es erwarten.

Wir werden in diesem Buch die Vereinfachungsmöglichkeiten überwiegend nutzen, ohne dabei zu übertreiben, damit Sie sich mit der neuen Optik vertraut machen können.

Neue Konventionen gesucht

Wie bei Java, so werden sich auch bezüglich Groovy im Laufe der Zeit Konventionen dazu herausbilden, wie mit den Möglichkeiten der Sprache in einer einheitlichen Weise umgegangen wird. Für Groovy gilt dies insofern verstärkt, als die Sprache Ihnen häufig viele unterschiedliche Möglichkeiten bietet, ein Problem zu lösen.

In jedem Fall scheint es sinnvoll, sich Konventionen für die eigene Arbeit zu überlegen, um das gleiche Problem im Code nicht mal so und mal so zu lösen. Insbesondere gilt dies natürlich für Teams, die an einem gemeinsamen Projekt arbeiten. Genau so, wie es üblich ist, sich in Java-Projekten auf gemeinsame Codierungsrichtlinien zu einigen, sollte man es auch in Groovy-Projekten tun. Wenn Sie das vorhaben, können Sie beispielsweise die üblichen Java-Richtlinien zur Grundlage nehmen und Groovy-spezifisch ergänzen.

Klassendefinition und Main-Methode

Bei einfachen, skriptartigen Programmen können Sie auf die explizite Definition einer Klasse verzichten und einfach die auszuführenden Anweisungen ohne `main()`-Methode der Reihe nach aufschreiben. Wir haben von dieser Möglichkeit schon im ersten Kapitel Gebrauch gemacht und werden sie im ganzen Buch für kurze Beispiele nutzen. Wenn Sie Ihre Programme durch Funktionen strukturieren möchten, schreiben Sie diese einfach an beliebiger Stelle dazu, sie haben die gleiche Form wie die Methoden von Klassen (und wir werden sie fortan, um nicht immer unterscheiden zu müssen, auch als »Methoden« bezeichnen). Der Groovy-Compiler generiert daraus eine lauffähige Klasse mit dem Namen der Quelldatei (ohne Endung), deren `main()`-Methode das Skript aufruft.

Skriptcode und Klassen können in einer Datei beliebig gemischt sein; wenn allerdings eine Klasse den gleichen Namen hat wie die Datei, in der sie definiert ist, also etwa `MeineKlasse` in der Datei *MeineKlasse.groovy*, darf sich dort kein Skriptcode befinden, denn dafür müsste ja ebenfalls eine Klasse namens `MeineKlasse` generiert werden.

Mehr zur Arbeit mit Groovy-Skripten in Kapitel 3.

Das Semikolon am Ende einer Anweisung

In Java müssen Sie eine Anweisung *immer* mit einem Semikolon abschließen. In einem Groovy-Programm ist das in aller Regel nicht erforderlich. Das Semikolon wird nur dann benötigt, wenn sich mehrere Anweisungen in einer Zeile befinden. Statt

```
println ("Hello World");
```

können Sie also auch schreiben:

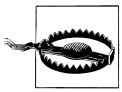
```
println ("Hello World")
```

Mit dieser kleinen Änderung verliert das Zeilenende gewissermaßen seine Unschuld, die es in allen Vorgängersprachen wie Pascal und C/C++ und auch in C# noch genießt. Ein Zeilenvorschubzeichen spielt in diesen Sprachen wie auch in Java genau die gleiche Rolle wie ein Leerzeichen oder ein Tabulatorzeichen: Außerhalb eines Strings trennt es zwei Elemente der Sprache (»Token«), ansonsten hat es keinerlei eigene Bedeutung. In Groovy dagegen hat das Zeilenvorschubzeichen eine eigene Bedeutung: Es schließt eine Groovy-Anweisung ab, sofern dies an der gegebenen Stelle im Code syntaktisch zulässig ist.

Wenn Sie verhindern möchten, dass ein Zeilenende als Ende einer Anweisung erkannt wird, können Sie dies durch einen Backslash am Ende der Zeile deutlich machen.

```
x = 1 \
    + 2
```

Das Backslash-Zeichen am Ende einer Zeile führt in jedem Fall dazu, dass die nächste Zeile vom Compiler logisch als Fortsetzung der aktuellen Zeile betrachtet wird.



An diese Eigenheit von Groovy, Zeilenenden als Anweisungsenden zu betrachten, wenn es passt, gewöhnen Sie sich schnell. Sie ist aber eine Quelle tückischer Fehler, da sie als Ursache des resultierenden Fehlverhaltens schwer zu erkennen ist. Wenn Sie beispielsweise im obigen Beispiel den Backslash vergessen:

```
x = 1
    + 2
```

haben Sie zwei gültige Anweisungen und bekommen keine Fehlermeldung, obwohl es sicher nicht Ihrer Erwartung entspricht, dass die Variable `x` am Ende den Wert 1 hat und die zweite Zeile überhaupt nichts bewirkt. Gewöhnen Sie sich am besten daran, öffnende Klammern und binäre Operatoren *immer* an das Ende der ersten Zeile und nicht an den Anfang der zweiten Zeile zu schreiben, um dem Compiler klarzumachen, dass in der nächsten Zeile noch etwas kommt.

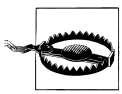
Die Klammern um Methodenparameter

Beim Aufrufen von Methoden auf der obersten Ebene brauchen Sie die Argumente nicht unbedingt einzuklammern. Dies gilt also nur, wenn der Methodenaufruf nicht schon selbst Teil eines Ausdrucks ist. Statt

```
println ("Hello World")
```

können Sie auch einfach dies schreiben:

```
println "Hello World"
```



Auch hier kommt es leicht zu Fehlern, über die schon mancher Groovy-Programmierer lange gegrübelt hat.

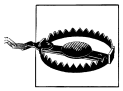
Leere Argumentlisten müssen ihre Klammern behalten. In der folgenden Programmzeile, die nur einen Zeilenvorschub ausgibt, können Sie die Klammern also nicht weglassen.

```
println ()  // Hier dürfen die Klammern nicht weggelassen werden.
```

Auch nicht weglassen können Sie die Klammern um die Argumente in Konstruktoraufrufen.

```
var = new StringBuilder ("Anfangstext") // Hier auch nicht.
```

Wenn Sie das nicht ganz konsistent finden, kann ich Ihnen schwerlich widersprechen.



Auch hier kommt es leicht zu Fehlern, über die schon mancher Groovy-Programmierer lange gegrübelt hat. Sehen Sie sich diese Anweisung an:

```
println (a+b)/c
```

Eigentlich sollte klar sein, dass wir einen Wert aus drei Variablen errechnen und diesen ausgeben wollen. Es tritt aber eine Fehlermeldung auf, da Groovy annimmt, dass nur der Inhalt der Klammer `(a+b)` an `println()` übergeben werden soll und das Ergebnis des `println()`-Aufrufs durch `c` geteilt werden soll, was natürlich nicht geht.

Machen Sie von der Möglichkeit, die Klammern um Parameter wegzulassen, nur in ganz trivialen Situation Gebrauch. Und setzen Sie die Klammern auf jeden Fall immer, wenn das erste zu übergebende Argument bereits mit einer Klammer beginnt.

Das Return am Ende einer Methode

Wenn sich am Ende eines Methodenrumpfs kein explizites `return` befindet, wird das Ergebnis der letzten Anweisung als Methodenergebnis zurückgegeben. Daher bestehen Methoden in Groovy bisweilen nur aus einem einzeln stehenden Ausdruck:

```
Integer addiere (Integer a, Integer b) {
    a+b
}
```

Wir hätten natürlich auch `return a+b` schreiben können, aber der Ausdruck allein reicht völlig aus. Wenn die letzte Anweisung eine `void`-Methode aufruft und daher kein Ergebnis hat, wird folgerichtig `null` zurückgegeben. Hierzu ein Beispiel:

```
String protokolliere (String text) {
     println("Protokoll "+new Date()+": "+text)
}
```

Die Methode `protokolliere` liefert unabhängig davon, dass sie mit einem `String`-Ergebnis deklariert ist, immer das Ergebnis `null`, denn die Methode `println()` in der einzigen und letzten Anweisung ist als `void` definiert.

Die am häufigsten gebrauchten Imports

In jedem Java-Programm ist das Package `java.lang` automatisch importiert, ohne dass Sie dies besonders erwähnen müssen. Damit stehen Ihnen unter anderem die String-Klasse, die Wrapper-Klassen zu den primitiven Typen wie `Boolean` und `Integer` sowie die grundlegenden Exception-Klassen zur Verfügung. Es sind genau diejenigen Klassen, die auch dem Compiler und der Laufzeitumgebung bekannt sein müssen, um überhaupt ein Java-Programm zu verstehen und ausführen zu können.

Groovy führt gleich noch ein paar weitere Importanweisungen automatisch aus; diese Zeilen können Sie also in einem Groovy-Programm immer weglassen:

```
import java.lang.*
import java.util.*
import java.net.*
import java.io.*
import java.math.BigInteger
import java.math.BidDecimal
import groovy.lang.*
import groovy.util.*
```

Das ist nicht nur für Sie bequemer (Sie wissen, dass etwa `java.util` am Anfang von fast jeder Java-Quellcode-Datei importiert wird), sondern auch deshalb notwendig, weil die Sprache Groovy wesentlich mehr Typen direkt unterstützt als Java; zum Beispiel gibt es spezielle Literale für `java.util.HashMap` und `java.util.ArrayList`. Daher müssen diese Klassen in jedem Programm bekannt sein.

Die nicht erforderliche Prüfung von Checked Exceptions

Wie es seiner dynamischen Natur entspricht, unterscheidet Groovy nicht zwischen *Checked* und *Unchecked Exceptions*. Exceptions werden wie Typen nur zur Laufzeit geprüft. Sie sind also nie gezwungen, Exceptions abzufangen, und demzufolge wird auch nicht die `throws`-Klausel für Methodendeklarationen benötigt. Ob dies ein Vorteil ist, kann sicher diskutiert werden; fest steht aber, dass es eine nicht zu vernachlässigende Anzahl von Java-Experten gibt, die die Einführung der checked Exception in Java für einen Missgriff halten.

Auf jeden Fall wird der Code lesbarer, wenn ein großer Teil der Exception-Prüfungen, die ja häufig den Charakter von Pflichtübungen haben, entfallen. Und Sie finden in Groovy-Programmen kaum noch jene erstrangige Ursache für äußerst verzwickte Fehler: das vergessene Error-Handling.

```
// Java
try {
    ...
} catch (EineException ex) {}
```

Natürlich geht das auch in Groovy, da Sie aber nie *gezwungen* sind, Fehler zu prüfen, besteht auch nicht die Notwendigkeit, derartige Provisorien zu schreiben.

Neue Sprachelemente

Die Sprache Groovy umfasst einige neue syntaktische Elemente, die es in Java nicht in dieser Form gibt. Im Wesentlichen dienen sie dazu, den Schreibaufwand bei häufig vorkommenden Programmierproblemen zu vermindern und die Programme übersichtlicher zu machen. Sie können ohne Weiteres Groovy-Programme schreiben, die von diesen Möglichkeiten keinen Gebrauch machen, in der Praxis der Groovy-Programmierung spielen sie jedoch eine wichtige Rolle, da sie einen wesentlichen Vorteil des Arbeitens mit Groovy gegenüber Java darstellen.

Wir werden auf die in diesem Abschnitt angesprochenen Themen in den folgenden Kapiteln noch ausführlicher eingehen, es ist aber hilfreich, wenn Sie schon mal einen Überblick haben und die wichtigsten Begriffe kennen, um das Verständnis des Folgenden zu erleichtern.

Neue Schlüsselwörter

Groovy kennt einige neue Schlüsselwörter, die es in Java nicht gibt.

in

Das Schlüsselwort `in` dient als Operator für die Prüfung, ob ein Objekt in einem Behälterobjekt, zum Beispiel einer `Collection`, enthalten ist oder nicht. Dieser Operator wird durch die Methode `contains()` implementiert. Außerdem wird das Wort in einer Form der `for`-Schleife benutzt.

```
if (zeichen in ['A','B','C']) ...
for (zeichen in ['A','B','C']) { ... }
```

as

Das Schlüsselwort `as` ist ein Operator für die explizite Typanpassung. Dieser bewirkt, dass Groovy versucht, den vor `as` stehenden Wert auf den dahinter angegebenen Typ abzubilden. Da Groovy weit gehende Möglichkeiten der Typanpassung bietet, kann dieser Operator in den verschiedensten Situationen eingesetzt werden, zum Beispiel bei der Initialisierung von Arrays mithilfe von Listen oder für die Übergabe von primitiven Variablen an Java-APIs. In vielen Fällen genügt allerdings auch die automatische Typanpassung durch Groovy.

```
stringArray = ['A','B','C'] as String[]    // Initialisierung eines Arrays
javaObjekt.aufruf (zahl as int)            // Übergabe der Zahl als int statt Integer
```

Unterstützte Typen

Groovy unterstützt einige häufig benutzte Java-Standardtypen direkt in der Sprache durch Initialisierungskonstrukte und Operatoren.

Listen

Zur Initialisierung einer Liste kann eine Aufzählung der Elemente in eckigen Klammern verwendet werden; sie erzeugt ein Objekt vom Typ `java.util.ArrayList`.

```
neueListe = ['alpha','beta','gamma','delta']
leereListe = []
```

Auf die Elemente einer Liste greifen Sie wie auf die Elemente eines Arrays zu.

```
zweitesMitIndex2 = neueListe[2] // entspricht neueListe.get(2)
```

Sie können auch mehrere Indizes oder ganze Bereiche von Indizes angeben, in diesem Fall ist das Ergebnis jedoch wieder eine Liste und nicht ein einzelnes Element der Liste.

```
println neueListe[1,2]  // ergibt ['beta','gamma']
println neueListe[2,1]  // ergibt ['gamma','beta']
println neueListe[1..3]  // ergibt ['beta','gamma','delta']
```

Der Ausdruck `1..3` in der letzten Zeile ist ein Beispiel für einen Wertebereich *(Range)*, den wir weiter unten behandeln.

Schließlich akzeptiert Groovy auch negative Indizes; sie werden vom Ende der Liste an rückwärts gezählt – `liste[x]` entspricht bei einem negativen `x` also dem Ausdruck `liste[x+liste.length()]`.

```
println neueListe[-1]  // ergibt 'delta'
```

Diese neuen Indizierungsmöglichkeiten gibt es für alle Objekte, deren Klassen das Interface `List` implementieren, sowie auch für alle Arrays.

Maps

Eine Map initialisieren Sie ebenfalls mit einer Aufzählung der Elemente in eckigen Klammern, dabei müssen aber die Schlüssel und Werte der Elemente durch einen Doppelpunkt getrennt angegeben werden. Man nennt Maps daher auch *Assoziative Arrays*. Eine leere Map wird durch einen in eckige Klammern eingeschlossenen Doppelpunkt erzeugt. Das Ergebnis ist immer eine Instanz von `HashMap`.

```
neueMap = ['alpha':'', 'beta':'', 'gamma':'', 'delta':'']
leereMap = [:]
```

Auch auf Maps können Sie wie auf Arrays zugreifen, allerdings muss hier der Schlüsselwert als Index angegeben werden:

```
buchstabeBeta = neueMap['beta']
```

Eine weitere Möglichkeit ist der Zugriff analog zu Attributen:

```
buchstabeBeta = neueMap.beta
```

Reguläre Ausdrücke

Ein Pattern-Objekt erzeugen Sie in Groovy sehr einfach mithilfe des Operators ~ aus einem String. Sie können sich dabei zunutze machen, dass in Groovy auch Strings mit einem Schrägstrich als Begrenzungszeichen definiert werden können; in diesem Fall werden die Backslash-Zeichen im String nicht interpretiert.

```
muster = ~/\w.+\w/      // neues Pattern-Objekt
```

Außerdem gibt es eine Reihe von Operatoren für reguläre Ausdrücke, beispielsweise den Muster-Vergleichsoperator ==~:

```
wort ==~ /[A-Za-z]*/    // Besteht ein Wort nur aus Buchstaben?
```

Closures

Die Closure ist ein neues Konstrukt, das einen starken Einfluss auf die Art und Weise des Programmierens ausübt. Mit ihrer Hilfe kann man ein Stück Programmcode in ein Objekt verpacken und es einer Variablen zuweisen oder als Argument einer Methode übergeben. Damit bieten Closures einen einfach zu handhabenden Ersatz für die häufigsten Anwendungsfälle von inneren Klassen, die es in Groovy selbst nicht gibt. Closures bieten aber darüber hinaus eine Fülle von Möglichkeiten und werden auch in der Groovy-Standardbibliothek sehr häufig angewendet.

Hier ist ein Beispiel für die Zuweisung einer Closure an eine Variable; im ersten Fall hat sie einen explizit benannten Parameter, im zweiten Fall wird ein implizit definierter Parameter namens `it` verwendet.

```
c1 = { x -> println x }
c2 = { println it }
```

Die Methode `each()` ist in Groovy für alle Objekte vordefiniert, die das Interface `Collection` implementieren; sie nimmt eine Closure als Argument an und ruft diese für alle enthaltenen Elemente einmal auf:

```
myCollection.each (c1)   // Gibt alle Elemente der Reihe nach aus.
```

Wenn die Closure direkt im Methodenaufruf definiert wird, verkürzt sich der Aufruf folgendermaßen:

```
myCollection.each { println it }
```

Ein typischer Anwendungsfall für Closures ist das Event-Handling, das in Java-Programmen typischerweise mittels anonymer innerer Klassen implementiert wird. Das folgende Beispiel zeigt einen Java-Ausschnitt, das einen Button mit `ActionListener` definiert.

```
// Java
JButton b = new JButton( "OK" );
b.addActionListener ( new ActionListener() {
    public void actionPerformed( ActionEvent e ) {
        System.exit(0);
    }
});
```

Mithilfe einer Closure sieht das in Groovy viel einfacher aus:

```
// Groovy
JButton b = new JButton( "OK" )
b.addActionListener ( { System.exit(0) } as ActionListener)
```

Der Abschnitt »Closures« in Kapitel 4 geht ausführlich auf die Funktionsweise von Closures und ihre Nutzung ein.

Wertebereiche (Ranges)

Groovy unterstützt einen speziellen Listentyp, mit dem Zahlenfolgen im Abstand 1 durch ihre Ober- und Untergrenzen dargestellt werden können. Es gibt ihn in einer inklusiven und einer rechts-exklusiven Variante:

```
intervall1 = 1..5     // Enthält die Werte 1,2,3,4,5.
intervall2 = 1..<5    // Enthält die Werte 1,2,3,4.
```

Ein Wertebereich implementiert das Interface `List` und kann damit überall eingesetzt werden, wo im Programm eine Liste benötigt wird, die durch den Anfangs- und den Endwert definiert ist. Das folgende Beispiel liefert den zehnten Buchstaben des Alphabets:

```
zehnterBuchstabe = ('A'..'Z')[9]
```

Ein anderes Beispiel ist die Programmierung von Zählschleifen; siehe weiter unten unter »Programmlogik«.

Sprachliche Unterschiede zwischen Groovy und Java

Trotz aller Ähnlichkeiten gibt es einige wenige Abweichungen in der Sprachsyntax zwischen Groovy und Java, die zu beachten sind. Gravierender ist eine Reihe von Unterschieden, die nicht sofort erkennbar sind, weil sie nicht die syntaktische Form der Sprache, sondern die Bedeutung dessen, was Sie programmieren, betrifft. So sieht beispielsweise die Anweisung `int i=0;` in beiden Sprachen gleich aus, aber die Bedeutung unterscheidet sich gravierend: In Groovy können Sie anschließend `i.toString()` schreiben, in Java würde dies zu einem Compilerfehler führen.

Wir wollen uns in diesem Abschnitt jene von Java bekannten Sprachelemente ansehen, die es in Groovy ebenfalls gibt, sich dort aber in der Form oder in der Funktionalität unterscheiden.

Vordefinierte Methoden

In unseren Beispielen haben Sie schon verschiedentlich gesehen, dass wir einfach die Methoden `print()` oder `println()` aufgerufen haben und nicht, wie es in Java nötig wäre, `System.out.print()` oder `System.out.println()`. Groovy kennt einen Mechanismus, mit dem vorhandene Klassen um zusätzliche Methoden erweitert werden können, und davon wird auch ausgiebig Gebrauch gemacht. Eine ganze Reihe solcher zusätzlicher Methoden

sind der Klasse `Object` zugeordnet, und dies bedeutet, dass sie an jeder Stelle des Programms ohne Qualifizierung durch einen Variablennamen verwendet werden können.

Zu diesen vordefinierten Methoden gehören unter anderen die folgenden:

- `void print (Object value)`
- `void println (Object value)`
- `void println ()`
- `void printf (String format, Object... values)`

Alle delegieren den Aufruf an die gleichnamigen Methoden in `System.out` weiter, wobei `printf()` allerdings nur zur Verfügung steht, wenn Groovy unter einer Java 5.0-JVM läuft.

Ein anderes Beispiel ist die Klasse `File`, die unter Groovy zahlreiche zusätzliche Methoden hat, die unter Java nicht zur Verfügung stehen. Darunter ist auch die Methode `getText()`, die den gesamten Inhalt der durch ein `File`-Objekt bezeichneten Datei in einen String einliest.

```
winini = new File("C:\\windows\\win.ini").getText()
```

Der Mechanismus wird in Kapitel 5 ausführlicher erklärt, eine Übersicht aller vordefinierten Methoden finden Sie in Anhang A. Dazu erhalten Sie Beispiele für die Verwendung der vordefinierten Methoden überall in diesem Buch.

Typen und Variablen

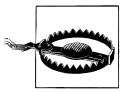
Groovy ist eine Sprache, die sowohl statische als auch dynamische Typisierung ermöglicht. Dies hat entsprechende Konsequenzen für den Umgang mit Objekten und Variablen.

Alles ist ein Objekt

Groovy arbeitet nicht wie Java mit primitiven Datentypen wie `boolean`, `char`, `byte`, `int`, `long`, `float` und `double`. Allerdings können sie einem Groovy-Programm durchaus *vorkommen*, denn Groovy muss mit anderen Java-Komponenten kooperieren können, und diese verzichten gewöhnlich nicht auf den Gebrauch von primitiven Datenelementen in ihren Schnittstellen. Groovy erlaubt also die Deklaration von Klassen-Membern und Array-Elementen sowie Methoden- und Konstruktorparameter mit primitiven Typen. Sobald Sie aber in irgendeiner Weise darauf zugreifen, wird der betreffende Wert erst einmal temporär in ein Objekt des korrespondierenden Wrapper-Typs umgewandelt, also in ein `Boolean`, `Character`, `Byte`, `Integer`, `Float` oder `Double`.

```
groovy> int i = 100
groovy> println i.getClass()
groovy> char c = 'x'
groovy> println c.getClass()
class java.lang.Integer
class java.lang.Character
```

Die primitiven Typen mögen in Java zwar ihre – vielleicht eher historisch zu sehende – Berechtigung haben, aber sie bilden dort als nicht objektorientierte Insel ein Problem, weil ständig eine besondere Behandlung dieser Daten erforderlich ist.



Etwas Vorsicht ist bei lokalen Variablen innerhalb von Methoden angebracht. Groovy erlaubt Ihnen auch dort, primitive Variablen zu definieren, verwendet aber in Wirklichkeit Objekte – und die können auch null sein. Denken Sie also daran, solche Variablen immer zu initialiseren.

```
int i
```

Die nicht initialisierte lokale Variable `i` ist in Groovy eine Instanz von `Integer` und hat den Wert `null`. Dies gilt aber nicht für nicht initialisierte Member-Variablen; diese werden auch in Groovy als primitive Werte angelegt und standardmäßig initialisiert wie in einer normalen Java-Klasse.

Bislang musste man etwa Zahlen, die man in einem Container-Objekt, zum Beispiel in einer Liste, speichern möchte, erst in Wrapper-Objekte hüllen, auf die man dann aber keine arithmetischen Operatoren anwenden kann. Ab Java 5 werden diese Manöver auch hier durch das Auto-Boxing und Auto-Unboxing automatisiert und unsichtbar im Hintergrund vorgenommen. In Groovy-Programmen entfällt dieses Hin-und-her-Umwandeln, auch wenn Sie noch mit Java 1.4 arbeiten:

```
groovy> List zahlen = new ArrayList()
groovy> zahlen.add(100)
groovy> zahlen.add(1.5)
groovy> println zahlen.[0] * zahlen.get[1]
150.0
```

Jedes Datenelement ist ein Objekt, und trotzdem sind arithmetische Operatoren anwendbar. Alle Daten können ohne Umwandlung in Container gespeichert werden, und auch sonst werden sie in konsistenter Weise behandelt. Beispielsweise ist ein Aufruf von `toString()` bei jedem beliebigen Datenelement möglich, denn es ist in jedem Fall von der Klasse `Object` abgeleitet, in der diese Methode definiert ist.

Dasselbe gilt auch für die Literale dieser Datentypen. Was Sie in Java nur mit String-Literalen machen können, nämlich direkt Methoden an ihnen aufrufen, z.B. `"abc".toUpperCase()`, geht bei Groovy auch mit Zahlen sowie mit `true` und `false` – und im begrenzten Umfang sogar mit `null`. Alle folgenden Anweisungen sind korrekt, denn wir rufen Methoden auf, die für `java.lang.Object` definiert sind und daher für alle Klassen gelten müssen.

```
groovy> Class c = 1.getClass()
groovy> String s = 1.2.toString()
groovy> Integer i = true.hashCode()
groovy> Boolean b = null.equals(s)
```

Wie bereits erwähnt, können Sie durchaus Arrays mit primitiven Typen definieren. Da Arrays auch in Java Objekte sind, ist hier für Groovy die Welt ebenfalls in Ordnung.

```
groovy> int[] i = new int[100]
groovy> i.getClass().getName()
==> [I
```

Wie Sie an dem etwas kryptischen Klassennamen »[I« erkennen können, ist `i` tatsächlich ein Array aus einfachen `int`-Zahlen. Beim Auslesen einzelner Werte aus einem solchen Array oder beim Speichern eines einzelnen Werts in ein solches Array findet eine automatische Typumwandlung statt.

```
groovy> i[0].getClass().getName()
==> java.lang.Integer
```

Der Typ des Elements mit dem Index 0 scheint `java.lang.Integer` zu sein; in Wirklichkeit ist dies aber der Wert, den das Element nach der Umwandlung annimmt. Derselbe Mechanismus wird übrigens beim Aufruf von Methoden fremder Java-Klassen angewendet, deren Schnittstellen primitive Typen umfassen (siehe unten im Abschnitt »Automatische Typanpassung«).

Dynamische Typisierung

Jede Variable in Groovy ist also, konsequenter als bei Java, eine Objektreferenz. Allerdings ist diese Typisierung von dynamischer Natur: Sie können Variablen definieren, deren Typ unbestimmt ist. Der Typ einer solchen Objektreferenz wird erst zur Laufzeit dadurch festgelegt, dass ein Objekt eines bestimmten Typs zugewiesen wird. Zur Definition einer typlosen Variablen geben Sie statt des Typnamens das Schlüsselwort `def` an.

```
groovy> def v = "Ein Text"
groovy> println v.toUpperCase()
groovy> v = 125
groovy> println v.doubleValue()
EIN TEXT
125.0
```

Wie Sie sehen, ist `v` anfangs vom Typ `String`, weil wir gleich bei der Definition einen String zugewiesen haben, und nachdem wir eine Zahl zugewiesen haben, hat `v` den Typ `Integer`. Obwohl die Variable selbst nicht mit einem Typ deklariert ist, können wir die für den jeweiligen Typ definierten Methoden `toUpperCase()` bzw. `doubleValue()` aufrufen, ohne vorher einen Typecast vornehmen zu müssen.

Das Schlüsselwort `def` können Sie einfach weglassen, wenn Sie einen Typmodifikator wie `static`, `private`, `final` usw. verwenden. Das `def` ist also immer dann erforderlich, wenn nicht klar ist, dass hier eine Variable deklariert werden soll.

```
groovy> static a = "Hallo"
```

Untypisierte Methoden definieren Sie in der gleichen Weise, also mit dem Wort `def` anstelle des Rückgabetyps. Bei den Parametern von Methoden ist das `def` nicht erforderlich, da der Compiler auch so weiß, dass hier Parameter deklariert werden.

Groovy und der Zoo

Diese Art der Typisierung wird gelegentlich als *duck typing* bezeichnet: Es sieht aus wie eine Ente, schnattert wie eine Ente und schwimmt wie eine Ente – also ist es für uns auch eine Ente. Groovy ruft eine Methode eines Objekts auf, wenn es diese in einer passenden Signatur an diesem Objekt gibt, und nicht nur, wenn die Variable entsprechend deklariert ist. Übertragen auf den Zoo heißt dies, wir betrachten ein Tier als Ente, wenn es so aussieht und sich so verhält wie eine Ente, und gehen nicht nur danach, was auf dem Schild am Gitter steht.

Der Ausspruch ist übrigens eine im englischen Sprachraum verbreitete Redewendung, die dem amerikanischen Dichter J.W. Riley zugeschrieben wird.

```
groovy> def addiere (a, b) {
groovy>     a+b
groovy> }
groovy> println addiere ("Hallo", "drian")
groovy> println addiere (100, 0.1)
groovy> println addiere (new Date(), 1)
Hallodrian
100.1
Mon Nov 06 15:52:56 CET 2006
```

In diesem Beispiel definieren wir eine Methode, die zwei Argumente jedes beliebigen Typs annimmt und ein Ergebnis liefert, dessen Typ auch nicht definiert ist. Das Beispiel selbst ist vielleicht etwas banal, aber es zeigt Ihnen, wie einfach es bei dynamischer Typisierung ist, völlig generische Methoden zu schreiben. Diese Methode addiert einfach zwei beliebige Werte, sofern bei ihnen der Plusoperator definiert ist. In allen anderen Fällen gibt es eine Exception – die wir in einem ordentlichen Programm abfangen sollten.[1]

Nicht nur die mit `def` definierten Variablen sind dynamisch typisiert. Sehen Sie sich folgendes Beispiel an:

```
groovy> List meineListe = new Stack()
groovy> meineListe.push("Hallo")
groovy> println meineListe
===> [Hallo]
```

Die Variable `meineListe` ist als `List` deklariert. Wir weisen ihr ein Objekt des Typs `Stack` zu, der das Interface `List` implementiert, und rufen dann an `s` die Methode `push()` auf, die zwar für den `Stack`, nicht aber für `List` definiert ist. Sie erkennen, dass die Variable `meineListe` dynamisch ihren Typ verändert und zu einer `Stack`-Referenz wird. In einem Java-Programm, in dem die Variablen immer nur entsprechend ihres deklarierten Typs behandelt werden, müssten wir erst einen Typecast auf `Stack` durchführen.

1 Falls Sie sich jetzt wundern, dass wir auch ein Datum und eine Integer-Zahl addieren können (die Operation addiert die entsprechende Anzahl von Tagen hinzu), müssen wir Sie bis zum Abschnitt »Operatoren« vertrösten, in dem das Überladen von Operatoren und die Erweiterungen der JDK-Bibliotheken erklärt werden.

Allerdings bleibt auch in Groovy bei typisierten Variablen die Typsicherheit gewahrt: Sie können einer Variablen nur ein Objekt zuweisen, das von ihrem Typ abgeleitet ist bzw. ihn implementiert. Folgende Anweisung ist also auch in Groovy nicht erlaubt, allerdings ist sie im Gegensatz zu Java kompilierbar und führt erst zur Laufzeit zu einer Fehlermeldung:

```
groovy> List meineListe = "ein String"
Cannot cast object 'ein String' with class 'java.lang.String' to class 'java.util.List'
```

Automatische Typanpassung

Das Nebeneinander von typisierten und untypisierten Variablen in Groovy erfordert einen Mechanismus zur automatischen Typanpassung, der im Englischen als *coercion* (Nötigung) bezeichnet wird. Wir »nötigen« einen Wert also dazu, einen bestimmten Typ anzunehmen, wenn er einer typisierten Variablen oder einem typisierten Parameter zugewiesen wird und die Typen nicht kompatibel sind. Dies erspart Ihnen nicht nur die Typecasts; in vielen Fällen brauchen Sie sich überhaupt keine Gedanken mehr darüber zu machen, mit welchen Datentypen Sie es gerade im Einzelnen zu tun haben.

```
groovy> def var1 = 10000
groovy> def var2 = 100000000000
```

Hier ist `var1` automatisch vom Typ `Integer`, dem Standardtyp für ganzzahlige Literale, und `var2` ist vom Typ `Long`, denn `Integer` reicht für eine so große Zahl nicht aus. Das macht das Leben einfacher, zumal die mathematischen Operationen in der Regel auf beide Typen gleichermaßen anwendbar sind.

Es kann aber auch komplizierter werden. Sehen Sie sich folgenden Programmausschnitt in Java an:

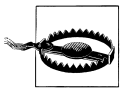
```
// Java
long longPrimitiv = 100000000000L;
int intPrimitiv = (int)longPrimitiv;     // Typecast erforderlich
Long longObjekt = Long.valueOf(100000000000L);
Integer intObjekt = (Integer)longObjekt; // Compilerfehler
```

Die Zuweisung eines `long`-Werts an eine `int`-Variable ist möglich, erfordert wegen des möglichen Informationsverlusts aber einen Typecast. Die direkte Zuweisung eines `Long`-Objekts an eine `Integer`-Variable erlaubt Java überhaupt nicht, da die eine Klasse nicht von der anderen abgeleitet ist. Hier müssten Sie `longObjekt.intValue()` anwenden, um die Zuweisung zu ermöglichen.

In einem Groovy-Programm gestaltet sich dies viel einfacher. Da können Sie Folgendes schreiben:

```
// Groovy
Long longObjekt = 100000000000L
Integer intObjekt = longObjekt           // Ohne Typecast direkt zuweisen.
```

Der Coercion-Mechanismus führt dazu, dass das Objekt der einen Klasse so gut es geht in ein Objekt der anderen Klasse umgewandelt wird.



Die automatische Typanpassung spart Tipparbeit, birgt aber auch Gefahren. In beiden Beispielen hat `intObjekt` am Ende den Wert `1215752192`, da beim Umwandeln in das kürzere Integer ein paar Bits des `Long`-Objekts verloren gehen. Im Java-Programm weist der erforderliche explizite Typecast Sie noch durch den möglichen Informationsverlust darauf hin, dass hier Gefahr droht; bei Groovy müssen Sie selbst aufpassen ...

Sie können die Typanpassung auch explizit steuern, indem Sie hinter einem Ausdruck das Schlüsselwort `as` und den gewünschten Zieltyp angeben.

```
groovy> def var = 123 as Long
groovy> var.getClass().getName()
java.lang.Long
```

Denselben Effekt haben Sie übrigens auch, wenn Sie den Typecast-Operator, z.B. `(Long)123`, anwenden, der in Groovy dieselbe Bedeutung wie `as` hat.

Natürlich lässt sich durch Coercion nicht jeder Wert in jeden beliebigen Typ umwandeln. Bei den Standardtypen weiß Groovy, welche Zuweisungen zulässig sind und wie sie durchgeführt werden müssen. In eigenen Klassen können Sie die Typumwandlung mithilfe einer Methode `asType()` selbst steuern, die Groovy aufruft, sobald eine Typumwandlung mit `as` gefordert ist; die Methode erhält den Zieltyp als Parameter übergeben und liefert ein Objekt dieses (oder eines davon abgeleiteten) Typs zurück.

Standardmäßige Sichtbarkeit von Klassen und Membern

Groovy verwendet die gleichen Schlüsselwörter für die Sichtbarkeit wie Java: `public`, `protected`, `private`, und deren Bedeutung unterscheidet sich prinzipiell nicht von Java. Ein wesentlicher Unterschied besteht in dem Fall, dass überhaupt kein Modifikator für die Sichtbarkeit angegeben ist.

- Die standardmäßige Sichtbarkeit für *Klassen* und *Methoden* ist `public`. Dieses Schlüsselwort muss also vor öffentlichen Klassen- und Methodendefinitionen nicht angegeben werden. Die Package-Sichtbarkeit, wie sie in Java beim Fehlen eines Sichtbarkeitsmodifikators gilt, wird von Groovy nicht unterstützt; stattdessen können Sie aber `protected` verwenden.
- Die Sichtbarkeit von Feldern innerhalb von Klassen ist `private`. Allerdings generiert Groovy eigenständig öffentliche Getter- und Setter-Methoden, sofern diese nicht explizit programmiert worden sind. Felder ohne Sichtbarkeitsmodifikator werden also automatisch zu Properties im Sinne der JavaBean-Spezifikation. Mehr dazu weiter unten in Kapitel 3.

Wir müssen der Ordnung halber darauf hinweisen, dass Groovy die Privatsphäre von Objekten zurzeit wenig achtet. Das merken Sie, wenn Sie beispielsweise einmal dies hier eingeben:

```
groovy> println "abc".value
[a, b, c]
```

Wir greifen hier munter auf das private Feld `value` des String-Objekts zu, ohne dass Groovy uns daran in irgendeiner Weise hindert. Man kann annehmen, dass dieses Verhalten in absehbarer Zeit geändert wird, derzeit müssen Sie jedoch selbst auf die Kapselung von Klasseninterna achten.[2]

Wenn für Variablen kein Typ festgelegt werden soll, kann die Variable einfach mit dem Schlüsselwort `def` definiert werden. Dies entspricht in Java einer Deklaration vom Typ Objekt. Innerhalb von Skripten kann auf das Schlüsselwort `def` verzichtet werden. Dies macht einen kleinen, subitlen Unterschied bezüglich des Variablen-Bindings, der hier vernachlässigt werden kann, auf den aber im nächsten Kapitel (im Abschnitt »Skripte und das Binding«) eingegangen werden wird.

Elementare Datentypen

Groovy arbeitet mit elementaren Datentypen für Zahlen und Texte etwas anders als Java. Die Unterschiede fallen nicht unbedingt gleich auf, führen aber dazu, dass Groovy-Programme unter Umständen andere Ergebnisse haben können als gleich aussehende Java-Programme.

Brüche und große Zahlen

Wenn eine Zahl (Literal oder Rechenergebnis) nicht mehr durch ein `Integer` oder `Long` abbildbar ist, weil sie zu groß oder weil sie nicht ganzzahlig ist, bedient sich Groovy stattdessen automatisch der Typen `BigInteger` oder `BigDecimal`. Dadurch können Rechenergebnisse geringfügig von den Werten abweichen, die man in Java bei Anwendung derselben Formeln erhält. Sie können aber auch wie gewohnt mit `Float` und `Double` rechnen, nur müssen Sie die Literale entsprechend mit den Kennbuchstaben `f/F` oder `d/D` kennzeichnen.

Die `BigInteger`- und `BigDecimal`-Objekte können Sie in arithmetischen Formeln einsetzen wie andere Zahlen auch.

2 Wir beziehen uns hier auf den aktuellen Codestand bei Redaktionsschluss des Buchs. Der Hintergrund dafür, dass dieser Fehler noch nicht beseitigt worden ist, besteht darin, dass existierende Testprogramme gern die Möglichkeit des Zugriffs auf Klasseninterna für Whitebox-Tests nutzen. Vermutlich wird es in Zukunft einen Schalter geben, mit dem das Verhalten von Groovy gegenüber privaten Member-Variablen und -Methoden gesteuert werden kann.
Allerdings müssen wir damit rechnen, dass als `protected` deklarierte Member weiterhin wie öffentlich behandelt werden, da die Prüfung des Zugriffsrechts wegen der dynamischen Eigenschaften von Groovy zu aufwendig ist.

```
groovy> println 1.1 + 1.7
2.8
groovy> println ((1.1 + 1.7).getClass())
class java.math.BigDecimal
```

Sie sehen, dass hier die Addition von 1.1 und 1.7 ein exaktes Ergebnis liefert, da die Berechnung mit `BigDecimal` ausgeführt wird. In kommerziellen Zusammenhängen ist dies im Allgemeinen angemessener, aber für technisch-wissenschaftliche Berechnungen können Sie nach wie vor auch die bekannten Fließkommatypen benutzen, indem Sie die Variablen entsprechend deklarieren und die Konstanten mit den Buchstaben `d` oder `f` kennzeichnen. Hier ist die gleiche Berechnung mit `Double`-Zahlen:

```
groovy> println 1.3d + 0.4d
1.7000000000000002
groovy> println ((1.3d + 0.4d).getClass())
class java.lang.Double
```

Dem Thema Zahlen und Arithmetik widmen wir uns noch ausführlich in Kapitel 5.

Strings und Zeichen

Normale String-Literale werden in Groovy in einfache oder doppelte Hochkommata eingeschlossen, also etwa so:

```
String s1 = 'Dies ist ein String-Literal'
String s2 = "Dies ist ein String-Literal"
```

Wenn Sie doppelte Anführungszeichen verwenden, können Sie im String interpolieren, das heißt beliebige Groovy-Variablen oder -Ausdrücke einfügen; sie werden dann zu »GString«-Objekten (mehr dazu in Kapitel 3).

```
String s3 = "Die Zeit ist: ${new Date()}"
```

Groovy kennt auch String-Literale, die über mehrere Zeilen reichen. Schließen Sie diese auf beiden Seiten durch drei einfache Anführungszeichen ab.

```
String s = '''Dies ist ein
   besonders langer String.'''
```

Dieses Literal ist gleichwertig zu `'Dies ist ein\n   besonders langer String'`. Beachten Sie, dass der Zeilenumbruch unabhängig von der Systemplattform immer in ein `\n`-Zeichen übersetzt wird und dass die Leerzeichen vor und hinter den Zeilenumbrüchen erhalten bleiben.

In Java dient das einfache Hochkomma als spezielles Begrenzungszeichen für `char`-Werte. In Groovy wird es nicht benötigt, verwenden Sie einfach String-Literale mit einem Zeichen; die Umwandlung nimmt Groovy bei Bedarf automatisch vor.

Operatoren

Die Operatoren lassen sich in Groovy größtenteils in der gleichen Weise nutzen wie in Java. Allerdings gibt es in Groovy einige zusätzliche Operatoren, und die von Java bekannten Operatoren können in vielen weiteren Zusammenhängen oder in anderer Weise verwendet werden. Beispielsweise dienen die eckigen Klammern in Java nur der Indizierung von Arrays; in Groovy können damit auch Listen und Maps indiziert werden, und es können sogar Indexlisten und negative Indizes angegeben werden.

Überladen durch spezifische Methoden

Der entscheidende semantische Unterschied besteht darin, dass Groovy die meisten Operatoren nicht direkt in den korrespondierenden Bytecode übersetzt, sondern in Methodenaufrufe. So sind in Groovy beispielsweise die folgenden beiden Ausdrücke vollständig äquivalent:

```
x + 1
x.plus(1)
```

Dies ermöglicht Ihnen, jede beliebige Klasse mit Operatoren zu versehen; sie brauchen nur die entsprechenden Methoden zu implementieren.

Die Groovy-Standardbibliotheken machen extensiv Gebrauch von dieser Möglichkeit, indem sie vorhandenen Klassen und Interfaces die zu den Operatoren gehörenden Methoden als vordefinierte Methoden zuordnet. So ist beispielsweise die obige Addition mit Integer-Zahlen in Groovy-Programmen nur dadurch möglich, dass der Klasse `Integer` die vordefinierte Methode `plus()` zugeordnet ist.

Dies beschränkt sich in Groovy aber keinesfalls auf die Typen, für die die Operatoren in Java definiert sind. Alle folgenden Beispiele beruhen auf vordefinierten Methoden:

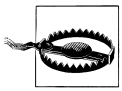
```
wert = meineMap['schlüssel']  // Element einer Map lesen
meineMap['schlüssel'] = wert  // Element einer Map schreiben
System.out << 'daten'         // Schreiben in einen PrintStream
morgen = new Date() + 1       // Anzahl Tage zu einem Date-Objekt zählen
```

In Kapitel 3 finden Sie weitere Erläuterungen und eine vollständige Übersicht aller überladbaren Operatoren und der zugehörigen Methoden.

Gleichheit und Identität

Die Implementierung von Operatoren über Methoden gilt auch für das doppelte Gleichheitszeichen (`==`). Groovy bildet es auf einen Aufruf der Methode `equals()` beim linken Operanden ab. Daraus folgt, dass der Operator `==` in Groovy prüft, ob zwei Objekte den gleichen Wert haben, während er in Java prüft, ob zwei Objekte identisch sind.

Wenn Sie möchten, können Sie übrigens auch jedes Objekt mit `null` vergleichen; ein Vergleich mit `null` ergibt immer `false`, einzig `null==null` ergibt `true`.



Die Abbildung des Gleichheitsoperators auf eine Methode eines der Operanden führt dazu, dass die Ausdrücke `x==y` und `y==x` nicht unbedingt dasselbe Ergebnis liefern. Diese Gefahr ist ganz real, wenn man den Fall bedenkt, dass `x` und `y` Objekte zweier verschiedener Klassen sind und dass die Klasse von `x` möglicherweise den Typ von `y` kennt, dies umgekehrt aber nicht unbedingt zutrifft. Allerdings sind die Klassen der Groovy-Standardbibliothek nach festgelegten Konventionen programmiert, die sicherstellen, dass die Ergebnisse von Vergleichsoperationen in der Regel nicht von der Reihenfolge der Operanden abhängig sind.

Wenn Sie doch einmal die *Identität* zweier Objekte prüfen möchten, können Sie die Methode `is()` verwenden. Sie ist für jede Klasse definiert, akzeptiert jedes Argument und liefert tatsächlich nur dann `true`, wenn das übergebene Objekt identisch mit dem aktuellen Objekt ist.

Somit können Sie also davon ausgehen, dass, obwohl auch Zahlen immer Objekte sind, der Ausdruck

```
123==123
```

immer `true` liefert, während dies bei

```
123.is(123)
```

nicht unbedingt der Fall ist, denn die beiden `Integer`-Objekte mit demselben Wert können durchaus verschieden sein.

Sie wundern sich vielleicht über die Behauptung, dass die Methode `is()` für jede Klasse definiert ist. Ein Blick ins JavaDoc zu J2SE zeigt schließlich, dass keine einzige Klasse der Standard-APIs eine Methode dieses Namens hat. Trotzdem funktioniert so etwas wie:

```
new Object().is(125)
```

Tatsächlich ist die Methode `is()` genau wie `println()` eine Groovy-eigene Erweiterung der Java-Standardbibliothek, die für alle Instanzen aller Klassen verfügbar ist.

Letztendlich ist der Gleichheitsoperator `==` in Groovy konsistenter definiert als in Java, da er *immer* auf Gleichheit prüft. In Java hingegen prüft er nämlich bei Basisdatentypen wie `int`, `char`, `double` etc. auf Gleichheit, während er bei Objektreferenzen auf Gleichheit der Referenz, also auf Objektidentität prüft.

Programmlogik

Während sich die Groovy-Befehle für Schleifen und Verzweigungen – abgesehen von der `for`-Anweisung – syntaktisch nicht wesentlich unterscheiden, können sie sich trotzdem unterschiedlich verhalten, da sich der Umgang mit der Wahrheit deutlich bei Groovy und Java unterscheidet.

Logische Ausdrücke

Logische Prüfungen, wie sie in `if`, `while`, `assert` usw. vorgenommen werden, erwarten anders als bei Java nicht unbedingt einen Booleschen Wert:

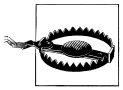
```
int i = 1
String s = ''
assert i
if (s) { println 'ausgeführt'; }
```

In diesem Beispiel läuft die Assert-Anweisung erfolgreich durch, und die Anweisung im `if`-Block wird nicht ausgeführt. Bei Java hätte Ihnen schon der Compiler zwei Fehler angezeigt, da die zu prüfenden Ausdrücke in `assert` und `if` einen Booleschen Wert ergeben müssen, was hier nicht der Fall ist.

Wenn Groovy die Wahrheit ermitteln will, prüft es in der folgenden Reihenfolge:

1. Es ist ein Boolescher Wert und hat den Wert `true`.
2. Es ist das Ergebnis eines erfolgreichen Pattern-Match.
3. Es ist eine `Collection` oder eine `Map` und ist nicht leer.
4. Es ist ein `String` und ist nicht leer.
5. Es ist eine Zahl und ist ungleich `0`.
6. Es ist keines dieser Typen und ist nicht `null`.

Das bedeutet beispielsweise, dass ein Objekt normalerweise als `true` gilt, sofern es nicht `null` ist. Sobald es aber – und sei es auf Umwegen, so dass Sie es ihm nicht ohne Weiteres ansehen – ein von `Collection` oder `Map` abgeleitetes Interface implementiert und der Aufruf von `size()` den Wert 0 ergibt, ist es `false`. Vorsicht ist also angesagt.



Eine unangenehme Nebenwirkung dieses flexibleren Umgangs mit der Wahrheit besteht darin, dass eine Verwechslung von einfachem und doppeltem Gleichheitszeichen nicht mehr auffällt. Eine Abfrage wie

```
while (x = 1) { ... }
```

bei der eigentlich geprüft werden sollte, ob `x` gleich `1` ist, würde in Java sofort zu einem Compilerfehler führen, da die Zuweisung eines numerischen Werts kein Boolesches Ergebnis haben kann. Bei `if`- und `assert`-Anweisungen ist dies zwar kein Problem, da dort die logischen Bedingungen nicht aus einer Zuweisung bestehen dürfen, bei `while` findet eine solche Prüfung jedoch nicht statt. Die Folge ist: Obige Anweisung führt zu einer Endlosschleife, da die Zuweisung `1` ergibt, und das ist gleichwertig mit `true`.

Die switch-Verzweigung

Die `switch`-Anweisung hat zwar die gleiche Form wie in Java – auch hier darf man das lästige `break` nach jedem Zweig nicht vergessen –, bietet aber viel mehr Möglichkeiten. Grundsätzlich ist `switch` nicht auf ganzzahlige Werte beschränkt, was mangels primitiver

Typen auch keinen Sinn ergäbe, und es kann auf alles Mögliche getestet werden. Sehen Sie sich dieses Beispiel an:

```
switch (meineVariable) {
    case 100:                          // Integer-Zahl
        println "Die Zahl 100"
        break
    case "ABC":                        // String
        println "Der String ABC"
        break
    case Long:                         // Klasse
        println "Ein Long-Wert"
        break
    case ['alpha','beta','gamma']:     // Liste
        println "alpha, beta oder gamma"
        break
    case {it > -0.1 && it < 0.1}:      // Closure
        println "Eine Zahl nahe Null"
        break
    case null:                          // null
        println "Ein leerer Wert"
        break
    case ~/Groov.*/:                    // Regulärer Ausdruck
        println "Beginnt mit Groov"
        break
    default:
        println "Etwas ganz anderes"
}
```

In diesem Beispiel wird der `switch`-Wert gegen sieben völlig verschiedene Dinge geprüft. Es sind Objekte – was kein Wunder ist, da es ja in Groovy keine einzeln stehenden primitiven Daten gibt –, es können durchaus Variablen sein und nicht nur Konstanten, und es sind sogar ein Klassenname, eine Closure und ein regulärer Ausdruck dabei. Tatsächlich können Sie in einem Groovy-Switch neben Klassennamen gegen jeden beliebigen Wert prüfen. Dabei wird normalerweise auf Gleichheit geprüft, in einigen Fällen führt der `case`-Ausdruck aber eine abweichende Prüfung durch:

- Eine `Collection` oder ein Array prüft, ob das Objekt darin enthalten ist.
- Ein `Pattern` prüft, ob ein `toString()` von dem Objekt mit dem Muster übereinstimmt.
- Eine `Closure` prüft, ob ihr Aufruf mit dem Objekt als Argument das Ergebnis `true` ergibt.
- Eine `Class` prüft, ob das Objekt eine Instanz der Klasse ist oder das Interface implementiert.

Um den passenden `case`-Zweig zu ermitteln, wird nämlich die `isCase()`-Methode aufgerufen. Damit ist es auch möglich, eigene Klassen zu definieren, die geschickt in einer `switch`-Anweisung verwendet werden können.

Asserts

Das `assert`-Statement hat eine etwas andere Syntax als in Java: Wenn Sie einen erläuternden Text hinzufügen wollen, trennen Sie ihn von dem zu prüfenden Ausdruck durch ein Komma ab und nicht wie in Java durch einen Doppelpunkt:

```
assert c=='Z', 'Der Wert von c sollte Z sein'
```

Wie in Java wird bei einem Fehlschlagen der Prüfung ein `java.lang.AssertionError` ausgelöst. Der Unterschied besteht darin, dass die Auswertung von Asserts in Groovy weder aus- noch eingeschaltet werden kann und immer aktiv ist. Der Grund dafür besteht in dem dynamischen Charakter von Groovy: Der Compiler kann in Groovy viel weniger Fehler finden als in Java, daher ist es ratsam, mithilfe zahlreicher Assert-Anweisungen dafür zu sorgen, dass Fehler zur Laufzeit schnell gefunden werden können, und diese Möglichkeit auch nicht zu deaktivieren.

Dynamische Methoden

Wenn Sie den Aufruf einer Objektmethode programmieren, z.B.

```
meinObjekt = new MeineKlasse()
meinObjekt.tuDies()
```

können Sie sich in Java darauf verlassen, dass eben diese Methode ihres Objekts aufgerufen wird, sei es, dass sie vom Objekt implementiert oder von einer übergeordneten Klasse geerbt ist. Wenn dies nicht möglich ist, erhalten Sie bereits beim Kompilieren einen Fehler. In Groovy können da ganz verschiedene Dinge passieren:

- Die Klasse `MeineKlasse` kann eine Methode namens `tuDies()` haben oder erben, die – wie in Java – direkt aufgerufen wird.
- Das Objekt kann eine Property namens `tuDies` haben, der eine Closure zugewiesen worden ist. Dann wird diese aufgerufen.
- Die Methode `tuDies()` könnte von der Groovy-Laufzeitbibliothek für die Klasse `MeineKlasse` oder eine übergeordnete Klasse oder ein von ihr implementiertes Interface vordefiniert sein. Dann wird die vordefinierte Methode aufgerufen.
- Sie können eine sogenannte Kategorienklasse definiert und aktiviert haben, die eine Methode namens `tuDies()` für `MeineKlasse` oder eine übergeordnete Klasse oder ein von ihr implementiertes Interface enthält. Dann wird diese aufgerufen.
- Die Klasse `MeineKlasse` kann die Methode `invokeMethod()` überschreiben und somit dafür sorgen, dass beim vermeintlichen Methodenaufruf etwas ganz anderes geschieht.
- Sie können dem Objekt `meinObjekt` eine bestimmte Metaklasse zuweisen und auf diese Weise ebenfalls dafür sorgen, dass etwas ganz anderes geschieht.

Sie werden diese verschiedenen Möglichkeiten der Beeinflussung des Objektverhaltens und die Möglichkeiten, die sich daraus ergeben, im Laufe der Lektüre dieses Buchs im

Einzelnen kennenlernen. An dieser Stelle soll der Hinweis genügen, dass in Groovy-Programmen Schein und Sein ziemlich weit auseinanderliegen können.

Sonstige Abweichungen gegenüber Java 1.4

Es gibt noch einige weitere Unterschiede zwischen Groovy und Java, die von untergeordneter Bedeutung sind, aber hier der Vollständigkeit halber aufgeführt werden.

Klassen und Quelldateien

Anders als in Java können in einer Groovy-Quelldatei mehrere öffentliche Klassen definiert sein, und – was zwangsläufig daraus folgt – die Namen der Klassen müssen nicht unbedingt mit den Dateinamen übereinstimmen. Beim Übersetzen mit `groovyc` entsteht für jede Klasse eine eigene *.class*-Datei, die den Namen der Klasse hat.

Innere Klassen

Sie können in Groovy keine Klassen innerhalb von anderen Klassen und damit auch keine anonymen Klassen definieren. Sie werden sie auch kaum vermissen, da der am häufigsten vorkommende Anwendungsfall von inneren Klassen, nämlich als Event-Handler zu dienen, in Groovy viel besser durch Closures erfüllt wird (siehe Kapitel 4).

Klassennamen

Der Name einer Klasse ist in Groovy immer auch zugleich eine Referenz auf das entsprechende `Class`-Objekt. Wenn Sie in Java beispielsweise `String.class.getName()` schreiben müssen, reicht in Groovy `String.name`.

Statisches this

In Groovy-Programmen können Sie das reservierte Wort `this` auch im statischen Kontext verwenden. In diesem Fall referenziert es nicht auf die aktuelle Instanz (die es ja nicht gibt), sondern auf die Instanz des aktuellen Klassenobjekts. Das `assert` in der folgenden Main-Methode ist also erfolgreich.

```
Class EineKlasse {
    static void (args) {
        assert this == EineKlasse
    }
}
```

Array-Initialisierung

Groovy kennt keine Array-Initialisierung, wie sie in Java mit Listen in geschweiften Klassen möglich ist:

```
// Java
int[] i = new int[] {1,2,3,4,5};
```

Stattdessen können Sie in Groovy Arrays mithilfe von Listen-Literalen passenden Typs initialisieren. Wenn die zu initialisierende Variable explizit typisiert ist, wird der Typ der Liste automatisch angepasst, andernfalls muss die Typanpassung mit dem Schlüsselwort `as` erzwungen werden.

```
int[] i = [1,2,3,4,5]           // mit impliziter Typanpassung
j = [1,2,3,4,5] as int[]        // mit expliziter Typanpassung
```

Groovy und Java ≥ 5.0

Groovy baut zwar auf Java 1.4 auf, aber wenn Sie J2SE 5.0 oder höher einsetzen, können Sie die Laufzeitbibliotheken dieser Java-Version problemlos auch mit Groovy einsetzen. Das gilt ebenfalls für Klassen, die neue Java-Fähigkeiten wie Typparameter und variable Argumentlisten ausnutzen. In der Groovy-Programmierung selbst können Sie zwar einige der neuen Möglichkeiten von Java verwenden, allerdings funktioniert dies nur eingeschränkt, da Groovy derzeit ja noch mit Java 1.4 laufen soll. Schließlich ist es in der Praxis immer noch recht üblich, dass im Zuge einer vereinheitlichten IT-Landschaft häufig eine bestimmte und manchmal recht alte JDK-Version vorgeschrieben wird. Es ist aber geplant, dass Groovy ab der Version 2.0 nur noch mit Java 5.0 oder höher laufen soll, so dass die Sprache dann noch besser an das neuere Java angepasst werden kann. Im Einzelnen kann derzeit Folgendes gesagt werden:

- *Generische Datentypen* gibt es in Groovy zwar ab der Version 1.1 sowohl bei der Instantiierung von typisierten Klassen als auch bei der Definition typisierter Klassen, allerdings wird die Typisierung beim Aufruf der Objekte nicht geprüft, d.h., Sie können beispielsweise einer mit `new ArrayList<String>{}` instantiierten String-Liste ohne Weiteres irgendein anderes Objekt zuweisen. Dadurch kann Groovy besser mit Java-Frameworks und Klassenbibliotheken zusammenarbeiten, die aus den generischen Typangaben Informationen ableiten.
- *Annotationen* können in Groovy ab Version 1.1 angewendet werden, sofern Groovy unter Java ab Version 1.5 läuft, und werden dann auch korrekt verarbeitet. Es gibt aber noch keine Möglichkeit, eigene Annotationen zu definieren.
- *Die »verbesserte« `for`-Schleife* gibt es in Groovy schon länger als in Java, allerdings in einer leicht veränderten Form mit dem Schlüsselwort `in` an der Stelle, an der bei Java der Doppelpunkt steht. Kurz vor Fertigstellung der Groovy-Version 1.0 hat man sich allerdings entschlossen, die Form mit Doppelpunkt auch in Groovy zu ermöglichen. Während in Java nur diese Formulierung möglich ist:

  ```
  for (String s : stringList) {...} // Java und Groovy
  ```

  können Sie in Groovy auch das schreiben

  ```
  for (String s in stringList) {...} // nur Groovy
  ```

  Die zweite Form ist in Groovy insofern etwas passender, weil `in` hier ein allgemeiner Operator mit der Bedeutung »enthalten in« ist; außerdem brauchen Sie in dieser

Variante die Laufvariable nicht zu deklarieren, `for (s in stringList)` würde es also auch tun.

- *Das Auto-Boxing und Auto-Unboxing* nimmt Groovy genau wie Java 5 automatisch vor, sobald eine Typumwandlung zwischen einem primitiven Datentyp (z.B. `int`) und dem korrespondierenden Wrapper-Typ (z.B. `Integer`) erforderlich ist.
- *Variable Argumentlisten*, die in Java durch Formalparameter in der Form *typ*`...`*name* dargestellt werden, kennt Groovy auch schon länger, jedoch muss hier der letzte Parameter einfach ein Array sein. Näheres hierzu in Kapitel 3.
- *Das typsichere* `Enum` kann in Groovy problemlos benutzt werden. Eigene `Enum`-Typen zu definieren ist ab Groovy 1.1 möglich, sofern es unter Java ab Version 5.0 läuft. Ganz so groß wie in Java ist der Nutzen allerdings auch in diesem Fall nicht, da die Typsicherheit in Groovy eine etwas geringere Rolle spielt und `switch`-Statements – in denen `Enum`-Objekte besonders hilfreich sind – in Groovy ohnehin sehr viel flexibler angewendet werden können.
- *Statische Imports* unterstützt Groovy ab Version 1.1, und zwar auch dann, wenn es noch unter Java 1.4 läuft.

Sie kennen nun die wesentlichen Dinge, die Groovy und Java hinsichtlich der Programmiersprache unterscheiden. Das sollte Sie schon jetzt in die Lage versetzen, mithilfe Ihrer vorhandenen Java-Kenntnisse die ersten Probleme auch schon mit Groovy – und etwas weniger Tipparbeit – lösen zu können. Sie haben damit aber auch das notwendige Verständnis für die folgenden Kapitel, in denen wir noch etwas tiefer in die Möglichkeiten einsteigen, die Groovy Ihnen beim Programmieren bietet.

KAPITEL 3

Objekte in Groovy

Nachdem wir uns im letzten Kapitel erst einmal mit den vielen kleinen sprachlichen Unterschieden zwischen Groovy und Java beschäftigen mussten, kommen wir nun zu den Dingen, die wirklich neu sind. Gegenstand dieses Kapitels sind Klassen in Groovy: Wie werden sie definiert, und wie werden sie verwendet, welche Besonderheiten gibt es bei Skripten? Objekte sollen in diesem Kapitel noch weitgehend statisch betrachtet werden – sie verhalten sich im Wesentlichen so, wie es die Klasse vorschreibt, die sie implementieren, genau wie wir es von Java gewohnt sind. Auf die Besonderheiten der dynamischen Programmierung von Objekten kommen wir dann später in Kapitel 7 zurück.

Objekte in Groovy

Auf den ersten Blick unterscheiden sich Groovy-Objekte nicht wesentlich von Objekten in Java: Jedes Objekt ist Instanz einer Klasse, in der die Konstruktoren, die Methoden und die Felder festgelegt sind. Solche Groovy-Objekte können auch von Java-Programmen aus verwendet werden, und sie stellen sich in diesem Zusammenhang – zumindest auf den ersten Blick – kaum anders dar, als wären sie ebenfalls in Java programmiert.

Sie sind aber nicht in Java programmiert, sondern in einer Programmiersprache, die sich Einfachheit und Flexibilität auf die Fahnen geschrieben hat. Und das kann man sich als Groovy-Programmierer zunutze machen.

Wenn Sie in Groovy ein Programm schreiben, müssen Sie zunächst unterscheiden, ob Sie dies in Form einer Klasse oder eines Skripts tun. Skripte sind einfach zu erstellen, weil Sie weder Klasse noch `main()`-Methode explizit definieren müssen und einige Freiheiten im Aufbau des Programms haben. Daher bieten sie sich vor allem für kleine, einfache Aufgaben an. Auch wenn Sie Groovy-Programme dynamisch in Java-Anwendungen integrieren, ist das Groovy-Skript dafür die geeignete Form. Auf die Besonderheiten des Skriptschreibens gehen wir weiter unten unter »Skriptobjekte« noch ausführlicher ein; zunächst ein-

mal kommen wir auf die eher gewohnte Form der Groovy-Klassen zu sprechen, die sicher vorzuziehen ist, sobald es komplexer wird oder Sie Programmteile wiederverwenden möchten.

Package-Struktur

Genau wie Java-Klassen werden auch Groovy-Klassen in Packages organisiert. Der Name des Package muss am Anfang der Quelldatei vor den Importanweisungen und der ersten Klassendefinition genannt sein. Dabei können Sie die gewohnte Form der Strukturierung anhand von Domainnamen in umgekehrter Reihenfolge verwenden, aber niemand zwingt Sie dazu.

```
package de.oreilly.groovy.beispiele.kap3
```

Wenn Sie in Ihrem Programm Klassen oder Interfaces aus anderen Packages referenzieren, müssen Sie entweder an Ort und Stelle den vollen Pfadnamen verwenden oder sie vor dem Beginn Ihrer Klassendefinition importieren. Sie können auch alle Klassen und Interfaces eines Package *en bloc* importieren, indem Sie anstelle des Klassennamens einen Stern einsetzen. Dabei macht es keinen Unterschied, ob die importierte Klasse eine Java- oder Groovy-Klasse ist.

```
import groovy.inspect.swingui.TableSorter
import javax.swing.*
```

Auch statische Importe sind erlaubt, und dies sogar, wenn Sie noch mit Java 1.4 arbeiten.

```
import static java.lang.Math.*
```

So weit nicht viel Neues. Anders als Java brauchen Sie in Groovy-Programmen jedoch die am häufigsten verwendeten Packages nicht eigens zu importieren, da Groovy dies schon für Sie vornimmt. Die folgenden Importanweisungen sind also alle überflüssig:

```
import java.lang.*   // Dieses Package wird auch von Java importiert.
import java.util.*
import java.net.*
import java.io.*
import java.math.BigInteger
import java.math.BigDecimal
import groovy.lang.*
import groovy.util.*
```

Außerdem ermöglicht Ihnen Groovy, Aliasnamen für importierte Klassen und Interfaces zu vergeben. Dadurch können Sie zwischen Typen mit gleich lautenden Namen unterscheiden oder lange Typnamen abkürzen.

```
import java.awt.ContainerOrderFocusTraversalPolicy as COFTP
import java.sql.Date as SqlDate
import javax.naming.Binding as JndiBinding

COFTP myPolicy = new COFTP()
def myDate = new SqlDate(106,11,3)
JndiBinding binding = new JndiBinding("Date",myDate)
```

Am Beispiel `JndiBinding` erkennen Sie, dass die Aliasnamen in Groovy auch wegen der vielen automatisch importierten Packages hilfreich sein können, da die Gefahr von Namenskonflikten viel größer ist als in Java. Eine Klasse namens `Binding` gibt es auch schon in dem Package `groovy.lang`, das von Groovy automatisch importiert wird. Durch ein normales `import` von `javax.naming.Binding` hätten Sie also zwei Klassen mit demselben einfachen Namen; die `as`-Klausel mit dem Aliasnamen hilft Ihnen, diesen Konflikt zu vermeiden.

Die Pfadstruktur der Packages muss sich genau wie in Java in der Verzeichnisstruktur widerspiegeln. Das heißt, die Quelldatei einer Groovy-Klasse im Package `de.groovy-grails.buch.beispiele.kap3` muss – relativ zu einem Verzeichnis im Klassenpfad – im Verzeichnis *de/groovy-grails/buch/beispiele/kap3* zu finden sein.

Klassenpfade

Wenn Sie Ihre Groovy-Programme mit `groovyc` in Bytecode übersetzen und mit `java` oder `javaw` ausführen, gilt bezüglich der Klassenpfade dasselbe wie für jedes beliebige Java-Programm: Alle benötigten *.class*-Dateien müssen in den im Klassenpfad aufgeführten Verzeichnissen und JAR-Dateien – unter Berücksichtigung der Package-Struktur – auffindbar sein. Der Klassenpfad setzt sich aus den JDK- und Extensions-Bibliotheken, dem Inhalt der Umgebungsvariablen `CLASSPATH` und den Argumenten des `java`-Aufrufparameters `-classpath` bzw. `-cp` zusammen. Sie müssen nur darauf achten, dass die von Groovy benötigten Bibliotheken (am besten *groovy-all-….jar*) in diesem Klassenpfad aufgeführt sind.

Wenn Sie ein Groovy-Programm mit dem Befehl `groovy` starten, sieht es etwas anders aus. Hinter dem Namen verbirgt sich ein Shell-Skript, das schon einiges für Sie erledigt, bevor es die Groovy-Hauptklasse `groovy.ui.GroovyMain` aufruft; unter anderem baut es schon einen Klassenpfad mit benötigten Bibliotheken zusammen. Wenn Sie `groovy` mit einem Klassennamen aufrufen, sucht es diese Klasse nicht in einer Klassendatei (*.class*), sondern in einer Quelldatei (*.groovy*), und dieser muss im Klassenpfad zu finden sein. Das Gleiche gilt, wenn Ihr Groovy-Programm andere Klassen referenziert, diese können normale *.class*-Dateien sein, die in der üblichen Weise lokalisiert werden, es können aber auch wiederum Groovy-Quelldateien sein, die über den Klassenpfad auffindbar sein müssen.

Da beim Aufruf von `groovy` letztendlich die virtuelle Maschine von Java gestartet wird, gelten im Prinzip die gleichen Pfadangaben wie oben genannt. Allerdings fügt Groovy noch einige weitere hinzu, die in der Konfigurationsdatei *%GROOVY_HOME%/conf/groovy-starter.conf* aufgeführt sind. Unter anderem finden Sie dort auch diese Zeilen:

```
# load user specific libraries
load ${user.home}/.groovy/lib/*
```

Dies bedeutet, dass alle Klassen-, JAR- und Groovy-Dateien, die Sie in das Verzeichnis *.groovy/lib/* unterhalb Ihres Benutzerverzeichnisses legen, für alle Groovy-Programme, die Sie ausführen, ohne explizite Benennung im Klassenpfad verfügbar sind.

Wenn Sie nicht genau wissen, wo Ihr Benutzerverzeichnis liegt, brauchen Sie nur in `groovysh` kurz

```
groovy> System.properties.get('user.home')
```

einzugeben.

Die Möglichkeit, den Klassenpfad auf diese Weise zu erweitern, ist natürlich hilfreich, wenn Sie häufig andere als die standardmäßig eingebundenen Bibliotheken verwenden und diese nicht bei jedem Skriptaufruf mit angeben möchten. Die Personalisierung des Klassenpfads kann aber auch dazu führen, dass Programme bei Ihnen anders funktionieren als bei anderen oder dass bei von Ihnen geschriebene Programme bei Ihnen laufen und bei anderen nicht. Eine gewisse Vereinheitlichung von gemeinsam genutzten Bibliotheken können Sie in einem Firmennetzwerk dadurch erreichen, dass Sie ein Verzeichnis für gemeinsam genutzte Bibliotheken auf einem Fileserver einrichten und in *groovy-starter.conf* auf dieses verweisen.

Als Skript (also als *.groovy*-Datei) aufgerufene Programme werden bisweilen unhandlich, wenn Sie vor der Verwendung erst dafür sorgen müssen, dass der Klassenpfad richtig gesetzt wird. Es ist aber durchaus möglich, in einem Groovy-Programm den Klassenpfad dynamisch zu erweitern.

Angenommen, Sie möchten ein Skript schreiben, das eine E-Mail versendet, und dazu die Klasse `SimpleEmail` in der Open Source-Bibliothek *commons-email-1.jar* verwenden. Dabei soll aber nicht erst vor Aufruf des Skripts der Klassenpfad gesetzt werden müssen. Um dies zu bewerkstelligen, müssen Sie sich zuerst den aktuellen Rootloader holen; dies ist eine spezielle, Groovy-eigene Implementierung des `ClassLoader`, dessen Klassenpfad zur Laufzeit erweiterbar ist.

```
def rootLoader = this.class.classLoader.rootLoader
assert rootLoader!=null
```

In normalen, mit `groovy` gestarteten Skripten oder Programmen ist dieser Rootloader immer vorhanden; wenn Groovy-Klassen oder -Skripte innerhalb von Java-Programmen verwendet werden, ist dies jedoch nicht unbedingt der Fall; die dynamische Erweiterung des Klassenpfads ist dann nicht möglich.

Sie können nun mit `addUrl()` dem Classloader das zusätzliche Quellcode-Verzeichnis oder die zusätzliche JAR-Datei in Form einer URL hinzufügen.

```
rootLoader.addURL(new URL("file:///C:/java/commons-email-1.0/commons-email-1.0.jar"))
```

Holen Sie sich nun die gewünschte Klasse als `Class`-Objekt und verwenden Sie es dazu, per Reflection eine Instanz der benötigten Klasse zu erzeugen. Es ist nicht möglich, hier

einfach über `new org.apache.commons.mail.SimpleEmail()` den Konstruktor dieser Klasse aufzurufen, denn diese kann dem Compiler ja noch nicht bekannt sein. Nachdem Sie aber erst einmal eine Instanz von `SimpleEmail` in der Hand haben, können Sie es behandeln wie jedes andere Objekt in Groovy.

```
def emailClass = Class.forName("org.apache.commons.mail.SimpleEmail")
def email = emailClass.newInstance()
email.addTo("tim@oreilly.de","Tim O'Reilly")
email.hostName = "mail.myserver.com"
email.from = "ich@user.org"
email.subject = "Testnachricht"
email.msg = "Dies ist eine Testnachricht."
email.send()
```

Klassen definieren und verwenden

Im Unterschied zu Java müssen Klassennamen und Dateinamen in Groovy nicht unbedingt übereinstimmen. Sie können durchaus mehrere öffentliche Klassen in einer Datei haben, und diese können anders benannt sein als die Datei. Lediglich wenn sich ungebundener Skriptcode in der Quelldatei befindet, darf keine der explizit in der Datei definierten Klassen denselben Namen haben wie die Quelldatei selbst.

Wenn Sie eine Quelldatei mit mehreren Klassendefinitionen kompilieren, entstehen mehrere *.class*-Dateien mit den Namen der Klassen, die Sie normal verwenden können.

Kompilieren Sie die Quelldatei nicht, sondern führen Sie sie mit `groovy` direkt aus, müssen Ihre Klassen allerdings von dem Groovy-eigenen Classloader gefunden werden können, der trotzdem nach einer Datei mit einem Namen der Form *Klassenname.groovy* sucht. Es genügt aber, wenn nur eine der in einer Datei befindlichen Klassen mit dem Dateinamen übereinstimmt und diese als erste gesucht wird. Groovy kompiliert dann auch die anderen Klassen in der Datei intern, so dass sie anschließend im Programm verwendet werden können.

Die standardmäßige Sichtbarkeit von Klassen ist in Groovy `public`, Sie brauchen also keinen Sichtbarkeitsmodifikator anzugeben, wenn eine Klasse aus anderen Packages heraus verwendbar sein soll. Möchten Sie das verhindern, müssen Sie `protected` voranstellen. Dies hat praktisch die gleiche Auswirkung, als würden Sie in Java gar keinen Sichtbarkeitsmodifikator angeben.[1]

```
class ErsteKlasse {
    //Diese Klasse ist public
}
protected class ZweiteKlasse {
    //Diese Klasse ist Package-sichtbar
}
```

1 Wie schon in Kapitel 2 erwähnt, behandelt Groovy `protected`-Deklarationen beim Aufruf derzeit nicht anders als `public`. Das Schlüsselwort `protected` hat also nur eine Auswirkung, wenn die betreffende Klasse aus einer Java-Klasse benutzt wird.

Ihre Groovy-Klassen können wie in Java von anderen Klassen abgeleitet sein, dabei gelten die gleichen Mechanismen, die Sie von Java her kennen. Die Klassen, von denen Sie ableiten, können natürlich auch in Java programmiert sein. Dasselbe gilt für die Implementierung von Interfaces.

Die Verwendung eigener Interfaces innerhalb von Groovy-Programmen ist in der Regel nicht erforderlich, weil das Vorhandensein bestimmter Methoden und Properties ohnehin erst zur Laufzeit am betreffenden Objekt überprüft wird; Interfaces sind in Groovy also kein Mittel, Typsicherheit zur Kompilierzeit herzustellen. Gleichwohl ist es durchaus möglich, Interfaces auch in Groovy zu definieren. Sie werden benötigt, um mit bestehenden Klassen und Bibliotheken zusammenzuarbeiten. Sie entsprechen in jeder Hinsicht ihren Java-Vorbildern; daher werden wir uns mit dem Erstellen von Interfaces in Groovy nicht näher beschäftigen.

Ausführbare Klassen

Damit eine Groovy-Klasse direkt aus der Quelldatei mit `groovy` ausgeführt werden kann, muss eine der folgenden Bedingungen erfüllt sein.

Zum einen kann es sich wie bei normalen Java-Programmen um eine Hauptklasse mit einer `main()`-Methode handeln.

```
class Hauptklasse {
    static void main(args) {
        println "Hallo Welt"
    }
}
```

Die zweite Möglichkeit ist eine Klasse, die das Interface `Runnable` implementiert. Hier wird die Methode `run()` aufgerufen.[2]

```
class RunnableKlasse implements Runnable {
    void run() {
        println "Hallo Welt"
    }
}
```

Drittens kann es eine Klasse sein, die von `groovy.util.GroovyTestCase` oder `groovy.util.GroovyTestSuite` abgeleitet ist. In diesem Fall wird die Klasse als JUnit-Test angesehen und durch den Groovy-eigenen Testrunner abgearbeitet (siehe Kapitel 9).

```
class ReverseTest extends GroovyTestCase {
    public void testReverse() {
        assertEquals "tleW ollaH", "Hallo Welt".reverse()
    }
}
```

2 Das Interface `Runnable` wird in Java eigentlich für Threads verwendet; in Groovy dient es unabhängig davon aber auch der Kennzeichnung ausführbarer Klassen.

Schließlich kann das Programm auch die Form eines Skripts mit ungebundenen Anweisungen haben; in diesem Fall wird der Skriptcode direkt ausgeführt. Wie das geht, werden Sie weiter unten im Abschnitt »Skriptobjekte« sehen.

Das Interface GroovyObject

Jedes vom Groovy-Compiler erzeugte Objekt implementiert (neben den im Programm angegebenen Interfaces) das Interface `GroovyObject`. Da das Objekt nicht von einer Groovy-eigenen Oberklasse abgeleitet sein darf – Sie könnten sonst selbst keine Ableitungshierarchien bilden –, müssen die in `GroovyObject` deklarierten Methoden vom Compiler generiert werden. In der Regel delegieren sie allerdings nur über mehrere Zwischenstationen an ein sogenanntes *Metaobjekt* weiter, das die eigentliche Arbeit macht. Das Metaobjekt gibt Ihnen diverse Möglichkeiten, in die Funktionalität einer Klasse einzugreifen; mehr dazu erfahren Sie in Kapitel 7.

Es ist von geradezu fundamentaler Wichtigkeit, dass Sie zumindest die Namen der in `GroovyObject` deklarierten Methoden kennen, denn wenn Sie diese überschreiben, wird Ihr Programm nicht mehr funktionieren, ohne dass Sie ohne Weiteres die Ursache erkennen können. Das Interface `GroovyObject` ist folgendermaßen definiert:

Beispiel 3-1: Das Interface GroovyObject

```
// Java
package groovy.lang;
public interface GroovyObject {
    Object invokeMethod(String name, Object args);
    Object getProperty(String property);
    void setProperty(String property, Object newValue);
    MetaClass getMetaClass();
    void setMetaClass(MetaClass metaClass);
}
```

Eine Dokumentation der einzelnen Methoden finden Sie in Anhang B.

Skriptobjekte

Unter einem Groovy-Skript verstehen wir ein Stück Groovy-Quellcode, das keine Klasse und keine Methode eingebunden hat. In Java gibt es so etwas nicht, auch das einfachste und kürzeste Programm muss in Klassen und Methoden organisiert sein. In Groovy können Sie ein ganzes lauffähiges Programm in eine einzige Zeile schreiben; das folgende Beispiel kennen Sie schon aus dem ersten Kapitel:

```
println "Hallo Welt!"
```

Wenn Sie diese Zeile in eine Datei *HalloWelt.java* speichern und dann mit `groovyc` kompilieren, erhalten Sie eine Java-Klassendatei namens *HalloWelt.class*, die Sie mit `java` ausführen können. Aber auch wenn Sie diese Datei mit `groovy` direkt ausführen oder mit

groovysh bzw. in der GroovyConsole interaktiv ausführen, übersetzt Groovy diese Zeile in eine normale Java-Klasse und führt diese anschließend aus.

```
> groovy HalloWelt
Hallo Welt!
```

Groovy-Skripte kommen nicht nur in der Form von Skriptdateien ins Spiel, sondern auch bei der Integration dynamischer Programme in Java-Anwendungen. Ein Beispiel dafür sind Groovlets, mit denen wir uns an anderer Stelle beschäftigen (siehe Kapitel 6).

Für Skripte gelten einige Besonderheiten gegenüber normalen, in Klassen organisierten Programmen.

Aufgelockerte Form von Groovy-Skripten

Ein Groovy-Skript kann ungebundene Anweisungen und Methodendefinitionen[3] in beliebiger Mischung enthalten. import-Anweisungen müssen nicht unbedingt am Anfang stehen; die Definitionen von Funktionen brauchen sich nicht vor der Stelle befinden, an der erstmalig auf sie zugegriffen wird. Eine Skriptdatei kann auch zusätzliche Klassendefinitionen enthalten, diese sind dann aber kein Teil des Skripts, sondern werden getrennt übersetzt.

Sehen wir uns ein kurzes Beispielskript an, das alle diese Elemente in einfacher Form enthält (Beispiel 3-2).

Beispiel 3-2: Das Skript BeispielSkript.groovy

```
import javax.swing.*

fenstertext = 'Das ist der Beispieltext'
zeigeFenster(fenstertext)

def zeigeFenster(text) {
    def bf = new BeispielFrame(text)
    bf.visible=true
}

class BeispielFrame extends JFrame {
    def BeispielFrame(text) {
    title = "Ein Beispiel-Frame"
    defaultCloseOperation = DISPOSE_ON_CLOSE
    contentPane.add(new JLabel(text.toString()))
    pack()
    }
}
```

3 Streng genommen definieren wir hier keine Methoden, sondern Funktionen, da sie auf Ebene der Groovy-Syntax keiner Klasse zugeordnet sind. Diese werden aber von Groovy in eine Methode einer Klasse übersetzt, die dem Skript entspricht.

Wenn Sie dieses Skript in eine Textdatei namens *BeispielSkript.groovy* schreiben und mit `groovy` ausführen, erscheint links oben auf dem Bildschirm ein kleines Fenster mit dem Text »Dies ist ein Beispieltext«. Das eigentliche Skript besteht aus zwei Zeilen, die erst den Text definieren und dann eine eingebettete Methode `zeigeFenster()` aufrufen. Diese wiederum instantiiert eine in derselben Datei definierte Klasse `BeispielFrame`. Sowohl die Methode als auch die Klasse werden vor ihrer Definition benutzt, und die für `JFrame` und `JLabel` erforderliche `import`-Anweisung steht ganz am Ende – ein deutliches Zeichen dafür, dass wir es hier mit einem Compiler zu tun haben, der das Programm erst als Ganzes übersetzt und dann ausführt, und nicht mit einem zeilenweise arbeitenden Skript-Interpreter.

Es fällt Ihnen vielleicht auf, dass wir den Konstruktor von `JLabel` mit `text.toString()` aufrufen. Das ist eine Sicherheitsmaßnahme, da das Argument `text` hier nicht typisiert ist, und wir möchten nicht, dass eine `ClassCastException` ausgelöst wird, falls jemand unseren `BeispielFrame` mit irgendetwas anderem als einem String aufruft. Innerhalb von Groovy kann man sehr gut mit untypisierten Variablen arbeiten, aber sobald Sie typisierte Java-APIs aufrufen, sorgen Sie besser dafür, dass die übergebenen Argumente auch den richtigen Typ haben. Groovy nimmt zwar einfache Typkonvertierungen für Sie vor, wird aber beispielsweise kein `Date`-Objekt in einen String umwandeln, wenn eine API-Methode einen String erwartet.

Sobald der Groovy-Compiler (egal ob in einem `groovy`-Aufruf oder beim expliziten Kompilieren mit `groovyc`) auf einzeln stehende Anweisungen trifft, generiert er eine Klasse, deren Name dem Namen der Skriptdatei entspricht, und in ihr eine Methode mit dem Namen `run()`, die die einzeln stehenden Anweisungen enthält. Innerhalb des Skripts definierten Methoden werden zu Methoden dieser Klasse. Außerdem erhält die Klasse eine `main()`-Methode, die (auf Umwegen) im Wesentlichen nichts weiter macht, als die Skriptklasse zu instantiieren und `run()` aufzurufen; dadurch wird aus dem zu einer *.class*-Datei kompilierten Skript ein mit dem `java`-Befehl ausführbares Programm. Alle innerhalb des Skripts definierten Klassen werden zu getrennten Java-Klassen; beim expliziten Kompilieren entstehen auch die entsprechenden *.class*-Dateien.

Das Skript als Klasse

Sie können das Skript auch mit `groovyc` kompilieren und erhalten dann die zwei Klassendateien *BeispielSkript.class* und *BeispielFrame.class*. Die erste der beiden ist folgendermaßen definiert (Sie können das mit dem Java-Disassembler des JDK (*javap*) oder besser noch mit einem Java-Decompiler, z.B. JAD (*http://www.kpdus.com/jad.html*), leicht nachprüfen):

```
public class BeispielSkript extends groovy.lang.Script {
  public BeispielSkript() {...}
```

```
    public BeispielSkript(groovy.lang.Binding context) {...}
    public static void main(java.lang.String[] args) {...}
    public java.lang.Object run(){...}
    public java.lang.Object zeigeFenster(java.lang.Object text) {...}
    public static {} {...}
    public static java.lang.Long __timeStamp = ...;
}
```

Unter anderem finden Sie dort die generierte `main()`-Methode, die selbst geschriebene Methode `zeigeFenster()` und das eigentliche Skript als nicht statische Methode `run()` wieder. Wenn Sie ein Skript erstellen, schreiben Sie also im Grunde diese `run()`-Methode einer von `Script` abgeleiteten Klasse. Das heißt, Sie befinden sich mit dem Skript in einer vollkommen objektorientierten Umgebung, auch wenn man es dem Skript selbst nicht unbedingt ansieht.

Die Variable `__timeStamp` dient übrigens dem Compiler zur Prüfung, ob die Klasse nach einer Änderung des Quellcodes neu übersetzt werden muss.

Skripte und das Binding

Vielleicht ist Ihnen aufgefallen, dass wir in dem obigen Beispiel eine Variable namens `fenstertext` verwenden, der ein String mit dem anzuzeigenden Text zugewiesen wird, obwohl sie nirgendwo deklariert ist. Genau wie in Java müssen in Groovy Variablen, egal ob es Felder einer Klasse oder nur innerhalb einer Methode benutzte lokale Variablen sind, immer deklariert werden, bevor sie verwendet werden können. In Skripten gibt es aber insofern eine Ausnahme, als jeder Aufruf einer unbekannten Variablen gegen das sogenannte *Binding* aufgelöst wird. Das Binding ist ein Objekt der Klasse `groovy.lang.Binding`, das dem Skriptobjekt als Property zugeordnet ist und dazu dient, Daten zwischen dem Skript und der Außenwelt auszutauschen.

Das Binding ist ein Behälterobjekt ähnlich einer Map (implementiert allerdings nicht das Interface `java.util.Map`), in dem beliebige Objekte unter einem String-Namen abgelegt werden können. Wenn Sie also in einem Skript versuchen, einen Wert einer Variablen zuzuweisen, die weder im `Script`-Objekt noch innerhalb des Skripts selbst angelegt worden ist, trägt Groovy einfach diesen Wert unter dem angegebenen Variablennamen in das Binding ein.

Ein im Binding eingetragener Wert kann überall im Skript, also auch in Methoden, die innerhalb des Skripts definiert sind, wie eine normale untypisierte Variable verwendet werden. Für normale Klassen (wie `BeispielFrame` im obigen Beispiel), die innerhalb derselben Skriptdatei definiert sind, sind die Binding-Variablen jedoch nicht sichtbar. Solche eingebetteten Klassen sind logisch vollständig vom Skript getrennt, da Groovy keine inneren Klassen kennt.

Lesend auf eine Binding-Variable zugreifen können Sie nur, wenn zuvor ein Wert unter dem entsprechenden Namen im Binding abgelegt worden ist. Dies muss nicht unbe-

dingt innerhalb des Skripts geschehen sein, da das Binding dem Skript in der Regel mit irgendwelchen vorbelegten Werten übergeben wird. Wenn Sie versuchen, im Skript auf einen Wert zuzugreifen, der nicht als Member-Variable, lokale Variable oder Binding-Variable bekannt ist, erhalten Sie wie in jeder anderen Methode auch eine `MissingPropertyException`.

Eine im Binding gespeicherter Wert verhält sich im Wesentlichen wie eine Property der Skriptklasse. So können Sie den Wert beispielsweise auch mit `getProperty()` abrufen. Die Methode `getProperties()` listet ihn allerdings nicht mit auf.

Sie können sich den Inhalt des Bindings leicht ansehen, indem Sie die Property gleichen Namens abrufen. Tragen Sie beispielsweise folgende Ausgabeanweisung in das obige Skriptbeispiel ein:

```
fenstertext = 'Das ist der Beispieltext'
println binding.variables
```

Sie erhalten dann so etwas wie die folgende Ausgabe im Konsolenfenster:

```
{fenstertext=Das ist der Beispieltext, args=[Ljava.lang.String;@162dbb6}
```

Daran können Sie sehen, dass zwei Binding-Variablen definiert sind, und zwar der von uns stammende `fenstertext` sowie eine Variable `args`, mit der wir uns gleich beschäftigen werden.

Die wichtigsten Methoden des `Binding`-Objekts sind `setVariable(String,Object)` zum Setzen einer Binding-Variablen, `getVariable(String)` zum Lesen einer Variablen und `getVariables()`, die wir eben angewendet haben und die ein `Map`-Objekt mit allen im Binding gehaltenen Werten liefert. Eine vollständige Auflistung enthält Anhang B.

Die Aufrufargumente eines Skripts

Groovy übergibt dem Skript die Aufrufargumente aus der Befehlszeile als String-Array in einer Binding-Variablen mit dem Namen `args`. Folgendes kleine Skript tut nichts weiter, als alle übergebenen Argumente aufzulisten.

```
for (arg in args) println arg
```

Speichern wir es als *ZeigeArgumente.groovy* ab und rufen wir es aus dem Konsolenfenster – im selben Verzeichnis – auf:

```
> groovy ZeigeArgumente Das sind "3 Argumente"
Das
sind
3 Argumente
```

GroovyBeans

In Java bezeichnet man Klassen, die nach bestimmten Konventionen programmiert sind, als JavaBeans. Eine der Konventionen besteht in der Regel, dass eine JavaBean über Properties (Eigenschaften) verfügt, auf die von außen nur mithilfe von Zugriffsmethoden (Akzessoren und Mutatoren) zugegriffen werden kann. Wenn es also eine Property namens `eigenschaft` gibt, kann sie aus einer anderen Klasse nur mit der Methode `getEigenschaft()` ausgelesen und mit der Methode `setEigenschaft()` gesetzt werden. Was sich hinter `eigenschaft` im Einzelnen verbirgt, ist nicht gesagt. Häufig aber implementiert die Klasse eine private Member-Variable gleichen Namens, auf die über die beiden Methoden zugegriffen wird. Umgangssprachlich werden die Zugriffsmethoden meist einfach als Setter- und Getter-Methoden bezeichnet.

Ursprünglich war die JavaBean-Konvention als Standard für die Gestaltung von Oberflächenkomponenten gedacht, die dann aufgrund der Namenslogik mit Werkzeugen konfiguriert werden konnten, die diese Komponenten nicht im Einzelnen kennen mussten. Inzwischen ist es aber weitgehend Usus geworden, die Regeln allgemein bei der Programmierung von Klassen anzuwenden, und es gehört zum Standard, dass man auf Member-Variablen von außen normalerweise nicht direkt, sondern immer über Setter und Getter zugreift.

Im Unterschied zu Java wird die JavaBean-Konvention bezüglich des Zugriffs auf Properties durch die Sprache Groovy direkt unterstützt. Hier nimmt Ihnen der Compiler die Definition der Zugriffsmethoden ab, und auch die Syntax für den Zugriff auf Objekt-Properties ist vereinfacht.

Bohnen-Konventionen lohnen

Die JavaBeans-Konvention hat sich nicht ohne Grund so stark verbreitet: Java bietet von sich aus keinen einheitlichen Zugriff auf Methoden und Attribute. Wenn es nun innerhalb einer Vererbungshierarchie oder während der Programmweiterentwicklung zu der Situation kommt, dass ein Attributwert beim Zugriff errechnet und nicht aus einem Datenfeld gelesen werden soll, reicht es nicht, das Attribut durch eine Methode zu ersetzen – es müssen auch alle Stellen, an denen das Attribut benutzt wird, geändert werden. Diese lästige Situation verhindert die JavaBeans-Konvention: Wenn konsequent nur getter- und setter-Methoden genutzt werden reicht es aus, die Implementierung der entsprechenden Methoden zu ändern. In Groovy ist das durch GroovyBeans dann noch viel einfacher.

Member deklarieren

Grundsätzlich werden die Member einer Klasse (Felder und Methoden) genau so deklariert wie in Java: Sie geben den Typ, bei Bedarf ergänzt durch Typmodifikatoren wie

static und private, und den Namen an. Wenn Sie möchten, können Sie, getrennt durch ein Gleichheitszeichen, gleich einen Initialwert zuweisen. Da weder Felder noch Methoden typisiert sein müssen, können Sie die Typangabe auch weglassen. Es muss aber für Groovy erkennbar sein, dass es sich um eine Deklaration handelt, wenn also kein Typmodifikator benötigt wird, muss die Deklaration durch das Schlüsselwort def kenntlich gemacht werden. Hier einige Beispiele für gültige Felddeklarationen:

```
Integer i       // Instanzvariable vom Typ Integer
Integer[] j     // Instanzvariable vom Typ Integer-Array
static a        // Untypisiertes statisches Feld
private b       // Untypisierte private Instanzvariable
def x = "Start" // Untypisierte Instanzvariable, mit String initialisiert
```

Bei der Variablen x ist das Schlüsselwort def erforderlich, weil sonst nicht zu erkennen wäre, dass es sich um eine Variablendeklaration handelt. Bei a und b wird es nicht benötigt, weil die Deklaration durch einen Typmodifikator kenntlich ist; allerdings stört es auch nicht, wenn Sie das def trotzdem hinzufügen.

Beachten Sie, dass die Regeln für die Sichtbarkeit der Member etwas von Java abweichen.

- Die standardmäßige Sichtbarkeit, die gilt, wenn Sie selbst keine Angabe machen, ist bei Methoden public und bei Feldern private, wobei bei Feldern automatisch Getter und Setter generiert werden (mehr dazu gleich im Anschluss).
- Die Package-Sichtbarkeit, die in Java beim Fehlen einer Deklaration angenommen wird, wird in Groovy nicht unterstützt, da sie in den Augen der Groovy-Entwickler auch in Java nur sehr selten verwendet wird.

Felder und Properties

Wenn Sie in Groovy ein Feld für eine Klasse deklarieren und keine Angaben über die Sichtbarkeit machen, nimmt der Compiler an, dass es sich um ein Property-Feld handelt. Das bedeutet, dass er das Feld selbst als private betrachtet und von sich aus öffentliche Getter- und Setter-Methoden hinzufügt, sofern diese Methoden nicht bereits explizit definiert sind.

Die folgenden fünf mageren Zeilen enthalten also die vollständige Definition einer Bean mit drei Properties.

```
class Datum1 {
    Integer jahr
    Integer monat
    Integer tag
}
```

Dieser Code ist vollständig äquivalent mit der folgenden ausführlichen Version.

```
class Datum1 {
    private Integer jahr
    private Integer monat
```

```
    private Integer tag
    Integer getJahr() {
       return jahr
    }
    void setJahr (Integer jahr) {
       this.jahr = jahr
    }
    Integer getMonat() {
       return monat
    }
    void setMonat (Integer monat) {
       this.monat = monat
    }
    Integer getTag() {
       return tag
    }
    void setTag (Integer tag) {
       this.tag = tag
    }
}
```

Groovy fügt die beiden Zugriffsmethoden tatsächlich als Bytecode hinzu. Dies ist insofern wichtig, als Sie vielleicht auch aus normalen Java-Programmen über die Getter- und Setter-Methoden zugreifen möchten, und dann müssen sie auch im normalen Code als richtige Methoden vorhanden sein. Damit sind GroovyBeans in Umgebungen wie *Inversion-of-control*-Containern, beispielsweise Spring und Apache Avalon, einsetzbar, bei denen die Einhaltung der Bean-Konventionen eine besondere Rolle spielt, ohne dass Sie sich mit der Programmierung der Zugriffsmethoden aufhalten müssen.

Eine Klasse für Datumsangaben wird Ihnen als Beispiel in diesem Buch von nun an öfter begegnen. Der Grund dafür ist nicht nur, dass sie sich als leicht verständliches Beispiel gut eignet. Sie kann auch – in einer etwas weiter ausgebauten Form – ausgesprochen nützlich sein, denn die zum JDK gehörenden Kalenderklassen sind zwar recht vielseitig, aber nicht gerade ein Musterbeispiel an Benutzerfreundlichkeit. An unseren in Groovy geschriebenen Datumsklassen können Sie sehen, wie einfach es Groovy Ihnen macht, komplizierte Dinge mit einer handhabbaren Kapsel zu versehen.

GroovyBeans implementieren auch Interfaces. Angenommen, Sie haben ein Interface `DatumIf` (egal, ob in Groovy oder Java programmiert):

```
public interface DatumIf {
    Integer getJahr()
    void setJahr (Integer jahr)
    Integer getMonat()
    void setMonat (Integer monat)
    Integer getTag()
    void setTag (Integer tag)
}
```

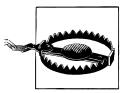
Als Implementierung genügt die folgende kurze Groovy-Klasse, denn die Getter und Setter werden ja vom Compiler hinzugeneriert.

```
class Datum1 implements DatumIf {
    Integer jahr
    Integer monat
    Integer tag
}
```

Der Groovy-Compiler hält sich jedoch vornehm zurück, wenn Sie selbst schon tätig geworden sind, also eigene Getter oder Setter geschrieben haben. Das gibt Ihnen die Möglichkeit, zusätzliche Funktionalität wie Plausibilitätsprüfungen oder Umformatierungen unterzubringen. Ein typisches Beispiel wäre Folgendes:

```
class Datum1 implements DatumIf {
    Integer jahr
    ...
        void setJahr (Integer jahr) {
           if (jahr<1800 || jahr>2100) {
               throw new IllegalArgumentException("Ungültiges Jahr: $jahr")
           }
           this.jahr = jahr
        ...
}
```

Während die Getter-Methode `getJahr()`, wie auch die übrigen Getter und Setter, nach wie vor von Groovy generiert wird, haben wir hier einen manuellen Setter, der eine Eingabeprüfung vornimmt.



Wenn Sie Getter oder Setter zu Groovy-Properties selbst schreiben, müssen Sie akribisich auf die korrekte Signatur achten. So dürfen Sie im obigen Code beispielsweise nicht den Setter mit `def` statt mit `void` definieren, sonst kann es passieren, dass Groovy trotzden seinen eigenen Setter generiert und Ihrer nie aufgerufen wird.

Wir können diese Möglichkeit natürlich auch nutzen, um mit unserer Datumsklasse einfach den Java-eigenen Kalender zu kapseln und etwas leichter nutzbar zu machen.

```
class KalenderDatum {
   Calendar calendar = new GregorianCalendar()
   void setJahr (Integer jahr) {
      calendar.set(Calendar.YEAR,jahr)
   }
   Integer getJahr () {
      calendar.get(Calendar.YEAR)
   }
   // Hier die übrigen Getter und Setter
   // nach dem gleichen Muster.
}
```

Nun können wir sogar das aktuelle Datum bekommen und weiterhin die Bestandteile des Datums bequem einzeln setzen und auslesen:

```
groovy> k = new KalenderDatum()
groovy> println k.getJahr()
groovy> k.setJahr(1949)
groovy> println k.getJahr()
2007
1949
```

Die erste ausgegebene Jahreszahl stammt aus dem innerhalb unseres `KalenderDatum` frisch instantiierten `GregorianCalendar`, und die zweite haben wir im Skript selbst gesetzt. Dies können Sie natürlich in Java genau so machen; die Besonderheiten des Kapselns in einer Groovy-Klasse werden erst weiter unten deutlicher werden. Immerhin können wir aber das Datum in unserem `KalenderDatum` schon einmal sehr bequem setzen:

```
groovy> k = new KalenderDatum()
groovy> k.setCalendar(new GregorianCalendar(1962,3,29))
groovy> println k.getJahr()
1962
```

Groovy-Properties sind von Hause aus immer les- und schreibbar. Sie können zwar Getter und Setter überschreiben, aber nicht verhindern, dass sie generiert werden, falls Sie etwa Nur-schreib- oder Nur-lese-Properties haben und die Getter bzw. Setter gar nicht wollen. In diesem Fall müssen Sie den Umstand ausnutzen, dass Groovy generell keine Getter und Setter generiert, sobald Sie die Sichtbarkeit eines Felds selbst explizit festlegen. Um beispielsweise eine Property zu definieren, auf die von außen nur lesend zugegriffen werden kann, deklarieren Sie diese als `private` und schreiben die noch benötigte Zugriffsmethode selbst, so wie Sie es in Java auch tun würden. Die folgende Variante der Datumsklasse ist immutabel, d.h., ihre Werte können lediglich im Konstruktor gesetzt und später nur noch gelesen werden.

```
class NurLeseDatum {
    private Integer jahr
    private Integer monat
    private Integer tag
    NurLeseDatum (Integer jahr, Integer monat, Integer tag) {
       this.jahr = jahr
       this.monat = monat
       this.tag = tag
    }
    Integer getJahr() {
       return jahr;
    }
    Integer getMonat() {
       return monat;
    }
    Integer getTag() {
       return tag;
    }
}
```

Auf Properties zugreifen

Wenn Sie auf die Property eines Objekts aus einem anderen Objekt zugreifen möchten, können Sie dies natürlich in der von Java gewohnten Weise tun, indem Sie die Getter und Setter der Property aufrufen. Wir probieren dies mit einem Skript aus, das unsere obige Klasse `Datum1` instantiiert.

```
def datum = new Datum1()
datum.setJahr(1984)
println datum.getJahr()
```

Dies ist korrekt, es ist aber nicht das, was man in Groovy einen Property-Zugriff nennen würde; es ist schlicht der Aufruf zweier Methoden (die freilich genau dies tun). In Groovy greifen Sie auf die Properties eines Objekts zu, als wären es öffentliche Felder.

```
def datum = new Datum1()
datum.jahr = 1984
println datum.jahr
```

Daneben gibt es noch eine weitere Möglichkeit. Sie können auf die Properties einer Klasse auch ähnlich den Elementen eines Arrays zugreifen; allerdings ist dabei der Index nicht numerisch, sondern ein String.

```
def datum = new Datum1()
datum['jahr'] = 1984
println datum['jahr']
```

Der Vorteil der alternativen Schreibweisen liegt weniger in deren Eleganz (über die man ohnehin streiten kann), sondern darin, dass Sie auf diese Weise die Flexibilität und Dynamik der Sprache Groovy erst so recht nutzen können. Wenn Sie *`referenz.feldname`* oder *`referenz['feldname']`* schreiben, anstatt die Getter und Setter zu benutzen, ist es nämlich egal, wie die Property implementiert ist; es muss nur die entsprechende Information geliefert werden. Diese Zugriffe funktionieren sogar, wenn `datum` auf eine `Map` verweist. Probieren Sie es:

```
def datum = new HashMap()
datum.jahr = 1984
println datum.jahr
datum['monat'] = 12
println datum['monat']
```

Groovy verfolgt das Prinzip, dass der Zugriff auf verschiedene Arten von Objekten möglichst gleichartig sein soll. Das macht es nicht nur einfacher, sich zu merken, was bei einem bestimmten Objekt zu tun ist, es ist auch das Gegenstück zu der dynamischen Typisierung, die Groovy Ihnen bietet. Wenn Sie eine Methode haben, die sich für die Jahreszahl eines Datums interessiert, ist es völlig egal, wie das entsprechende Objekt implementiert ist, Hauptsache, es liefert unter dem Namen `jahr` die entsprechende Information.

Was geschieht nun genau, wenn ein Groovy-Programm eine Referenz auf eine Property auflöst:

1. Wenn es sich um einen lokalen Zugriff handelt und es ein Feld mit dem genannten Namen gibt, wird dieses direkt adressiert.
2. Wenn das Objekt das Interface `Map` implementiert, wird deren Methode `get()` bzw. `set()` aufgerufen.
3. Wenn es eine korrespondierende Zugriffsmethode (Getter bzw. Setter) gibt, wird diese für den Zugriff verwendet.
4. Wenn es ein Feld mit dem Namen der Property mit passenden Zugriffsrechten gibt, wird dieses ausgelesen bzw. gesetzt.
5. Wenn in der betreffenden Klasse eine Methode mit der Signatur `Object get(String)` bzw. `void set(String,Object)` existiert, wird diese mit dem Namen der Property als Argument aufgerufen.

Aus der zweiten Option folgt, dass Sie auf die Elemente einer Map wie auf die Properties eines Objekts zugreifen können. Im obigen Beispiel bedeutet dies also:

```
def datum = new HashMap()
datum.jahr = 1984            // entspricht: datum.put('jahr',1984)
println datum.jahr           // entspricht: println datum.get('jahr')
datum['monat'] = 12          // entspricht: datum.put('monat',12)
println datum['monat']       // entspricht: println datum.get('monat')
```

Die letzte der fünf Möglichkeiten erlaubt es Ihnen, mit Properties recht kreativ umzugehen. Was die Methoden `get()` und `set()` im Einzelnen anstellen, interessiert Groovy nicht, und was Sie mit den übergebenen bzw. angeforderten Werten machen, bleibt gänzlich Ihnen überlassen. Sowohl `get()` als auch `set()` werden nur aufgerufen, wenn keine der anderen Möglichkeiten anwendbar ist. Damit bieten sie Ihnen gewissermaßen einen letzten Ausweg, wenn Sie Zugriffe auf nicht existierende Properties abfangen möchten. Wir können dies mithilfe einer leicht erweiterten Version der ersten Version der `Datumsklasse` zeigen.

```
class Datum2 {
    Integer jahr
    Integer monat
    Integer tag
    Integer getJahrhundert() {
        return jahr/100
    }
    def get(String propertyname) {
        println "Lies unbekannte Property: $propertyname"
    }
    def set(String propertyname, Object wert) {
        println "Setze unbekannte Property: $propertyname=$wert"
    }
}
```

Dazu ein neues Experiment mit groovysh:

```
groovy> d = new Datum2()
groovy> d.jahr = 2000
groovy> println d.jahr
groovy> println d.jahrhundert
groovy> d.jahrhundert = 21
groovy> println d.wochentag
2000
20
Setze unbekannte Property: jahrhundert=21
Lies unbekannte Property: wochentag
null
```

Das Ergebnis zeigt, dass die Property jahr wie bisher direkt gesetzt und ausgelesen und die Property jahrhundert über die explizit geschriebene Zugriffsmethode getJahrhundert() erreicht wird; nur beim schreibenden Zugriff auf die Property jahrhundert sowie beim Lesen von wochentag muss Groovy auf get() und set() ausweichen, was an den beiden Meldungen zum Schluss zu sehen ist. (Die Null am Ende rührt daher, dass wir die nicht existierende Property d.wochentag ausgeben wollten, dabei aber an die get()-Methode geraten sind, die ja für get('wochentag') kein Ergebnis liefert.)

Der Flexibilität noch lange nicht genug. Um die obigen Property-Zugriffsarten 2 bis 5 ausführen zu können, generiert Groovy die beiden Methoden setProperty(String,Object) und getProperty(String). Auch diese können Sie überschreiben und damit interessante Kunststücke vollführen; dies gehört aber schon zum Thema Metaprogrammierung, auf die wir in Kapitel 7 zu sprechen kommen. Eine weitere in diesen Zusammenhang passende und von Groovy generierte Methode ist getProperties(). Sie ermöglicht es Ihnen, eine Map der in einem Objekt vorhandenen Properties selbst wie eine Property auszulesen.

```
groovy> def d = new Datum2()
groovy> d.jahr = 1990
groovy> d.monat = 10
groovy> d.tag = 3
groovy> println d.properties
["tag":3, "jahrhundert":19, "monat":10, "metaClass":groovy.lang.
MetaClassImpl@208506[class Datum2], "jahr":1990, "class":class Datum2]
```

Das Ergebnis ist vielleicht etwas erklärungsbedürftig. Die Properties tag, monat und jahr kennen wir, denn die haben wir selbst ins Leben gerufen. In jahrhundert spiegelt sich die selbst geschriebene Getter-Methode wider. Die Property class ist auch eine Bekannte: Es ist die Java-Klasse des Objekts, also das Ergebnis eines Aufrufs von getClass(). Neu und Groovy-spezifisch ist dagegen die Property metaClass, die von Groovy gesetzt wird und das standardmäßige Verhalten der zugeordneten Objekte bestimmt.

Bei der Suche nach Properties orientiert sich getProperties() an den in der Klasse definierten konventionellen Getter-Methoden. Und so findet sie in dem obigen Beispiel die explizit programmierte Methode getJahrhundert(), die anhand unserer Property-Felder generierten Methoden getJahr(), getMonat() und getTag(), die zur Java-Basisklasse

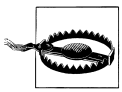
Object gehörende Methode getClass() und schließlich die von Groovy generierte Methode getMetaClass(). Fällt Ihnen etwas auf? Die ja offenbar vorhandene Methode getProperties() fehlt. Das ist vielleicht nicht ganz konsequent, verhindert aber eine Endlosschleife.

Bei manchen Klassen funktioniert getProperties() etwas anders. Wir benutzen noch einmal unsere HashMap von oben, mit der wir oben nur eine Bean vorgegaukelt haben.

Die Tatsache, dass getProperties() nur die »gemeinen« Getter berücksichtigt, führt auch dazu, dass es bei vorgegaukelten Beans wie dem obigen HashMap-Beispiel nicht funktioniert.

```
groovy> def datum = new HashMap()
groovy> datum.jahr = 1984
groovy> datum['monat'] = 12
groovy> println datum.getProperties()
["empty":false, "class":class java.util.HashMap, "forNullKey":null]
```

Hier bekommen wir konsequenterweise die Ergebnisse von isEmpty(), getClass() und getForNullKey() zu sehen, nicht aber die vermeintlichen Properties jahr und monat. An diesem Beispiel können Sie übrigens auch erkennen, dass properties doch keine richtige Property ist, auch wenn es eine vordefinierte Methode getProperty() gibt. Bei der HashMap würde die Schreibweise datum.properties versuchen, ein Map-Element »properties« auszulesen, das es nicht gibt, und als Folge nur null liefern. Daher mussten wir die Methode hier direkt aufrufen.



Generell empfiehlt es sich, lieber die Getter- oder Setter-Methoden anstelle der Property-Notation zu verwenden, wenn im Vorhinein nicht mit Sicherheit klar ist, mit was für einem Objekt Sie arbeiten. Beispielsweise besteht ein verbreiteter Fehler in Groovy-Programmen darin, dass die Klasse eines Objekts als Property abgefragt wird, etwa so:

```
println meinObjekt.class.name
```

Das geht meistens gut, weil meinObjekt.class in meinObjekt.getClass() übersetzt wird. Sobald meinObjekt aber eine Map ist, versucht Groovy, die Methode meinObjekt.get('class') auszuführen. Und wenn die Map zufällig unter dem Schlüssel »class« keinen gültigen Wert gespeichert hat, führt dies zu einer NullPointerException.

Schreibweisen für Property-Namen

Da ein Zugriff auf eine Property nicht unbedingt gleichbedeutend mit dem Zugriff auf ein Bean-Feld ist, kann es sein, dass der Name einer Property Zeichen enthält, die in einem Java-Feldnamen nicht erlaubt sind, zum Beispiel Leerzeichen, Bindestriche usw. Um die Verwendung solcher Namen auch in der Punktnotation zu ermöglichen, erlaubt Groovy die Angabe des Feldnamens in Anführungszeichen.

```
groovy> def datum = new HashMap()
groovy> datum.'tag im jahr' = 100
groovy> println datum.'tag im jahr'
```

Sie können sogar doppelte Anführungszeichen verwenden und mittels GString-Interpolation die Property-Namen zusammensetzen.

```
groovy> def feld = 'tag'
groovy> def datum = new HashMap()
groovy> datum."$feld im jahr" = 100
groovy> println datum."$feld im jahr"
```

Wir nutzen diesen Umgang mit Properties in unserer obigen Klasse `KalenderDatum` aus, in der wir zum Speichern des Datums einfach ein Java-`Calendar`-Objekt verwenden. Die einzelnen Datums- und Zeitfelder dieser Klasse werden durch eine spezielle `get()`- und eine spezielle `set()`-Methode ausgeführt, die jeweils als erstes bzw. einziges Argument eine Indexnummer des gewünschten Felds annimmt. Für diese Indexnummer sind in der `Calender`-Klasse Konstanten vordefiniert. Wir ergänzen nun `KalenderDatum` um eine `get()`- und eine `set()`-Methode, die als Auffangnetz für Property-Zugriffe dienen, für die es keine spezifischen Getter und Setter gibt. Die Aufrufe dieser Methoden leiten wir nun an die `get()`- und `set()`-Methoden des `Calendar`-Objekts weiter und wandeln dabei die Property-Namen durch String-Interpolation in die Namen der Konstanten für die Felder um. Das klingt komplizierter, als es ist; sehen Sie es sich einfach mal an:

```
class KalenderDatum {
   Calendar calendar = new GregorianCalendar()
   def get(String propertyname) {
       calendar.get(Calendar."$propertyname")
   }
   def set(String propertyname, wert) {
       calendar.set(Calendar."$propertyname", wert)
   }
}
```

Nun können wir einfach die Konstantennamen aus der `Calendar`-Klasse als Property-Namen für `KalendarDatum` verwenden und recht elegant die Datumsfelder auslesen oder setzen, als wären es Properties unserer `KalenderDatum`-Klasse.

```
groovy> k = new KalenderDatum()
groovy> k.setCalendar(new GregorianCalendar(1999,11,12))
groovy> println k.DAY_OF_MONTH+'.'+k.MONTH+'.'+k.YEAR
groovy> k.DAY_OF_MONTH = k.DAY_OF_MONTH+42
groovy> println k.DAY_OF_MONTH+'.'+k.MONTH+'.'+k.YEAR
12.11.1999
23.0.2000
```

In der Ausgabe erscheint erst das Datum, das wir per neuem `GregorianCalendar` frisch gesetzt haben, und dann – nachdem wir die Tage um 42 (= 6 Wochen Urlaub) erhöht haben – ein entsprechend weiter geschaltetes Datum. Natürlich stören uns hier die Property-Namen in Großbuchstaben, und wir hätten gern, dass `get()` und `set()` auch mit dem Aufruf `k.dayOfMonth` anstelle von `k.DAY_OF_MONTH` zurechtkommen, denn wir wollen ja keinen konstanten Wert auslesen. In Java müssten wir nun eine lange, verschachtelte `if`-Verzweigung programmieren. Zum Glück haben wir es in einer dynamischen Sprache wie Groovy leichter.

```
class KalenderDatum {
   def get(String propertyname) {
       calendar.get(calendarConst(propertyname))
   }
   void set(String propertyname, wert) {
       calendar.set(calendarConst(propertyname), wert)
   }
   private calendarConst(String propertyname) {
       try {
          def constname = propertyname.replaceAll('[A-Z]','_$0').toUpperCase();
         return Calendar."$constname"
       } catch (MissingPropertyException ex) {
          ex.printStackTrace()
          throw new MissingPropertyException(propertyname,KalenderDatum)
       }
   }
}
```

Wir haben eine kurze Methode namens `calendarConst()` eingeführt, die den Property-Namen von einer CamelCase-Notation mithilfe eines regulären Ausdrucks und der String-Methode `toUpperCase()` in einen Konstantennamen aus Großbuchstaben und Unterstrichen umsetzt und dann die `Calendar`-Konstante als dynamische Property ermittelt. Aus dem Argument `"dayOfMonth"` macht sie also `"DAY_OF_MONTH"` und ermittelt dann den Konstantenwert von `Calendar.DAY_OF_MONTH`. Außerdem fängt sie die `MissingPropertyException` von Groovy ab, falls es den konvertierten Konstantennamen nicht gibt, und löst dann die gleiche Exception mit dem ursprünglichen Property-Namen und dem Namen der `KalenderDatum`-Klasse aus, da andernfalls die Fehlermeldungen sehr verwirrend sind. Nun können wir die Kalenderwerte abrufen, als wären es ganz normale Properties.

```
groovy> k = new KalenderDatum()
groovy> k.setCalendar(new GregorianCalendar(1999,11,12))
groovy> println k.dayOfMonth+'.'+k.month+'.'+k.year
groovy> k.dayOfMonth += 42
groovy> println k.dayOfMonth+'.'+k.month+'.'+k.year
12.11.1999
23.0.2000
```

Ein letztes Problem gilt es allerdings noch zu lösen: Eine Datumsangabe wie 23.0.2000 kann man nicht gerade als intuitiv ansehen. Der `GregorianCalendar` zählt die Monate von 0 bis 11; das würden wir unserem `KalendarDatum` aber gern abgewöhnen. Wir erinnern uns der konventionellen Zugriffsmethoden, die ja Vorrang vor dem generischen `get()` und `set()` haben, und fügen einen Getter und einen Setter für die Monatsangabe hinzu:

```
   def getMonth() {
       calendar.get(Calendar.MONTH) + 1
   }
   void setMonth (wert) {
       calendar.set(Calendar.MONTH,wert-1)
   }
```

Jetzt erfahren alle Zugriffe auf den Monat eine besondere Behandlung, während die Zugriffe auf alle anderen Datumsfelder nach wie vor über unsere generischen Methoden laufen. Um das Ganze rund zu machen, fügen wir auch gleich einen vernünftigen Konstruktor hinzu, der Tag, Monat und Jahr als Zahlen erwartet und das Datum entsprechend setzt, und eine `toString()`-Methode, der wir etwas Sinnvolles entnehmen können.

```
Calendar calendar = new GregorianCalendar(0,0,0,0,0,0)
KalenderDatum(dayOfMonth,month,year) {
   this.dayOfMonth = dayOfMonth
   this.month = month
   this.year = year
}
String toString() {
    "${dayOfMonth}.${month}.${year}"
}
```

Sie sehen an dem Beispiel, wie wir auch innerhalb der Klasse mit den Properties ganz genau so umgehen können, als wären es Felder der Klasse. Eine kurze Überprüfung mit einem interaktiven Test zeigt, dass es auch funktioniert, und zwar mit korrekten Monatszahlen.

```
groovy> k = new KalenderDatum(12,12,1999)
groovy> println k.dayOfMonth+'.'+k.month+'.'+k.year
groovy> k.dayOfMonth += 42
groovy> println k.dayOfMonth+'.'+k.month+'.'+k.year
12.12.1999
23.1.2000
```

Das sieht alles ganz harmlos aus. Aber halten Sie sich mal vor Augen, dass beispielsweise das Erhöhen eines Kalenderdatums um einen Tag in Java mit dem `Calendar` so aussieht:

```
cal.set(Calendar.DAY_OF_MONTH,cal.get(Calendar.DAY_OF_MONTH)+1);
```

Nett, aber haben Sie dasselbe schon mal mit unserer kleinen `KalenderDatum`-Klasse gesehen?

```
k.dayOfMonth ++
```

So können wir die Datumsklasse schon einmal belassen, es wird sich aber zeigen, dass mithilfe von Groovy noch einige weitere reizvolle Verbesserungen möglich sind.

Methoden

Für die Definition von Methoden gelten ähnliche Regeln wie für die Definition von Variablen. Auch hier kann der Typ weggelassen werden, und wenn weder ein Typ noch irgendwelche Typmodifikatoren noch `void` angegeben sind, muss wenigstens das Schlüsselwort `def` die Definition kennzeichnen. Das ist auch häufig der Fall, denn im Unterschied zu Java sind Methoden nicht auf Package-Sichtbarkeit begrenzt, sondern öffentlich, wenn die Sichtbarkeit nicht explizit festgelegt ist.

Fluss ohne Wiederkehr

Achtung, Sie haben soeben den sicheren Boden der konventionellen Java-Programmierung verlassen. Wie Marylin Monroe auf ihrem Floß passieren Sie gerade die Grenze zu einer Welt voller Wunder und Abenteuer, in der vieles möglich ist, was Sie bisher nicht für machbar hielten. Es ist aber auch eine Welt voller Gefahren, in der Dinge passieren, mit denen Sie nie gerechnet hätten. Und es gibt keinen Weg zurück: Dynamische Klassen wie unser `KalenderDatum`, bei denen Sie auf Properties zugreifen können, die es gar nicht gibt, funktionieren in Java nur rudimentär und können dort nicht sinnvoll verwendet werden. Und rechnen Sie damit, dass Ihr Kollege, der von Groovy noch nicht so viel weiß, Ihnen ebenso wenig folgen kann wie Robert Mitchum Marilyn folgen konnte.

Dabei ist dies nur ein Vorgeschmack, in Kapitel 7 werden Sie ganz andere Zaubereien mit Groovy kennenlernen ...

Rückgabewerte

Jede Methode, die nicht als `void` typisiert ist, kann einen Wert zurückgeben. Im Unterschied zu Java muss aber ein Rückgabewert nicht explizit angegeben werden. Endet der Methodendurchlauf nicht mit `return` und einem benannten Rückgabewert, dient einfach das Ergebnis der letzten ausgeführten Anweisung als Ergebnis. Im einfachsten Fall schreiben Sie lediglich das Ergebnis ohne `return` an das Ende der Methode.

In unserer obigen `KalenderDatum`-Klasse haben wir schon stillschweigend davon Gebrauch gemacht:

```
def get(String propertyname) {
    calendar.get(calendarConst(propertyname))
}
def getMonth() {
    calendar.get(Calendar.MONTH) + 1
}
```

Eine `return`-Anweisung ist in beiden Methoden nicht nötig, da ohnehin das Ergebnis der letzten Anweisung zurückgegeben wird, und das ist hier jeweils das Ergebnis des Aufrufs von `calendar.get()`. Wenn die letzte Anweisung kein Ergebnis hat, z.B. weil darin eine `void`-Methode aufgerufen worden ist, oder wenn überhaupt keine Anweisung ausgeführt wurde, ist das Ergebnis `null`. Wichtig ist, dass hier mit »letzter Anweisung« tatsächlich die Anweisung gemeint ist, die unmittelbar vor der schließenden geschweiften Klammer der Methode steht. Wenn diese aufgrund von Verzweigungen, oder weil eine `return`-Anweisung ohne Argument durchlaufen wurde, nicht erreicht wird, ist das Ergebnis `null`.

Wir haben hier eine einfache Methode, die eine Wettervorhersage nach einem Pseudo-Zufallsverfahren als Text generieren soll (sie wird definitiv *nicht* vom Wetterdienst verwendet):

```
def wettervorhersage() {
    if (System.currentTimeMillis() % 3) {
        "Es wird regnen."
    } else {
        "Es wird die Sonne scheinen."
    }
}
```

Die Methode liefert immer `null`, da eine letzte Anweisung vor der schließenden Klammer nicht existiert und daher auch nie erreicht werden kann. (Also gar kein Wetter, in manchen Zeiten durchaus eine brauchbare Prognose, aber nicht das, was wir wollten.) Wir müssen also auf das gute alte `return` zurückgreifen:

```
def wettervorhersage() {
    if (System.currentTimeMillis() % 3) {
        return "Es wird regnen."
    } else {
        return "Es wird die Sonne scheinen"
    }
}
```

Allenfalls könnten wir uns das zweite `return` sparen, da wir den `else`-Zweig eigentlich nicht benötigen.

```
def wettervorhersage() {
    if (System.currentTimeMillis() % 3) {
        return "Es wird regnen."
    }
    "Es wird die Sonne scheinen."
}
```

Unabhängig vom Ergebnis der letzten Anweisung ist der Rückgabewert einer Methode auch dann immer `null`, wenn sie explizit als `void` deklariert worden ist. Insofern ist die Bedeutung des Schlüsselwortes `void` auch etwas anders als in Java: Groovy erlaubt Ihnen, jede Methode aufzurufen, auch eine als `void` deklarierte – allerdings erhalten Sie als Ergebnis dann immer nur `null`.

Das Einsparen der `return`-Anweisung ist nicht unproblematisch. Wenn Sie in Java in einer Methode mit Rückgabewert vergessen, eine `return`-Anweisung mit passendem Argument explizit anzugeben, meldet schon der Compiler einen Fehler. In Groovy merken Sie das unter Umständen überhaupt nicht, da die Methode einfach irgendetwas, und sei es `null`, zurückgibt. Das Problem ist insbesondere bei längeren, tief verzweigten Methoden gravierend, weil Sie die Ausstiegspunkte Ihrer Methode unter Umständen nur noch schwer überblicken können.

An einem einfachen Beispiel lässt sich das Problem demonstrieren. Nehmen wir an, wir bauen eine Methode, die uns sagt, ob ein `KalenderDatum` vor oder nach Weihnachten liegt.

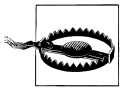
```
def pruefeWeihnachtszeit (KalenderDatum datum) {
    if (datum.month<12 || datum.dayOfMonth<24) {
       return "Es ist Vorweihnachtszeit."
    }
```

```
    if (datum.month==12 && datum.dayOfMonth>24) {
       return "Es ist Nachweihnachtszeit."
    }
}
```

Sie liefert Ihnen immer einen Text, der Ihnen sagt, ob Sie schon an die Beschaffung der Geschenke denken müssen oder noch nicht. Ausgerechnet am 24.12. versagt sie aber, weil der Programmierer an diesen Fall nicht gedacht hat und der Compiler ihn nicht gewarnt hat. Die Methode liefert `null`, Ihr Programm stürzt ab – schöne Bescherung!

Aufrufparameter

Kennen Sie auch jene Klassen, die *n*-fach überladene Methoden mit jeweils bis zu *m* Parametern haben? Wenn Sie sie benutzen wollen, müssen Sie mühsam herausfinden, welche Variante Sie brauchen und in welcher Reihenfolge welche Argumente anzugeben sind. Eigentlich sollen die vielen Varianten dem Benutzer der Methode ermöglichen, bestimmte Argumente wegzulassen, die dann durch Vorgabewerte ersetzt werden. Groovy erleichtert Ihnen hier die Arbeit sowohl beim Programmieren der Methoden als auch beim Aufrufen durch ein paar kleinere syntaktische Leckerbissen.



Es empfiehlt sich keinesfalls, die drei folgenden Möglichkeiten der Aufbesserung von Parameterlisten durch gleichlautende Methodennamen zu mischen. Wenn Sie beispielsweise eine Methode, die Vorgabewerte hat, mit einer anderen überladen, die benannte Parameter hat, ist nur noch schwer zu sagen, welche beim konkreten Methodenaufruf zum Zuge kommt.

Parameter mit Vorgabewerten

In Groovy können Sie die Parameter von Methoden mit einem Default-Wert belegen. Wenn die Parameter mit Vorgabewert beim Aufruf der Methode nicht belegt werden, erhält die Methode jeweils den Vorgabewert übergeben. Ein Beispiel:

```
def vorgabeMethode (Integer param1, param2, param3="Vorgabe", Long param4=null) {
    ...
}
```

Alle Parameter ohne Vorgabewert müssen, sofern es welche gibt, vor den Parametern mit Vorgabewert angeordnet sein. Beim Aufruf dieser Methode müssen Sie die Parameter `param1` und `param2` angeben, die Parameter `param3` und/oder `param4` können Sie weglassen, wenn Sie mit den Vorgabewerten einverstanden sind. Die Reihenfolge muss eingehalten werden, das heißt, Sie können `param3` nur weglassen, wenn Sie auch `param4` weglassen.

Die virtuelle Maschine von Java kennt so etwas nicht. Daher muss sich der Groovy-Compiler bemühen und die notwendigen überladenen Methoden selbst erzeugen. Das Ergebnis sieht ziemlich genau so aus, als hätten Sie Folgendes programmiert:

```
def vorgabeMethode (Integer param1, param2) {
    vorgabeMethode (param1, param2, "Vorgabe", null)
}
def vorgabeMethode (Integer param1, param2, param3) {
    vorgabeMethode (param1, param2, param3, null)
}
def vorgabeMethode (Integer param1, param2, param3, Long param4) {
...
}
```

Achten Sie darauf, dass Sie nicht selbst eine Methode mit derselben Signatur schreiben, die eine der generierten Methoden hat. Der Compiler ignoriert diesen Konflikt stillschweigend und verwendet dann nur die von Ihnen geschriebene Methode.

Variable Parameterlisten

Die mit Java 5.0 verfügbaren variablen Parameterlisten gibt es in Groovy schon länger, allerdings funktionieren sie geringfügig anders und erfordern keine spezielle Syntax wie die drei Punkte in Java. In Groovy genügt es, den letzten Parameter der Methode als Array zu deklarieren. Wenn dann beim Aufruf der Methode das letzte Argument fehlt oder wenn das Argument an derselben Position ist und gegebenenfalls alle noch folgenden Argumente vom Typ des Array-Elements sind, wird der Methode ein Array mit dem letzten und allen noch folgenden Argumenten übergeben. Die Methode erhält also immer ein solches Array, es kann aber auch leer sein. Ein Beispiel sagt mehr als tausend Worte:

```
def variableMethode (String fix, Integer[] variabel) {
    println "$fix - $variabel"
}
```

Wenn wir die Methode interaktiv ausprobieren, erhalten wir folgende Ergebnisse:

```
groovy> variableMethode ("Kein variables Argument")
groovy> variableMethode ("Ein variables Argument",1)
groovy> variableMethode ("Drei variable Argumente",1,2,3)
groovy> variableMethode ("Variables Argument ist null",null)
Kein variables Argument - {}
Ein variables Argument - {1}
Drei variable Argumente - {1, 2, 3}
Variables Argument ist null - null
```

Am vierten Aufruf können Sie die Ausnahme von der Regel erkennen: Wenn an der Stelle der variablen Argumentliste nur `null` übergeben wird, bekommen Sie auch diese Null anstelle des Arrays. Wollen Sie auf Nummer sicher gehen, sollten Sie also den variablen Parameter mit einem `assert` abprüfen.

»Benannte« Parameter

Groovy kennt auch so etwas wie benannte Parameter, die sich schon in vielen Programmiersprachen – außer in Java – bewähren durften. Bei benannten Parametern ist die Reihenfolge der angegebenen Argumente egal, es muss aber jeweils der Name des Parame-

ters mit angegeben werden, wodurch sich insbesondere auch bei langen Parameterlisten Verwechslungen vermeiden lassen. Leider kennt die virtuelle Maschine keine benannten Parameter, und so musste man sich bei Groovy mit einem kleinen Trick behelfen, der den Eindruck erweckt, als würde man die Parameter benennen. In Wirklichkeit instantiiert man nur ein `Map`-Objekt.

```
def benannteMethode ( Map args) {
    println "args: $args"
}
```

Damit die Methode benannte Parameter akzeptiert, muss sie also mit einem Parameter vom Typ `Map` deklariert werden. Beim Aufruf geben Sie dann Argumente in der Form *`name`*`:`*`wert`* an. Das ist genau die gleiche Notation, die Groovy für die Map-Initialisierung verwendet (mehr über die Verwendung von Maps in Groovy in Kapitel 5).

```
groovy> benannteMethode ()
groovy> benannteMethode (arg1:"eins",arg2:"zwei")
args: null
args: ["arg1":"eins", "arg2":"zwei"]
```

Das Problem bei der Methodendeklaration mit einer Map ist, dass ein Anwender der Methode nicht ohne Weiteres erkennen kann, was für Argumente überhaupt erwartet werden. Als Hilfe erweist sich, wenn man in den ersten Zeilen der Methode die fehlenden Parameter initialisiert und dabei gleich kundtut, was für Erwartungen man an den Aufrufenden der Methode stellt. Beachten Sie auch, dass Sie statt der Map nur eine `null` erhalten, wenn gar kein benannter Parameter angegeben ist.

```
def benannteMethode ( Map args) {
   if (args==null) args = [:]
   def arg1 = args.get('arg1',"Erstes Argument")
   def arg2 = args.get('arg2',"Zweites Argument")
   //...
}
```

In diesem Beispiel werden einfach die Map-Elemente in lokale Variablen überführt. Die dabei verwendete `get()`-Methode ist eine für das Interface `Map` vordefinierte Methode, die das benannte Element aus der Map liest und diesen Wert oder, sofern er nicht vorhanden ist, einen Vorgabewert zurückgibt.

Sie können übrigens auch benannte Parameter und Positionsparameter in einer einzigen Methode haben. Die Positionsparameter werden dann am Ende der Argumentliste übergeben und müssen natürlich genau mit der definierten Parameterzahl übereinstimmen.

Methodenaufruf mit dem Spread-Operator

Groovy bietet durch den *Spread-Operator* die Möglichkeit, beim Aufruf von Methoden gleich mehrere Parameter zusammen zu übergeben, die in einem `List`-Objekt oder einem Array zusammengefasst sind. Angenommen, Sie wollten in einem XML-Dokument einen Zeitabstand in einer standardkonformen Weise eintragen. Dabei haben Sie den Abstand

aus einem anderen Anwendungsteil in der Form einer Liste übernommen, die die Anzahl Tage, Stunden, Minuten und Sekunden als einzelne Zahlen enthält. Die Methode `newDurationDayTime()` der Klasse `DatatypeFactory` akzeptiert – neben einem Kennzeichen dafür, ob es sich um eine positive oder negative Differenz handelt – diese Werte als einzelne Argumente. Statt beim Methodenaufruf die Werte aus der Liste einzeln anzugeben, können Sie einfach die ganze Liste übergeben. Wir werden dies hier ganz einfach mit `groovysh` demonstrieren.

```
groovy> dtf = javax.xml.datatype.DatatypeFactory.newInstance()
groovy> zeitabstand = [0,3,45,10]
groovy> dur = dtf.newDurationDayTime(true, *zeitabstand)
groovy> println dur
P0DT3H45M10S
```

Die Liste `zeitabstand` enthält eine Liste mit vier Zahlen, die einen zeitlichen Abstand von 0 Tagen, 3 Stunden, 45 Minuten und 10 Sekunden repräsentieren. Im Methodenaufruf `newDurationDayTime()` übergeben wir diese Liste anstelle der korrespondierenden Einzelwerte und stellen ein Sternchen voran – den Spread-Operator. Dies führt dazu, dass die Liste beim Aufruf der Methode aufgelöst und durch ihre Elemente ersetzt wird. Der Spread-Operator lässt sich auf Listen und Arrays anwenden und funktioniert innerhalb der Argumentlisten beim Aufruf von Methoden, Konstruktoren und Closures.

Konstruktoren

Für Konstruktoren gelten dieselben syntaktischen Aufbesserungen wie für normale Methoden, Sie können also Vorgabewerte, variable Parameterlisten, benannte Parameter und den Spread-Operator in der gleichen Weise anwenden. Darüber hinaus gibt es noch zwei weitere Möglichkeiten: benannte Argumente zur Initialisierung von Properties und Argumentlisten.

Property-Initialisierung in Argumentlisten

Bei Konstruktoren können Sie noch eine zweite Form von benannten Parametern anwenden. Sie kommt allerdings nur dann zum Tragen, wenn in der Klasse ein argumentloser Standardkonstruktor vorhanden ist und kein Konstruktor mit einer Map als Parameter vorhanden ist. In diesem Fall ruft Groovy den Standardkonstruktor auf und setzt anschließend die als benannte Argumente angegebenen Properties. Das bedeutet, dass es zu jedem benannten Argument eine gleichnamige Property in der Klasse geben muss.

Erinnern Sie sich an unsere weiter oben als Beispiel für den Umgang mit Properties eingeführte Klasse `KalenderDatum`:

```
class KalenderDatum {
   Calendar calendar = new GregorianCalendar(0,0,0,0,0,0)
   KalenderDatum(dayOfMonth,month,year) {
      this.dayOfMonth = dayOfMonth
```

```
        this.month = month
        this.year = year
    }
...
}
```

Wir sehen, dass wir den Konstruktor eigentlich gar nicht bräuchten; wir können ihn aber auch stehen lassen, da er nicht weiter stört und für ein schnelles Instantiieren ganz handlich sein kann. Vorausgesetzt, wir fügen noch einen Standardkonstruktor hinzu, können wir die Klasse auch so instantiieren:

```
groovy> maifeiertag = new KalenderDatum(year:2007,month:5,dayOfMonth:1)
groovy> println maifeiertag
1.5.2007
```

Damit ersparen wir uns das getrennte Instantiieren der Klasse und das anschließende einzelne Setzen von Properties, wie es insbesondere beim Umgang mit Beans häufig erforderlich ist und wie es auch bei diesem typischen Ausschnitt aus einem Swing-Programm zu sehen ist:

```
// Java
JButton startButton = new JButton("Start");
startButton.setMnemonic('S');
startButton.setEnabled(false);
startButton.setAction(buttonAction);
buttonPanel.add(startButton);
```

In Groovy sieht das aufgrund des praktischen Konstruktoraufrufs viel übersichtlicher aus:

```
buttonPanel.add(new Button(
                text:"Start",
                mnemonic:'S',
                enabled:true,
                action:buttonAction))
```

Instantiieren mit Listen

Wie erwähnt, können Sie auch in Konstruktoren den Spread-Operator (*`*MeineListe`*) anwenden und damit mehrere Argumente auf einen Schlag als Liste oder Array übergeben. Noch weiter gehend ist aber die Möglichkeit, Listen direkt als Konstruktoren von Objekten zu benutzen.

Als Beispiel kommt der Konstruktor mit den Positionsparametern in unserem `KalenderDatum` noch einmal zu Ehren.

```
groovy> tag = [1,1,2008]
groovy> KalenderDatum neujahr = tag
groovy> println neujahr
1.1.2008
```

Wenn Sie versuchen, einer typisierten Variablen, die nicht mit `List` kompatibel ist, eine Liste zuzuweisen, wendet Groovy einen seiner Mechanismen zur Typanpassung an. Es

sucht nach einem Konstruktor, dessen Parameterliste genau zu den Elementen des übergebenen `List`-Objekts passt, und ruft ihn mit den Werten der Liste als Parameter auf. Das funktioniert natürlich nur, wenn die Zielvariable typisiert ist, denn irgendwoher muss Groovy wissen, an welcher Klasse der Konstruktor überhaupt zu suchen ist. Alternativ können Sie aber auch mit dem Schlüsselwort `as` die Typanpassung erzwingen, dann kann die Zielvariable auch untypisiert sein.

```
groovy> def neujahr = tag as KalenderDatum
```

Operatoren überladen

Groovy kann, was Java nicht kann und was von vielen als eine der wesentlichen Schwächen und von anderen wiederum als wesentlicher Vorteil von Java angesehen wird: Operatoren überladen. Und das noch auf eine geradezu frappierend einfache Weise. Wie vieles in Groovy lädt auch dies dazu ein, Unfug anzustellen (und das ist auch genau der Grund, warum die Java-Erfinder es sich verkniffen haben). Aber Groovy ist für Erwachsene, und die können schließlich selbst entscheiden, was gut für sie ist.

Das Überladen von Operatoren ist vor allem dann sinnvoll, wenn sie zum Anwendungsgebiet passen. Gutes Beispiel hierfür ist die Finanzmathematik mit ihren speziellen Rundungsvorschriften, die von Java-Standardtypen nicht unterstützt werden. In Groovy kann man sich seine eigenen Datentypen mit Operator-Overloading schaffen, die es dann ermöglichen, Formeln kompakt anzugeben. Dieser Ansatz ermöglicht daher die leichte Definition von Domain Specific Languages (DSL).

Fast alle von Java bekannten arithmetischen und bitweisen Operatoren sowie noch einige weitere stehen zur Disposition. Jedem von ihnen entspricht eine bestimmte Methode; bei binären (zweistelligen) Operatoren hat sie ein Argument, bei unären (einstelligen) jedoch nicht. Wenn ein Objekt eine dieser Methoden ausführen kann, ist auch der korrespondierende Operator für die Instanzen dieser Klasse definiert. Wenn also beispielsweise an einem Objekt `x` der Methodenaufruf

```
x.plus(y)
```

möglich ist, können Sie in Groovy stattdessen auch schreiben:

```
x + y
```

Nun ist es aber so, dass die numerischen Wrapper-Klassen von Java ebenso wie die Klasse `java.lang.String` über keine Methode `plus()` verfügen, trotzdem soll der +-Operator mit ihnen verwendbar sein. Dieses Problem löst Groovy mithilfe seiner vordefinierten Methoden; sie stellen sicher, dass etwa bei einem String die Methode `plus()` aufgerufen werden kann, obwohl die Klasse `String()` diese nicht implementiert:

```
groovy> println "abc".plus("de")
```

In Kapitel 5 gehen wir noch genauer auf diesen Mechanismus ein.

Tabelle 3-1 zeigt die Operatoren, denen Methoden zugeordnet sind, sowie jeweils einen Ausdruck mit diesem Operator und die Form, wie dieser in einen Methodenaufruf umgesetzt wird. In Anhang A finden Sie im Zusammenhang mit den vordefinierten Methoden eine vollständige Liste, die aufzeigt, für welche der Java-Standardtypen diese Methoden implementiert sind.

Tabelle 3-1: Operatoren und Operatormethoden

Operator	Bezeichnung	Ausdruck	Methodenaufruf
Arithmetische Operatoren			
-	Negation	`-x`	`x.negate()`
+	Addition	`x + y`	`x.plus(y)`
-	Subtraktion	`x – y`	`x.minus(y)`
*	Multiplikation	`x * y`	`x.multiply(y)`
/	Division	`x / y`	`x.divide(y)`
%	Modulo	`x / y`	`x.mod(y)`
++	Inkrement	`x++` `++x`	`x = x.next()`
--	Dekrement	`x--` `--x`	`x = x.previous()`
Binäre Operatoren			
|	Binäres Oder	`x | y`	`x.or(y)`
&	Binäres Und	`x & y`	`x.and(y)`
~	Komplement	`~x`	`x.negate()`
>>	Rechtsverschiebung	`x >> y`	`x.rightShift(y)`
>>>	Vorzeichenlose Rechtsverschiebung	`x >>> y`	`x.rightShiftUnsigned(y)`
<<	Linksverschiebung	`x << y`	`x.leftShift(y)`
Vergleichsoperatoren			
==	Gleichheit	`x == y`	`x.equals(y)`
!=	Ungleichheits	`x != y`	`! x.equals(y)`
<=>	Vergleich	`x <=> y`	`x.compareTo(y)`
>	Größer	`x > y`	`x.compareTo(y) > 0`
<	Kleiner	`x < y`	`x.compareTo(y) < 0`
>=	Größer oder gleich	`x >= y`	`x.compareTo(y) >= 0`
<=	Kleiner oder gleich	`x <= y`	`x.compareTo(y) <= 0`
Sonstige Operatoren			
[]	Index	`x[y]`	`x.getAt(y)`
[]=	Indexzuweisung	`x[y] = z`	`x.putAt(y,z)`

Tabelle 3-1: Operatoren und Operatormethoden (Fortsetzung)

Operator	Bezeichnung	Ausdruck	Methodenaufruf
(Case)	Switch-Case	`switch (x) { case y: }`	`if(y.isCase(x)) {}`
in	Enthalten	`x in y`	`y.isCase(y)`
as	Typanpassung	`x as y`	`x.asType(y)`

Die Tabelle sollte weitestgehend selbsterklärend sein; bezüglich der sonstigen Operatoren wird an anderer Stelle ausführlich auf ihre Anwendung eingegangen.

Projekt: Rechnen mit physikalischen Einheiten

Wer schon mal damit zu tun gehabt hat, weiß, dass das Rechnen mit dimensionierten Größen eine höchst knifflige und fehlerträchtige Angelegenheit sein kann. Weit verbreitete Beispiele sind Währungsbeträge und physikalische Größen. Es gibt Java-Lösungen dafür: Man definiert Klassen für die Beträge, die ihre jeweilige Dimension kennen und die sich gegeneinander umrechnen lassen. Das Problem ist nur, dass auf die Objekte dieser Klassen keine Operatoren angewendet werden können und mathematische Berechnungen dadurch schnell sehr unübersichtlich werden.

In Groovy ist das einfacher. Wir wollen uns ein paar Typen definieren, die mit den verschiedenen Einheiten für Energie rechnen können: Joule (J), Kilowattstunde (kWh), Elektronenvolt (eV), Kilopondmeter (kpm), Kilokalorie (kcal). Als Basisgröße verwenden wir Kilowattstunden; alle Umrechnungen erfolgen in der Weise, dass ein Wert zunächst in kWh und dann in die Zielgröße umgerechnet wird (die daraus resultierende Ungenauigkeit nehmen wir billigend in Kauf). Die anderen Größen lassen sich aufgrund der folgenden Formeln ableiten:

- $1\ \text{eV} = 4{,}45 \cdot 10^{-26}\ \text{kWh}$
- $1\ \text{J} = 2{,}778 \cdot 10^{-7}\ \text{kWh}$
- $1\ \text{kpm} = 2{,}724 \cdot 10^{-6}\ \text{kWh}$
- $1\ \text{kcal} = 1{,}163 \cdot 10^{-3}\ \text{kWh}$

Mit Objekten dieser Typen soll man normal in Formeln rechnen können, und bei einer Zuweisung zwischen Objekten verschiedener Typen soll automatisch eine Umrechnung erfolgen, z.B. soll Folgendes möglich sein:

```
def v1 = new KWh(1)       // 1 kWh
Joule v2 = [2]            // 2 Joule über List-Konstruktor
def v3 = (v1*2) as Joule  // 2 * 1 kWh, in Joule umgerechnet
def v4 = (new Kcal(1000) + new KWh(1))*3 // 1000 Kcal + 1 KwH, multipliziert mit 3
```

Wir implementieren dazu eine abstrakte Klasse `WertMitEinheit` für dimensionierte Werte (Beispiel 3-3). Die einzelnen Ausprägungen (hier nur für kWh, Joule und kcal realisiert)

brauchen nur noch im Konstruktor das Symbol für die Einheit und den Umrechnungsfaktor anzugeben.

Beispiel 3-3: Die Klasse WertMitEinheit

```
protected abstract class WertMitEinheit {

  final String symbol   // Symbol für die Dimension, z.B. "kWh"
  final Number faktor   // Umrechnungsfaktor vom normalisierten Wert
  final Number wert     // Anzahl der Einheiten

  // Konstruktor für Symbol, Umrechnungsfaktor und Wert
  // Dabei kann Wert ein Skalar oder ein anderer WertMitEinheit sein
  protected WertMitEinheit(String symbol, Number faktor, arg) {
    this.symbol = symbol
    this.faktor = faktor
    switch (arg) {
      case Number:
        // Wenn es ein Skalarwert ist, als Wert übernehmen
        wert = arg
        break;
      case WertMitEinheit:
        // Wenn es ein anderer WertMitEinheit ist, umrechnen
        normalwert = arg.normalwert * faktor
        break
      default:
        // Falscher Argumenttyp
        throw new IllegalArgumentException()
    }
  }

  // Liefert den normalisierten Wert
  public getNormalwert() {
      wert / faktor
  }

  // Setzt den Wert anhand des Normalwerts
  public void setNormalwert (Number arg) {
      wert = arg * faktor
  }

  // Wandelt den Wert in einen anderen WertMitEinheit um
  // Wenn es derselbe Typ ist, können wir this zurückgeben,
  // da WertMitEinheit immutabel ist
  final asType(Class type) {
    if (type==this.class) {
      return this
    } else {
      return neuerWert(type,this)
    }
  }
```

Beispiel 3-3: Die Klasse WertMitEinheit (Fortsetzung)

```
  // Addiert anderen WertMitEinheit
  final plus(WertMitEinheit e) {
    return neuerWert (this.class, this.wert + e.normalwert * faktor )
  }

  // Subtrahiert anderen WertMitEinheit
  final minus(WertMitEinheit e) {
    return neuerWert (this.class, this.wert - e.normalwert * faktor )
  }

  // Multipliziert mit Skalar
  final multiply (Number n) {
    return neuerWert(this.class, this.wert*n)
  }

  // Dividiert durch Skalar
  final divide (Number n) {
    return neuerWert(this.class, this.wert/n)
  }

  // Erzeugt per Reflection eine neue Instanz
  // des angegebenen Typs mit Konstruktor-Argument
  private WertMitEinheit neuerWert(Class type, wert) {
      return type.getConstructor(Object).newInstance(wert)
  }

  // Anzeige Wert und Symbol
  String toString() {
    "$wert $symbol"
  }
}

// Kilowattstunde soll Normalwert sein
class KWh extends WertMitEinheit {
  public KWh (wert) {
    super('kWh',1,wert)
  }
}

// 1 Kilowattstunde sind 3,6E6 Joule
class Joule extends WertMitEinheit {
  public Joule (wert) {
    super('J',3.6e6,wert)
  }
}

// 1 Kilowattstunde sind 860,1 Kilokalorien
class Kcal extends WertMitEinheit {
  public Kcal (wert) {
    super('kcal',860.1,wert)
  }
}
```

Der Konstruktor von `WertMitEinheit` nimmt neben dem Absolutwert auch einen beliebigen anderen `WertMitEinheit` an und wandelt dessen Wert anhand der beiden Umrechnungsfaktoren um – die virtuelle Property `normalwert` hilft bei der Umrechnung.

Beachten Sie aber insbesondere die Operatormethoden `plus()`, `minus()`, `multiply()` und `divide()`, mit denen die Grundrechenarten implementiert sind, sowie `asType()`, die die Umwandlung in einen anderen `WertMitEinheit`-Subtyp ermöglicht.

Das Einfachste ist jetzt, diese vier Groovy-Klassen in einer Datei zu speichern und mit `groovyc` zu übersetzen, wobei vier getrennte *.class*-Dateien entstehen. Wenn wir sie nicht übersetzen, sondern als Skript verwenden wollen, müssen wir sie als vier getrennte *.groovy*-Dateien unter ihrem jeweiligen Klassennamen speichern, damit die Klassen zur Laufzeit gefunden werden können. Die folgenden interaktiven Experimente funktionieren in beiden Fällen.

```
groovy> def v1 = new KWh(1)
groovy> println v1
groovy> Joule v2 = [2]
groovy> println v2
groovy> def v3 = (v1*2) as Joule
groovy> println v3
groovy> Joule v4 = [v1*2]
groovy> println v4
groovy> def v5 = (new Kcal(1000) + new KWh(1))*3
groovy> println v5
1 kWh
2 J
7.2E+6 J
7.2E+6 J
5580.3 kcal
```

Die Ergebnisse stimmen, prüfen Sie sie ruhig nach. Sie sehen, dass man mit diesen dimensionierten Werten schon ganz gut rechnen kann. Die `WertMitEinheit`-Klasse ist ganz abstrakt geraten, sie kann auch für diverse andere vergleichbare Umrechnungsprobleme verwendet werden. Zwei Dinge stören freilich noch: die umständlichen Konstruktoraufrufe in den Formeln und die Tatsache, dass `WertMitEinheit` nur mit Größen arbeiten kann, die multiplikativ abbildbar sind. Damit scheitern wir schon bei einer einfachen Umrechnung von Fahrenheit nach Celsius. Aber auch diese beiden Schwächen lassen sich noch mit den Mitteln beseitigen, die wir in Kapitel 7 behandeln.

Sichere Objektnavigation und GPath

Groovy bietet einige Erleichterungen beim Navigieren durch Objektnetze an. Mithilfe von zwei zusätzlichen Operatoren und einiger Unterstützung durch das GDK ist es möglich, Navigationsausdrücke in der Art zu bilden, wie es sie unter der Bezeichnung *XPath* für XML-Dokumente gibt; bei Groovy heißen sie natürlich GPath. Wie das funktioniert, demonstrieren wir zunächst einmal an einem einfachen Beispiel, einer Liste mit URLs, an der wir einige Untersuchungen vornehmen wollen.

```
groovy> bookmarks = [new URL('http://groovy.codehaus.org'),
    new URL('http://www.google.de/search?hl=de&q=Groovy&btnG=Google-Suche&meta='),
    new URL('http://de.wikipedia.org/wiki/Groovy'),
    new URL('mailto:info@oreilly.de')]
```

Dereferenzieren von Listen

Als Erstes wollen wir sehen, welche Protokolle in den URLs vertreten sind. Anstatt die Listenelemente alle einzeln durchzugehen und bei jedem einzelnen die Methode `getProtocol()`, nutzen wir die angenehme Eigenschaft aller Listen, alle Member- und Property-Anfragen sowie Methodenaufrufe, die sie selbst nicht zufriedenstellen kann, an ihre Listenelemente weiterzuleiten und die Ergebnisse wiederum ordentlich verpackt als Liste zurückzuliefern. Da eine `List` keine Methode `getProtocol()` kennt, können wir diese also direkt bei der Liste aufrufen:

```
groovy> println bookmarks.getProtocol()
["http", "http", "http", "mailto"]
```

Dasselbe ginge natürlich auch in Form der Property-Abfrage `bookmarks.protocol`. Um die Mehrfachnennungen zu entfernen, können wir auf das Ergebnis noch die vordefinierte Listenmethode `unique()` anwenden.

```
groovy> println bookmarks.protocol.unique()
["http", "mailto"]
```

Der Spread-Dot-Operator

Nun interessiert uns, in welchen der URLs das Wort »Groovy« vorkommt. Wir schreiben:

```
groovy> println bookmarks.toString().contains('Groovy')
true
```

Das Ergebnis überrascht jetzt etwas. Wir hätten analog zum ersten Versuch natürlich erwartet, dass wir eine Liste von Wahrheitswerten bekommen. Das Problem ist, dass die Methode `toString()` auch für die `List` definiert ist und daher dort aufgerufen wird. Dabei entsteht ein String, der auch das Wort »Groovy« enthält. Das Ergebnis ist korrekt, aber so nicht erwartet. Damit wir auch in diesen Fällen auf die Elemente der Liste zugreifen können, bietet Groovy einen speziellen, aus einem Stern und einem Punkt (`*.`) bestehenden Operator, der auch als *Spread-Dot-Operator* bezeichnet wird. Hinter der Referenz auf eine Liste sorgt er dafür, dass die nachfolgende Referenzierung eines Felds, einer Property oder einer Methode sich nicht auf die Liste selbst, sondern auf deren Elemente bezieht.

```
groovy> println bookmarks*.toString()*.contains('Groovy')
[false, true, true, false]
```

Auch hinter dem `toString()`-Aufruf müssen wir einen Stern einfügen, denn `bookmarks*.toString()` liefert wiederum eine Liste von Strings, und der `contains()`-Aufruf würde sich sonst auf die Liste beziehen und prüfen, ob eines ihrer Elemente der String »Groovy« ist – und schlicht `false` liefen.

Sicheres Dereferenzieren mit ?.

Angenommen, wir wollten alle unsere URLs durchgehen und deren Query-Strings ausdrucken, wobei diese allerdings an den &-Zeichen in ein String-Array aufgespalten werden sollen. Das geht beispielsweise in einer Schleife:

```
groovy> for (url in bookmarks) { println url.query.split('&') }
Exception thrown: java.lang.NullPointerException: Cannot invoke method contains() on null
object
```

Autsch, das ging schief. Warum? Die Methode `getQuery()` liefert `null`, wenn in der URL kein Query-String enthalten ist. Und beim Aufruf von `split()` an einer Nullreferenz wird natürlich eine `NullPointerException` ausgelöst. Ein Mittel dagegen besteht darin, vor dem kritischen Punkt im Pfad ein Fragezeichen einzufügen. Wenn das Dereferenzieren des Pfads an dieser Stelle den Wert `null` ergibt, wird dieser Vorgang abgebrochen und gleich `null` zurückgegeben.

```
groovy> for (url in bookmarks) { println url.query?.split('&') }
null
{"hl=de", "q=Groovy", "btnG=Google-Suche", "meta="}
null
null
```

Nun verzichtet Groovy darauf, wenn der Aufruf von `getQuery()` den Wert `null` liefert, auch noch `split()` aufzurufen und damit die Exception zu provozieren.

Diese Fragezeichennotation für Navigationspfade in Objektstrukturen ist vor allem dann außerordentlich praktisch, wenn man an vielen Stellen mit Nullreferenzen rechnen muss. Angenommen, Sie hätten in Java einen Pfad wie diesen:

```
x = getRef1().getRef2().getRef3().toString();
```

Wenn jede dieser Referenzen null sein kann, müssen Sie schreiben:

```
if (getRef1()!=null && getRef1().getRef2()!=null
          &&getRef1().getRef2().getRef3()!=null) {
   x = getRef1().getRef2().getRef3().toString();
} else {
   x = null;
}
```

Eine Alternative ist das Einkleiden in einen Try-Catch-Block:

```
try {
    x = getRef1().getRef2().getRef3().toString();
} catch (NullPointerException x) {
    x = null;
}
```

Dieses Vorgehen ist aber eher problematisch, weil Sie es nicht mehr bemerken können, wenn eine der Getter-Methoden auf dem Navigationspfad aus irgendeinem anderen Grund eine `NullPointerException` auslöst. Da sind Sie mit groovy immer auf der sicheren Seite, wenn Sie einfach schreiben:

```
x = getRef1()?.getRef2()?.getRef3()?.toString()
```

Oder in der Property-Notation:

```
x = ref1?.ref2?.ref3?.toString()
```

Projekt: Genealogie

Der Umgang mit langen GPath-Ausdrücken lässt sich am besten an umfangreichen Objektnetzen demonstrieren, z.B. an einem Familienstammbaum. Um eine einfache Ahnentafel oder einen Familienstammbaum aufzubauen, genügt eigentlich eine ganz einfache Klasse, die wir `Person` nennen und die in Beispiel 3-4 abgebildet ist.

Beispiel 3-4: Die Klasse Person

```
class Person {
    def name
    def vater, mutter
    def kinder = []
    def getEltern() {
        [vater,mutter].findAll {it != null}
    }
    void setVater(person) {
        vater = person
        vater.kinder.add(this)
    }
    void setMutter(person) {
        mutter = person
        mutter.kinder.add(this)
    }
    String toString() {
        name
    }
}
```

Die `Person` hat einen Namen, zwei Referenzen auf Vater und Mutter, eine Liste mit Kindern, eine Quasi-Property `eltern`, die eine Liste aus Vater und Mutter bildet und dabei die Null-Objekte gleich herausfiltert. Außerdem hat sie noch zwei Setter, die nicht nur Vater oder Mutter setzen, sondern die Person gleich noch bei diesen als Kind eintragen, sowie eine `toString()`-Methode, die den Namen liefert. Das genügt erst einmal.

Ein ganzes Netz von Eltern-Kind-Beziehungen können wir aufbauen, indem wir die Personen in folgender Weise instantiieren:

```
wilhelm = new Person(name:'Wilhelm')
elfriede = new Person(name:'Elfriede')
otto = new Person(name:'Otto')
henriette = new Person(name:'Henriette')
karl = new Person(name:'Karl', vater:wilhelm, mutter:elfriede)
hanna = new Person(name:'Hanna', vater:otto, mutter:henriette)
helene = new Person(name:'Helene', vater:wilhelm, mutter:elfriede)
```

```
egon = new Person(name:'Egon', mutter:henriette)
franz = new Person(name:'Franz', vater:karl)
ina = new Person(name:'Ina',vater:karl)
kurt = new Person(name:'Kurt',vater:karl, mutter:hanna)
heinz = new Person(name:'Heinz',mutter:hanna)
gigi = new Person(name:'Gigi',mutter:ina)
klaus = new Person(name:'Klaus',mutter:ina)
manuela = new Person(name:'Manuela',vater:kurt)
jeanette = new Person(name:'Jeanette',vater:heinz)
```

Hieran können wir nun verschiedene verwandtschaftliche Beziehungen durch Navigation durch das Geflecht abfragen.

```
groovy> println 'Großeltern von Kurt'
groovy> println kurt.eltern.eltern
groovy> println 'Geschwister von Kurt'
groovy> println (kurt.eltern.kinder.unique() - kurt)
groovy> println 'Onkel und Tanten von Kurt'
groovy> println (kurt.eltern.eltern.kinder.unique() - kurt.eltern)
groovy> println 'Cousins von Manuela'
groovy> println ((manuela.eltern.eltern.kinder.unique()-manuela.eltern).kinder)
groovy> println 'Neffen und Nichten von Kurt'
groovy> println ((kurt.eltern.kinder.unique() - kurt).kinder.unique())
Großeltern von Kurt
[Wilhelm, Elfriede, Otto, Henriette]
Geschwister von Kurt
[Franz, Ina, Heinz]
Onkel und Tanten von Kurt
[Helene, Egon]
Cousins von Manuela
[Gigi, Klaus, Jeanette]
Neffen und Nichten von Kurt
[Gigi, Klaus, Jeanette]
```

Diese Abfragen arbeiten alle mit Listen, die nie leer sind und daher immer funktionieren. Bei den folgenden Abfragen können aber Referenzen null sein, daher ist der Fragezeichenoperator am Platze.

```
groovy> println 'Name des Großvaters mütterlicherseits von Kurt, Egon und Jeannette'
groovy> def grossvaterM(p) { p.mutter?.vater?.name }
groovy> println grossvaterM(kurt)
groovy> println 'Von Egons Mutter ist der Vater nicht bekannt.'
groovy> println grossvaterM(egon)
groovy> println 'Von Jeanette ist die Mutter nicht bekannt.'
groovy> println grossvaterM(jeanette)
Name des Großvaters mütterlicherseits von Kurt, Egon und Jeannette
Otto
Von Egons Mutter ist der Vater nicht bekannt.
null
Von Jeanette ist die Mutter nicht bekannt.
null
```

Sehen Sie, so leicht können Sie auch Ihre Verwandtschaft ergründen. Wenn Sie sich allerdings auch noch Halbschwestern, Schwiegeronkel und Schwippschwager anzeigen lassen möchten, müssen Sie die Klasse `Person` noch etwas erweitern. Dies sei Ihnen zur Übung selbst überlassen.

Nachdem wir uns nun eingehend mit den verschiedenen Möglichkeiten beschäftigt haben, wie man in Groovy Klassen und Skripte – wobei Letztere im Grunde auch Klassen sind – programmieren kann, kennen Sie nun alle wesentlichen Veränderungen und Neuerungen der Sprache Groovy gegenüber Java. Aber Groovy ist nicht nur eine Programmiersprache; dazu gehört auch eine umfangreiche Klassenbibliothek, mit deren Hilfe die neuen Möglichkeiten der Sprache erst richtig zum Tragen kommen. Ihr wenden wir uns ab dem folgenden Kapitel zu. Dabei beschäftigen wir uns zunächst mit denjenigen APIs in der Groovy-Library, die für die meisten Java-Programmierer Neuland sein dürften.

KAPITEL 4

Neue Konzepte

Wenn Sie hier angekommen sind, liebe Leser, sollten Sie bereits einigermaßen vertraut mit Groovy als Programmiersprache sein und wissen, welche Mittel Groovy Ihnen beim Programmieren eigener Klassen und Skripte zur Verfügung stellt. Sie haben ebenfalls schon gemerkt, dass es in Groovy einige Dinge gibt, die in Java bislang nicht bekannt sind, weil sie von der Sprache Java nicht unterstützt werden. In erster Linie sind hier die Closures zu nennen, die sicher den größten Einfluss auf die spezifische Art und Weise der Groovy-Programmierung im Unterschied zu Java haben. Eine weitere wichtige Errungenschaft in Groovy sind die sogenannten *Builder*, die auf Basis der Groovy-Syntax einen semi-deklarativen Programmierstil ermöglichen. Wir gehen in diesem Kapitel etwas ausführlicher auf diese beiden Konzepte ein, denn sie zu kennen ist eine wichtige Grundlage für das Verständnis vieler Klassen in den Groovy-eigenen Bibliotheken.

Closures

Die Closure ist ein Sprachkonstrukt, das in ähnlicher Form – teils unter anderem Namen – auch schon in anderen Programmiersprachen Einzug gehalten hat und einen nachhaltigen Einfluss auf den gesamten Programmierstil ausübt. Tatsächlich stellen sich viele bekannte Patterns der Java-Programmierung ganz anders und oft viel einfacher dar, wenn man Closures zur Problemlösung heranziehen kann.

Es deutet einiges darauf hin, dass Closures in absehbarer Zeit – gemeint ist Java 7.0 – auch Eingang in die Sprache Java finden werden. Wenn wir uns jetzt in Zusammenhang mit Groovy mit Closures beschäftigen, ist dies also auch gleich eine Vorbereitung auf das, was in Java auf uns zukommt.

Closures definieren

Man kann sich eine Closure quasi als einen Behälter für ein Stück Programmcode vorstellen, ähnlich einer Methode, der aber kein Member einer Klasse ist. Closures sind namenlos und können einer Variablen zugewiesen oder einer Methode als Argument übergeben werden. Das englische Wort *closure* kann am ehesten mit »Abschluss« oder auch »Funktionsabschluss« übersetzt werden. Damit ist gemeint, dass mit einem solchen Element gewissermaßen ein offen gelassenes Stück Programm abgeschlossen werden kann. Da das deutsche Wort auch nicht viel aussagekräftiger und dazu nicht sehr weit verbreitet ist, bleiben wir hier bei der englischen Fassung des Begriffs.

Einfache Closures

Die Closure beginnt und endet mit einer geschweiften Klammer, ähnlich einem Anweisungsblock, wie er überall im Code vorkommen kann. Die folgende Programmzeile definiert eine Closure und weist sie einer Variablen zu.

```
groovy> Closure c1 = { println 'Hello world!' }
```

Der Groovy-Compiler generiert intern aus dem Codestück zwischen den geschweiften Klammern eine namenlose Klasse als Erweiterung der abstrakten Klasse `groovy.lang.Closure`. Man kann das Codestück ausführen, indem man die Methode `call()` eines Closure-Objekts aufruft.

```
groovy> c1.call()
Hello World!
```

Sie können es aber auch noch etwas einfacher haben und einfach die Variable, der die Closure zugewiesen worden ist, wie eine Methode behandeln.

```
groovy> c1()
Hello World!
```

Closures mit Parametern

Closures können auch Parameter haben. Die Parameter werden, durch Komma getrennt, nach der öffnenden geschweiften Klammer angeordnet und vom auszuführenden Rumpf durch einen Bindestrich und ein Größer-Zeichen (->) separiert. Die Parameter werden genau so deklariert wie Methodenparameter, können untypisiert sein und Vorgabewerte haben.

```
groovy> def c2 = { s1, s2 -> println s1 + ' ' + s2 +'!' }
groovy> c2('Hello','World')
Hello World!
```

Wenn die Closure nur einen Parameter hat, brauchen Sie ihn nicht zu benennen; er hat dann standardmäßig den Namen `it`. Rufen Sie eine solche Closure ohne Parameter auf, hat `it` den Wert `null`.

```
groovy> def c3 = { println 'Hello ' + it +'!' }
groovy> c3('World')
Hello World!
groovy> c3()
Hello null!
```

Sie können Closures genau wie Methoden auch mit Default-Werten für die Parameter versehen. Die folgende Abwandlung des obigen Beispiels `c2` gibt »World« aus, wenn der zweite Parameter nicht angegeben ist:

```
groovy> def c2a = { s1, s2='World' -> println s1 + ' ' + s2 +'!' }
groovy> c2a('Goodbye')
Goodbye World!
```

Außerdem kennen auch Closures variable Argumentlisten, bei denen Sie als letzten Parameter einfach ein Array verwenden:

```
groovy> def c4 = { Object[] args -> println "Args: "+args }
groovy> c4(100,'Alpha',new Date())
Args: {100, "Alpha", Mon Jul 16 18:16:07 CEST 2007}
```

Mit Ausnahme der drei letztgenannten Fälle müssen Closures immer mit genau der deklarierten Anzahl von Argumenten aufgerufen werden, und wenn Parametertypen angegeben sind, müssen natürlich auch die Typen der Argumente zuweisungskompatibel sein.

Closures, die Methoden referenzieren (Multimethoden)

Für Closures, die nichts weiter machen, als an eine andere Methode zu delegieren, gibt es noch eine andere Notation. Angenommen, wir bräuchten eine Closure, die nur die Aufgabe hat, den übergebenen Parameter mit `System.out.println()` auszugeben. Sie sähe normalerweise so aus:

```
groovy> def c4 = { wert -> System.out.println wert }
```

Das können wir einfacher haben, indem wir eine Closure definieren, die keinen Rumpf enthält, sondern nur einen Verweis auf eine vorhandene Methode. Dazu dient eine spezielle Notation mit einem Punkt und einem Und-Zeichen (`.&`).

```
groovy> def c5 = System.out.&println
groovy> c5("Hallo Welt!")
Hallo Welt!
```

Unser Aufruf von `c5()` ist ohne weitere Umwege an die `println()`-Methode in `System.out` weitergeleitet worden. Vielleicht fällt Ihnen auf, dass `c5` keinerlei Hinweis auf die Parameter der Methode enthält, es kann sich also nicht um eine vollständige Methodensignatur handeln. Und tatsächlich bestimmt Groovy wie vieles andere die aufzurufende Methode erst zur Laufzeit anhand der aktuellen Argumente. Probieren Sie doch einmal:

```
groovy> c5(); println 'OK'

OK
```

Siehe da, es wird nur ein Zeilenvorschub ausgegeben, da mit `c5()` die überladene, argumentlose Methode `System.out.println()` aufgerufen worden ist. Dabei wird die zutreffende Methode erst zur Laufzeit anhand der Anzahl und des Typs der aktuell angegebenen Argumente bestimmt. Die Variable `c5` repräsentiert also eigentlich den Zugriff auf verschiedene Methoden, und dies wird in Groovy als *Multimethode* bezeichnet.

Beachten Sie, dass in einer Methodenreferenz nur auf Instanzmethoden verwiesen werden kann, Referenzen auf statische Methoden sind nicht zulässig. Das obige Beispiel ist insofern korrekt, als `System.out` eine vom System vordefinierte Instanz der Klasse `PrintStream` ist.

Closures, die Closures referenzieren (Currying)

Closures sind Objekte und bieten daher verschiedene Methoden, darunter auch die Methode `curry()`. Sie erzeugt eine Variante der Closure, bei der ein oder mehrere Parameter vorbelegt sind. Erinnern Sie sich an die obige Closure `c2`.

```
groovy> def c2 = { s1, s2 -> println s1 + ' ' + s2 +'!' }
```

Mithilfe der Methode `curry()`, der wir ein Argument mitgeben, bilden wir eine neue Closure, die nur noch ein Argument hat, nämlich `s2`, während `s1` schon belegt ist.

```
groovy> def c2b = c2.curry('Hello')
groovy> c2b('World')
Hello World!
```

Das ist ungefähr äquivalent mit folgender Deklaration einer Closure mit einem Parameter, den sie als zweites Argument an `c2` weiterreicht, während sie das erste Argument für `c2` selbst liefert.

```
groovy> def c2c = { param -> c2.call('Hello',param) }
```

Das funktioniert übrigens auch mit Methodenreferenzen. Die folgende Curry-Closure referenziert die oben schon verwendete Closure `c5`, die ihrerseits die Methode `System.out.println()` referenziert.

```
groovy> def c5 = System.out.&println
groovy> def c5a = c5.curry('Hallo Universum!')
groovy> c5a()
Hallo Universum!
```

Der Begriff *curry* hat nichts mit indischen Gewürzen zu tun, sondern geht auf den amerikanischen Logiker und Mathematiker Haskell Brooks Curry zurück, der sich viel mit funktionaler Programmierung beschäftigt hat. Vor Mr. Curry soll bereits ein russisch-deutscher Mathematiker namens Moses Schönfinkel so etwas wie die `curry()`-Methode als logische Operation erfunden haben; daher wird dieser Vorgang gelegentlich auch als »schönfinkeln« bezeichnet.

Rückgabewerte

Genau wie normale Methoden können auch Closures einen Wert zurückgeben. Wenn kein `return` explizit angegeben ist, wird das Ergebnis der innerhalb der Closure zuletzt ausgeführten Anweisung zurückgegeben. Hat die letzte Anweisung kein Ergebnis, z.B. weil es eine `void`-Methode ist, ist das Ergebnis `null`.

Die folgende Closure berechnet rekursiv die Fakultät einer angegebenen Zahl.

```
groovy> def fakult
groovy> fakult = { val ->
groovy>     val>1 ? val+fakult(val-1) : 1
groovy> }
groovy> println fakult(10)
55
```

Beachten Sie, dass die Variable `fakult` hier nicht erst in derselben Anweisung angelegt werden darf, in der auch die Closure definiert wird, da Sie aus der Closure heraus sonst nicht auf diese Variable zugreifen können.

Am Rande sei auch noch angemerkt, dass Sie den rekursiven Aufruf der Closure aus derselben Closure heraus nicht etwa mit `this.call(val-1)` oder `this(val-1)` bewerkstelligen können, denn die Closure hat, obwohl sie ein eigenes Objekt ist, gewissermaßen keine eigene Identität. Vielmehr verweist `this` auf die umgebende Klasse, hier also auf die Skriptklasse.

Closures anwenden

Nachdem Sie erfahren haben, wie man Closures definiert, werden Sie vielleicht sagen: »Tolle Sache, aber was nützt mir das?« Nun wird Ihnen dieses Syntaxelement fortan in diesem Buch in den verschiedensten Zusammenhängen begegnen, wo Sie selbst feststellen werden, wie einfach und übersichtlich manche Programmieraufgaben durch Closures gelöst werden können. Und am Ende können Sie sich vielleicht gar nicht mehr vorstellen, wie Sie ohne Closures programmieren können. So weit ist es aber noch nicht, daher wollen wir Ihnen an dieser Stelle schon einmal die typischen Situationen vor Augen führen, in denen Closures nutzbringend eingesetzt werden können.

Closures als Methodenparameter verwenden

Die obigen Beispiele sind insofern untypisch für die Anwendung von Closures, als man jedes Beispiel auch mit Hilfe einer Methode hätte implementieren können. Der ganz typische Anwendungsfall für Closures sind Methoden, deren Funktionalität eine Lücke aufweist, die durch die Closure geschlossen wird. Dies ermöglicht es, manche Klassen und Methoden sehr abstrakt zu formulieren, da das für eine bestimmte Situation Spezifische erst in der Anwendung in der Form einer Closure hinzugefügt werden kann.

Das vielleicht am häufigsten verwendete Beispiel dafür ist die Methode `each()`, die Groovy standardmäßig für alle möglichen Containerobjekte, darunter auch Listen, implementiert. Sie geht alle in dem Objekt enthaltenen Elemente durch und ruft für jedes dieser Elemente die in `each()` übergebene Closure auf. Wir simulieren dies in Beispiel 4-1 an einer von `ArrayList` abgeleiteten Beispielklasse mit einer Methode `mitJedem()`, die eine Closure als Argument annimmt.

Beispiel 4-1: Skript zum Experimentieren mit Closures

```
class ListeMitClosure extends ArrayList {
    public mitJedem (Closure c) {
        for ( element in this) {
            c(element)
        }
    }
}

def liste = new ListeMitClosure();
liste.add("Eins")
liste.add("Zwei")
liste.add("Drei")

Closure ausgeben = {println it}
liste.mitJedem(ausgeben)
```

Wir übergeben dem Methodenaufruf von `mitJedem()` hier eine Closure namens `ausgeben`, die ihren Parameter `it` einfach auf der Konsole ausgibt, und tatsächlich erscheinen die drei Elemente der Liste nacheinander auf dem Bildschirm.

```
Eins
Zwei
Drei
```

Der Aufruf von `mitJedem()` ist noch zu umständlich. Natürlich können Sie die Closure auch direkt als Literal angeben.

```
liste.mitJedem({println it})
```

Und wenn die Closure an der letzten Stelle einer Argumentliste steht, kann sie auch aus dem Klammerpaar herausgezogen werden.

```
liste.mitJedem() {println it}
```

Wenn dann zwischen den Klammern nichts mehr steht, können Sie diese außerdem gleich weglassen:

```
liste.mitJedem {println it}
```

Und damit haben Sie die Form, die auch in aller Regel benutzt wird, wenn die Closure das einzige Methodenargument ist.

Die Parametertypen einer Closure abfragen

Wie erwähnt, muss die Anzahl der definierten Parameter einer Closure immer mit der Anzahl der Argumente im Aufruf der Closure übereinstimmen, sofern die Parameter keinen Vorgabewert haben. Sie können aber beim Aufrufen der Closure die Anzahl und die Typen der definierten Parameter berücksichtigen und entsprechend reagieren. Für diesen Zweck hat die Closure eine Methode namens `getParameterTypes()`, die ein `Class`-Array mit den Typen aller definierten Parameter liefert.

```
groovy> def cl = {x,y -> println x+y }
groovy> println Arrays.toString(cl.getParameterTypes())
[class java.lang.Object, class java.lang.Object]
```

Wir möchten, dass in unserer Beispielklasse `ListeMitClosure` der Methode `mitJedem()` auch eine Closure mit zwei Parametern übergeben werden kann; in diesem Fall soll sie im zweiten Parameter den Index des gerade abgearbeiteten Elements erhalten. Dazu ergänzen wir die Methode `mitJedem()` um eine Abfrage der Parameterzahl.

```
class ListeMitClosure extends ArrayList {
    public mitJedem (Closure c) {
        int index = 0
        for ( element in this) {
            if (c.getParameterTypes().length==2) {
                c(element, index++)
            } else {
                c(element)
            }
        }
    }
}
```

Nun können wir in der Listenausgabe auch den Index des jeweiligen Elements mit anzeigen:

```
liste.mitJedem { element, index -> println(index +'. '+element) }
```

wobei der obige Aufruf mit nur einem Parameter immer noch funktioniert.

Closures zum Klassifizieren

Jede Closure hat eine Methode `isCase()` mit einem Parameter, die ihren Aufruf direkt an `call()` weiterleitet und das Ergebnis als `Boolean` zurückgibt. Dies qualifiziert Closures dazu, als `case`-Wert in `switch`-Anweisungen zu dienen. Sie erhalten dann das zu prüfende Objekt und entscheiden, ob der Fall gegeben ist oder nicht, indem sie `true` oder `false` zurückgeben.

Mit dem folgenden Skript können Sie prüfen, ob der Verzehr von frischen Muscheln zur aktuellen Jahreszeit zu empfehlen ist.

```
switch (new Date()) {
case {new java.text.SimpleDateFormat('MMMM').format(it).contains('r')}:
```

```
        println "Muscheln essen erlaubt"
        break
    default:
        println "Muscheln essen nicht erlaubt"
    }
```

Die Methode `isCase()` wird auch in einigen vordefinierten GDK-Methoden angewendet, zum Beispiel in `grep()`:

```
groovy> def liste1 = [1,5,7,2,8,3]
groovy> def liste2 = liste1.grep { it % 2 }
groovy> println liste2
[1, 5, 7, 3]
```

In dem `grep()`-Aufruf werden alle Elemente in `liste1`, deren Modulo-2-Wert nicht 0 ergibt, in einer neuen Liste zusammengefasst und als Ergebnis zurückgegeben. Mehr über die Funktionsweise der vordefinierten Methode `grep()` erfahren Sie in Kapitel 5.

Closures als Event-Handler

Ein weiterer Anwendungsbereich, in dem Closures äußerst effizient eingesetzt werden können, ist das Event-Handling, wie es insbesondere bei der Oberflächenprogrammierung mit Swing Anwendung findet. Statt wie in Java spezielle Event-Handler zu programmieren, können Sie einfach dem Quellobjekt des Events eine Closure zuweisen. Beispiel 4-2 macht dies deutlich:

Beispiel 4-2: Closure als Event-Handler

```
import javax.swing.*

def frame = new JFrame(title="Testfenster")
frame.defaultCloseOperation = WindowConstants.DISPOSE_ON_CLOSE
def button = new JButton("Zum Beenden hier drücken!")
button.actionPerformed = {
      if (! JOptionPane.showConfirmDialog(frame,"Wirklich beenden?")) {
          frame.dispose()
      }
}
frame.contentPane.add(button)
frame.pack()
frame.visible = true
```

Dieses kurze Skript öffnet ein Fenster mithilfe eines Buttons. Die Aktion, die beim Anklicken des Buttons ausgeführt werden soll, ist in einer Closure definiert, die einer Property `actionPerformed` des `JButton` zugewiesen wird. Was an dieser Stelle genau geschieht (der `JButton` hat ja bekanntlich gar keine Property `actionPerformed`), ist ein typisches Stück Groovy-Magie.

Closures als Threads

Closures implementieren von Hause aus das Interface `Runnable` und können somit auch asynchron ausgeführt werden. In dem folgenden Beispiel starten wir zwei Closures als parallel laufende Threads.

```
groovy> Thread.start { ('A'..'Z').each {sleep 100; println it} }
groovy> Thread.start { (1..26).each {sleep 100; println it} }
A
1
B
2
```

Jeder Thread schreibt eine Liste von Buchstaben bzw. Zahlen, wobei er bei jedem Durchlauf eine kurze Pause einlegt, damit jeweils der andere Thread zum Zug kommen kann – und damit wir das Ganze besser verfolgen können.

Mit Closures und Maps beliebige Interfaces implementieren

Eine Closure kann jedes beliebige Interface implementieren. Sie brauchen nur mit dem Schlüsselwort `as` eine Typanpassung zu erzwingen, und schon werden alle Methodenaufrufe an Ihre Closure weitergeleitet. Sie können diese Möglichkeit dazu nutzen, jenen Typ von Interface zu implementieren, mit dem häufig eine Art Callback-Funktionalität hergestellt wird. In Java benutzt man in diesem Zusammenhang gewöhnlich innere anonyme Klassen.

Ein typisches Beispiel ist das Interface `Comparator`, das in Zusammenhang mit `Comparable`-Klassen benutzt wird, zur Abstraktion von Vergleichsmethoden für Sortierfunktionen dient und dessen einzige hier relevante Methode `compare()` heißt. Das folgende Skriptbeispiel sortiert eine Liste mit String-Elementen ohne Beachtung der Groß- und Kleinschreibung:

```
groovy> liste = ["gamma","alpha","BETA"]
groovy> Collections.sort(liste, {x,y -> x.compareToIgnoreCase(y)} as Comparator)
groovy> println liste
["alpha", "BETA", "gamma"]
```

Der statischen `sort()`-Methode von `Collections`, die beliebige Listen sortiert, kann als zweites Argument ein `Comparator` für die Sortierung mitgegeben werden. Wir verwenden dazu einfach eine Closure mit zwei Argumenten, die einfach die `String`-Methode `compareToIgnoreCase()` aufruft und sich als Closure verkleidet.

Sie können mit einer Closure natürlich auch ein Interface mit mehreren Methoden implementieren; der Closure werden einfach alle Methodenaufrufe übergeben. In diesem Fall empfiehlt es sich, als Parameter ein Objekt-Array (`Object[]`) zu verwenden, sodass eine beliebige Anzahl von Parametern beliebigen Typs übergeben werden kann. Mit einer solchen Closure kann man allerdings nicht allzu viel anfangen, da es keine Möglichkeit gibt, den Namen der aufgerufenen Methode zu ermitteln.

Wenn es gilt, ein Interface mit mehreren Methoden mit Closures zu implementieren, bietet sich noch eine weitere Möglichkeit an. Legen Sie einfach eine Map an, der Sie für jede zu implementierende Methode eine Closure unter dem Methodennamen hinzufügen, und lassen Sie die Map mit `as` auf das Interface abbilden. Sie brauchen dabei nur Closures für diejenigen Methoden zuzuweisen, die auch tatsächlich benötigt werden. Diese Verwendung von Closures entspricht auch dem Vorgehen bei dynamischen Proxies (*java.lang.reflect.Proxy*).

Die folgende Programmzeile erzeugt einen Endlos-Iterator:

```
Iterator it = [hasNext:{true}, next:{1}] as Iterator
```

In diesem Beispiel implementieren wir nur die beiden Methoden `hasNext()` und `next()`, wobei die eine Methode immer `true` und die andere immer die Zahl `1` liefert. Die dritte Methode des Interface `Iterator` wird annahmegemäß nicht benötigt und bleibt daher unimplementiert. Solche Dummy-Implementierungen von Interfaces können sehr praktisch im Zusammenhang mit automatisierten Tests sein, um auf einfache Weise kontrollierte Bedingungen herzustellen. Dabei können die getesteten Programme durchaus normale Java-Programme sein, denn diese »sehen« nur das implementierte Interface und brauchen ansonsten mit Groovy nichts weiter zu tun zu haben.

Bei Redaktionsschluss des Buchs war noch nicht klar, ob Groovy in der Version 1.1 auch die Ableitung von abstrakten Klassen mittels Closures oder Closure-Maps ermöglichen wird. Probieren Sie es doch einfach mal in einem Skript aus:

```
def liste = [get:{int index->100},size:{1}] as AbstractList
println liste.get(1)
println liste.size()
```

Wenn Sie dabei keine `GroovyCastException` bekommen, steht die Möglichkeit inzwischen zur Verfügung.

Closures von innen

Sie werden, wenn Sie mit Groovy arbeiten, sehr viel mit Closures zu tun haben. Sie sind allgegenwärtig im GDK und in den sonstigen Groovy-eigenen Bibliotheken, aber auch in Ihren eigenen Anwendungen werden Sie sehr schnell die Vorteile der kompakten Programmierung mit Closures schätzen lernen. Es ist daher nicht verkehrt, sich die Funktionsweise von Closures etwas genauer anzusehen.

Implementierung

Wenn Sie diese, Ihnen schon vom Anfang des Abschnitts bekannte Closure definieren:

```
groovy> def c2 = { s1, s2 -> println s1 + ' ' + s2 +'!' }
groovy> c2('Hello','World')
```

generiert der Groovy-Compiler daraus eine eigene namenlose Klasse, die die abstrakte Klasse `groovy.lang.Closure` erweitert und ihr eine Methode namens `doCall()` mit dem Rumpf der Closure hinzufügt. Als Java-Programm würden die beiden obigen Skriptzeilen sinngemäß ungefähr folgendermaßen aussehen:

```
// Java
import groovy.lang.Closure;

// Closure-Implementierung
class AnonymeClosure extends Closure {
    // Konstruktor
    public AnonymeClosure (Object owner) {
        super(owner);
    }
    // Closure-Methode
    void doCall(Object s1, Object s2) {
        System.out.println (s1+" "+s2+"!");
    }
}

// Main-Klasse zum Aufruf der Closure
public class ClosureTest {
    public static void main (String[] args) {
        new ClosureTest().runTest();
    }
    void runTest () {
        // Closure anlegen
        Closure c2 = new AnonymeClosure(this);
        // Closure ausführen
        c2.call(new String[]{"Hello","World"});
     }
}
```

Der eigentliche Rumpf der Closure befindet sich in der Methode `doCall()`. Sie überschreibt nicht, wie man erwarten könnte, eine in der Klasse `Closure` definierte abstrakte Methode gleichen Namens, sondern wird aus `call()` dynamisch aufgerufen.

Worauf kann die Closure zugreifen?

Möglicherweise reichen einer Closure nicht die Informationen, die sie als Parameter übergeben erhält. In dem obigen Beispiel zum Event-Handling haben wir schon eine Closure gesehen, die auf eine Variable in ihrer Umgebung zugreift, in diesem Fall auf `frame`. Wir haben bereits festgestellt, dass eine Closure zwar eine eigene Klasse ist, aber keine wirkliche eigene Identität hat. Innerhalb der Closure-Methode verweist `this` auf das Objekt, innerhalb dessen die Closure definiert worden ist, und der Sichtbarkeitsbereich umfasst demzufolge den Kontext der umfassenden Klasse. Sehen wir uns das an einem Beispiel an.

```
def zahl=42;

def c6 = {
    println this.class
```

```
    println 'Die Zahl ist '+zahl
}
c6()
println c6.class
```

Nehmen wir an, wir speichern dieses Skript unter dem Namen *TestClosure1.groovy* und rufen es dann aus dem Befehlszeilenfenster auf. Dann sehen Sie auf dem Bildschirm Folgendes:

```
> groovy TestClosure.groovy
class TestClosure1
class TestClosure1
Die Zahl ist 42
class TestClosure1$_run_closure1
```

Wir sehen, dass `this.class` auf die umgebende Klasse verweist, ebenso wie `owner.class`. Das Objekt, in dessen Kontext sich die Closure befindet, in diesem Fall also eine Instanz der Skriptklasse `TestClosure1`, wird der Closure beim Instantiieren übergeben und ist dort als Property `owner` abrufbar. Und `owner` verweist auf das Objekt, an das alle Aufrufe weitergeleitet werden – bis auf ein paar Ausnahmen, darunter die Abfrage der Property `owner`.

Nur so ist es erklärlich, dass die obige Closure auf die außerhalb ihres Sichtbereichs definierte Variable `zahl` zugreifen kann, deren Wert sie in der dritten Zeile ausgibt. Dieser Mechanismus entspricht im Effekt weitgehend den inneren Klassen von Java, die es in Groovy als solche nicht gibt.

Der Aufruf von `c6.class` von außerhalb der Closure zeigt den wirklichen Klassennamen der Closure. Die Weiterleitung der Sichtbarkeit gilt also nur innerhalb der Closure-Methode; die Abfrage von `c6.zahl` würde daher einen Fehler auslösen, denn von außen gesehen hat die Closure keine Property mit dem Namen `zahl`.

Eine Closure kann auch auf lokale Variablen und Parameter zugreifen. Diese werden in einem Kontext gesammelt und der Closure übergeben. Das ist so mit anonymen inneren Klassen in Java nicht möglich, sie können nur auf Attribute der umgebenden Klasse zugreifen. Beispiel:

```
class ClosureTest{
      def createClosure(int i) {
           return {it + i}
      }
}
c = new ClosureTest().createClosure(5)
assert 6, c(1)
```

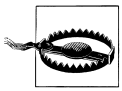
Den Sichtbereich einer Closure modifizieren

Es gibt eine Möglichkeit, Einfluss darauf zu nehmen, worauf eine Closure zugreifen kann. Jede Closure hat zusätzlich zum Eigentümerobjekt eine als *Delegate* bezeichnete Objektreferenz, an die Methoden- und Property-Aufrufe weitergereicht werden, sofern die Closure sie nicht selbst ausführen kann. Normalerweise ist `delegate` identisch mit

owner, es gibt aber die Möglichkeit, ein anderes Objekt als delegate zu benennen. Dazu ein Beispiel:

```
groovy> def c7 = { println substring(12) }
groovy> c7.delegate = "Schöne neue Welt"
groovy> c7()
Welt
```

Die Closure, die wir hier der Variablen c7 zuweisen, ergibt an sich überhaupt keinen Sinn. Sie soll eine Methode substring() aufrufen, aber eine solche Methode gibt es weder in der Closure selbst noch in der umgebenden Klasse, die hier ein Skript ist. Normalerweise würde ein Aufruf der Closure eine MissingMethodException hervorrufen. Vorher weisen wir der Closure aber noch einen String als Delegate-Objekt zu, und das hat zur Folge, dass die Closure den substring()-Aufruf an eben diesen String weiterreicht – und hier wird er dann auch ausgeführt, sodass der Teilstring von »Schöne neue Welt« ab Indexposition 12 ausgegeben wird.



An diesem simplen Beispiel sehen Sie schon, dass Konstrukte wie das Closure-Delegate nicht unbedingt geeignet sind, Ihre Programme übersichtlicher zu gestalten. Sie eröffnen zwar interessante neue Möglichkeiten – eine davon sind die Builder-Klassen, auf die wir weiter unten eingehen –, in normalen Anwendungen sollte man solche Mittel aber – wenn überhaupt – nur vorsichtig und gut dokumentiert anwenden.

Beispiel: Logging

Das Beispiel 4-3 soll einige der Möglichkeiten, die durch Closures gegeben sind, im Zusammenhang zeigen. Wir wollen eine Klasse schreiben, die irgendetwas tut, was uns hier nicht weiter interessieren soll, und dies in einer flexiblen Weise protokollieren. Die Protokollmethode ist einfach durch eine Property definiert, der eine Closure zur Ausgabe des Protokolltexts zugewiesen werden kann.

Beispiel 4-3: Closure als Logger

```
class LoggingWorker {

  Closure log = { stufe, text -> println  (['INFO','FEHLER'][stufe]+': '+text) }

  def tuwas () {
    Log(0, "Tuwas begonnen")
    //...
    Log(1, "Fehler aufgetreten")
    //...
    Log(0, "Tuwas beendet")
  }
}

def worker = new LoggingWorker()
worker.tuwas()
```

Builder

Hierarchische Baumstrukturen sind Dinge, die in der Programmierung immer wieder vorkommen. Geläufige Beispiele hierfür sind die Strukturen von XML-Dokumenten und die innere Anordnung der Elemente grafischer Benutzeroberflächen. Erstaunlicherweise bieten die traditionellen Programmiersprachen wenig systematische Unterstützung dafür, sodass solche Strukturen immer wieder neu entwickelt werden und mit den verfügbaren Mitteln nur umständlich bearbeitet werden können. Mit Groovy können Sie hierarchische Objektstrukturen in einer quasi-deklarativen Weise aufzubauen. Möglich wird dies durch die dynamischen Eigenschaften der Sprache, durch die Methodenaufrufen und -argumenten sowie den Closures neue Bedeutungen verliehen werden können.

Das Prinzip

Eine hierarchische Struktur, wie wir sie hier meinen, besteht aus Knotenelementen, die jeweils beliebig viele untergeordnete Elemente, aber nur ein übergeordnetes Element haben können. Genau ein Element, nämlich das Wurzelelement, hat kein übergeordnetes Element. Jedes Element hat einen Namen und null oder mehrere benannte Attribute, die es beschreiben. Genau dies lässt sich nun akkurat mithilfe der Groovy-Syntax abbilden:

- Der Methodenname ist der Name eines Knotens.
- Die benannten Argumente (als Map) definieren die Attribute des Knotens.
- Ein Closure-Argument bildet eine neue Ebene für untergeordnete Knoten.

Dabei muss der Methodenname natürlich immer vorhanden sein, aber die benannten Argumente und die Closure können auch wegfallen, wenn sie nicht benötigt werden.

Groovy kennt nun eine spezielle Art von Klassen, die als *Builder* bezeichnet werden, die alle Methodenaufrufe, die in dieses Muster fallen, nicht als Aufrufe wirklich existierender oder virtueller Methoden interpretiert, sondern als Definitionen von Knotenelementen in einer Baumhierarchie. Was mit diesen Definitionen dann geschieht, hängt von der jeweiligen Implementierung des Builders ab.

Ein einfaches, konkretes Beispiel für einen solchen Builder ist die Klasse `groovy.util.NodeBuilder`. Sie baut aus den Knotendefinitionen ein Netz von Objekten der Klasse `groovy.util.Node`, einer Allzweckklasse in der Groovy-Bibliothek, die zur Abbildung aller möglichen Baumstrukturen gedacht und auch unabhängig vom `NodeBuilder` einsetzbar ist. Ein `Node`-Objekt hat einen Namen, eine Referenz auf ein übergeordnetes Objekt, eine Map mit den Namen und Werten von Attributen sowie eine Liste mit untergeordneten Objekten. Mit Groovy kommen aber noch ein paar weitere nützliche Klassen, die auf dem Builder-Prinzip beruhen, z.B. ein `SwingBuilder` für den Aufbau von Anwendungsoberflächen und ein `MarkupBuilder` zum Erzeugen von XML- und HTML-Dokumenten. In Kapitel 6 werden wir uns näher mit ihnen beschäftigen.

Beispiel: Taxonomie

Mithilfe des `NodeBuilder` wollen wir nun eine hierarchische Taxonomie der Wildkatzenarten anlegen. Dabei bilden Familie, Unterfamilie, Gattung und Art die Hierarchieebenen. Wir verwenden diese Hierarchieebenen als Elementnamen; alle Elemente haben ein zusätzliches Attribut `name` für den eigentlichen Namen, und auf der Ebene der Arten gibt es noch einige zusätzliche Attribute. Beispiel 4-4 zeigt einen Ausschnitt aus dieser Hierarchie als ein Groovy-Skript namens *Wildkatzen.groovy*.

Beispiel 4-4: Das Skript Wildkatzen.groovy

```
builder = new NodeBuilder()

familie = builder.familie(name:'Felidae') {
    unterfamilie(name:'Acinonychinae') {
        gattung(name:'Acinonyx') {
            art(name:'A. jubatus',bez:'Gepard',wiss:'Schreber',jahr:1775)
        }
    }
    unterfamilie(name:'Felinae') {
        gattung(name:'Caracal') {
            art(name:'C. caracal',bez:'Karakal',wiss:'Schreber',jahr:1776)
        }
        gattung(name:'Catopuma') {
            art(name:'C. badia',bez:'Borneo-Goldkatze',wiss:'Gray',jahr:1874)
            art(name:'C. temminickii',
                    bez:'Asiat. Goldkatze',wiss:'Vigors&Horsfield',jahr:1827)
        }
        gattung(name:'Felis') {
            art(name:'F. bieti',bez:'Graukatze',wiss:'Milne-Edwards',jahr:1892)
            art(name:'F. chaus',bez:'Rohrkatze',wiss:'Schreber',jahr:1777)
            art(name:'F. margarita',bez:'Sandkatze',wiss:'Loche',jahr:1858)
        }
    }
}

new NodePrinter().print(familie)
```

Das Skript legt zuerst eine Instanz des `NodeBuilder` an und ruft dann bei ihm die scheinbare Methode `familie()` auf. Diesen Methodenaufruf fängt der `NodeBuilder` ab und legt stattdessen eine `Node`-Instanz mit dem Namen der Methode, also »familie«, an und setzt bei ihr ein Attribut »name« mit dem Wert »Felidae«. An dieses Wurzelobjekt hängt der `NodeBuilder` nun nach und nach die untergeordneten Objekte, die durch die scheinbaren Methodenaufrufe in den verschachtelten Closures definiert werden.

Am Ende erhalten wir das Wurzelobjekt als Rückgabewert. Indem wir an die `print()`-Methode eine Instanz der Klasse `groovy.util.NodePrinter` übergeben, können wir den ganzen Baum ordentlich formatiert auf der Konsole ausgeben. Wir verzichten hier auf die Wiedergabe des Ergebnisses – es sieht genau so aus wie der Programmcode, mit dem wir den Baum definiert haben.

Navigation im Baum

Im Ergebnis kann man jetzt mit GPath-Ausdrücken navigieren. Beispielsweise können Sie, wenn Sie am Ende des Skripts folgende Zeile hinzufügen, die deutschsprachigen Bezeichnungen aller verzeichneten Katzenarten ausgeben:

```
familie.unterfamilie.gattung.art.each { println it.'@bez' }
```

Die normale Property-Navigation (z.B. `familie.unterfamilie`) liefert immer den oder die Unterknoten mit dem betreffenden Namen. Wenn wir ein Attribut referenzieren wollen, müssen wir ein @-Zeichen vor dem Attributnamen einfügen, wie hier bei `bez`, und da dies keinen gültigen Groovy-Namen ergibt, muss `'@bez'` dann auch noch in Anführungszeichen eingeschlossen werden. (Sie können nicht einfach `it.@bez` schreiben, da Groovy sonst versuchen würde, auf ein Feld der Klasse `Node` mit dem Namen `bez` zuzugreifen, was ja hier nicht erwünscht ist.

Es gibt noch ein paar weitere Besonderheiten beim Zugriff auf die virtuellen Properties eines `Node`-Objekts, Tabelle 4-1 enthält eine Zusammenfassung der entsprechenden Regeln. Dabei sei angenommen, dass die Variable `n` auf den Knoten für die Unterfamilie »Felinae« verweist.

Tabelle 4-1: Behandlung von Property-Zugriffen in Node-Objekten

Form	Beispiel	Bedeutung
name	`n.gattung`	Liefert eine Liste mit allen direkt untergeordneten Knoten, deren Name gleich *name* ist. Im Beispiel sind es die drei Knoten für die Gattungen »Caracal«, »Catopuma« und »Felis«.
`'@`*name*`'`	`n.'@name'`	Liefert den Wert des Attributs mit dem Namen *name*. Im Beispiel ist das Ergebnis der String »Felinae«. Der Ausdruck `n.name` würde dagegen »unterfamilie« liefern.
`'..'`	`art.'..'`	Liefert das direkt übergeordnete `Node`-Objekt oder `null`, wenn das aktuelle Element ein Wurzelknoten ist. Im Beispiel wäre das Ergebnis der Wurzelknoten.
`'*'`	`art.'*'`	Liefert eine Liste mit allen direkt untergeordneten Knoten. Im Beispiel sind es die drei Knoten für die Gattungen »Caracal«, »Catopuma« und »Felis«.
`'**'`	`art.'**'`	Liefert rekursiv alle untergeordneten Knoten. Im Beispiel sind es die Knoten für »Caracal«, »C. caracal« »Catopuma«, »C. badia«, C. temminickii«, »Felis«, »F. bieti, »F. chaus« und »F. margarita«.

Programmieren und Deklarieren

Momentan sind Sie vielleicht noch geneigt zu sagen, dass Sie dies mit einer XML-Datei auch ganz gut hinbekommen hätten – wenn auch mit etwas mehr Schreibarbeit. Das Besondere an den Builder-Klassen in Groovy ist aber etwas anderes: Sie können hier den deklarativen Programmierstil mit normalen prozeduralen Teilen mischen und dadurch Teile der Struktur auch generieren oder in Subroutinen auslagern. Denn alle Groovy-Anweisungen, z.B. Schleifen, und alle Methoden, die nicht vom Builder-Objekt abgefangen werden, stehen Ihnen weiterhin für die normale prozedurale Programmierung zu Verfügung. Im obigen Beispiel stellen wir beispielsweise fest, dass das Schreiben der

untersten Hierarchieebene relativ mühsam ist, weil immer wieder alle Feldnamen redundant angegeben werden müssen. Also schreiben wir im Skript eine Methode, die uns die Arbeit abnimmt.

```
def art(name,bez,wiss,jahr) {
    builder.art(name:name,bez:bez,wiss:wiss,jahr:jahr)
}
```

Die Methode `art()` hat vier Parameter, deren Werte sie einfach als benannte Parameter ebenfalls durch einen `art()`-Aufruf an die Builder-Instanz weitergibt. Nun können Sie diese Zeile:

```
art(name:'F. margarita',bez:'Sandkatze',wiss:'Loche',jahr:1858)
```

ersetzen durch jene:

```
art('F. margarita','Sandkatze','Loche',1858)
```

und sich damit etwas Schreibarbeit ersparen.

Neben dem Ausgliedern von Teilen der Baumdeklaration in getrennte Methoden sind typische Beispiele für die Mischung von deklarativer und prozeduraler Programmierung in den Builder-Objekten Schleifen für die Generierung von Teilbäumen aus irgendwelchen Daten oder die Berechnung der Attributwerte.

Eigene Builder programmieren

Wie erwähnt, umfasst die Groovy-Bibliothek bereits eine ganze Reihe von Builder-Klassen; in Anhang B sind alle mit aufgeführt. Es ist aber auch nicht schwierig, selbst einen Builder zu bauen. In diesem Beispiel wollen wir einen einfachen Builder programmieren, der ein formatiertes Strukturabbild der Baumstruktur direkt auf der Konsole ausgibt. Wir nennen ihn – etwas hochtrabend – `PrettyPrintBuilder`.

Wir brauchen eigentlich nichts weiter zu tun, als eine von `groovy.util.BuilderSupport` abgeleitete Klasse zu definieren, die folgende abstrakte Methoden implementiert.

`Object createNode(Object name)`
`Object createNode(Object name, Map attributes)`
: Diese drei Methoden legen jeweils einen neuen Knoten an und liefern diesen als Ergebnis zurück. Dabei ist `name` der Name der scheinbar aufgerufenen Methode und damit zugleich der Knotenname. Die beiden Methoden unterscheiden sich lediglich darin, dass die eine noch zusätzlich Attribute als Map übergeben erhält.

`Object createNode(Object name, Object value)`
`Object createNode(Object name, Map attributes, Object value)`
: Diese beiden Methoden legen ebenfalls neue Knoten an, haben aber zusätzlich noch ein Argument, das typischerweise für Textknoten verwendet wird.

`void setParent(Object parent, Object child);`
: Diese Methode können Sie implementieren, um einem Knoten einen übergeordneten Knoten zuzuorden.

`protected void nodeCompleted(Object parent, Object node)`
: Wird aufgerufen, wenn die Bearbeitung eines Knotens (mit allen Unterknoten) beendet ist. Im Gegensatz zu den anderen Methoden ist diese nicht abstrakt, muss also nicht zwingend überschrieben werden.

Eine hilfreiche Property des `Node`-Objekts, die Sie benutzen können, ist `current`. Sie verweist auf das aktuelle Knotenobjekt und ermöglicht Ihnen, eigene Verknüpfungen herzustellen.

Beispiel 4-5 zeigt die Quelldatei `PrettyPrintBuilder.groovy` komplett.

Beispiel 4-5: Die Klasse PrettyPrintBuilder()

```
public class PrettyPrintBuilder extends BuilderSupport {

    protected createNode(name) {
        createNode(name,null,null)
    }

    protected createNode(name, value) {
        createNode(name,null,value)
    }

    protected createNode(name, Map attr) {
        createNode(name,attr,null);
    }

    protected createNode(name, Map attr, value) {
        def level = current ? current : 0 // Aktuelle Tiefe
        println "| " * level
        if (level) print "| " * (level-1) + '+-'
        if (name=='art') {
            print "$attr.name - $attr.bez ($attr.wiss, $attr.jahr)"
        } else {
            print "$attr.name (${name[0].toUpperCase()}${name[1..-1]})
        }
        level+1 // Die neue Tiefe wird current-Objekt
    }

    protected void setParent(Object parent, Object child) {
    }
}
```

Wie Sie sehen, erzeugen wir hier keine `Node`- und auch keine sonstigen Objekte, sondern schreiben einfach die Inhalte der Knoten optisch etwas aufbereitet aus. Die `current`-Property, die jeweils den aktuellen Knoten bezeichnet, benutzen wir einfach dazu, die Schachtelungstiefe mitzuführen.

Im Grunde implementieren wir hier nur die Methode `createNode()` mit drei Argumenten; die anderen `createNode()`-Methoden delegieren einfach an diese Methode. Und `setParent()` ist lediglich der Form halber implementiert, da sie in `BuilderSupport` abstrakt definiert ist.

Um unseren neuen Builder auszuprobieren, ändern wir das obige Skript *Wildkatzen.groovy*, sodass es diesen anstelle des `NodeBuilder` verwendet. Korrigieren Sie also die erste Zeile:

```
builder = new PrettyPrintBuilder()
```

und entfernen Sie dann noch die letzte Zeile mit dem `NodePrinter`-Aufruf; den brauchen wir nicht mehr, da unser `PrettyPrintBuilder` ja selbst etwas ausgibt. Wenn wir es nun aufrufen, sieht es so aus:

```
> groovy Wildkatzen
Felidae (Familie)
|
+-Acinonychinae (Unterfamilie)
| |
| +-Acinonyx (Gattung)
| | |
| | +-A. jubatus - Gepard (Schreber, 1775)
|
+-Felinae (Unterfamilie)
| |
| +-Caracal (Gattung)
| | |
| | +-C. caracal - Karakal (Schreber, 1776)
| |
| +-Catopuma (Gattung)
| | |
| | +-C. badia - Borneo-Goldkatze (Gray, 1874)
| | |
| | +-C. temminickii - Asiat. Goldkatze (Vigors&Horsfield, 1827)
| |
| +-Felis (Gattung)
| | |
| | +-F. bieti - Graukatze (Milne-Edwards, 1892)
| | |
| | +-F. chaus - Rohrkatze (Schreber, 1777)
| | |
| | +-F. margarita - Sandkatze (Loche, 1858)
```

Damit haben Sie ein sehr einfaches Beispiel für einen Builder kennengelernt. Es soll deutlich machen, welche Möglichkeiten Ihnen mit einem Builder prinzipiell zur Verfügung stehen und wie einfach es ist, diese Art der semi-deklarativen Programmierung für eigene Zwecke zu nutzen.

Templates

Relativ häufig besteht die Notwendigkeit, irgendwelche Daten, die innerhalb des Programms eingelesen oder berechnet werden, in einem zusammenhängenden Text auszugeben. Ein mögliches Vorgehen dabei ist, diesen zu erzeugenden Text als Muster zu erfassen, in dem für die konkreten Daten spezielle Platzhalter vorgesehen sind. Im Programm wird dann nur noch der Mustertext komplett eingelesen und mit den darzustellenden Daten gemischt, und schon haben Sie das fertig formatierte Resultat. Der Vorteil ist, dass das Formatieren von Texten auf diese Weise viel übersichtlicher ist, als wenn sie stückweise zusammengesetzt werden. Außerdem können Sie die Texte getrennt vom Programm speichern und bei Bedarf ändern, ohne das Programm selbst anfassen zu müssen. Auch mehrsprachige Ausgabeformate sind so leicht zu realisieren.

Besonders beliebt ist dieses Vorgehen bei der Generierung von dynamischen Webseiten; in Java gibt es dafür beispielsweise *JavaServer Pages* oder *Velocity*, und viele andere Programmierumgebungen bieten vergleichbare Werkzeuge, die man generell als *Template-Engines* bezeichnet.

Ein einfaches Beispiel für das Mischen von Text und Daten kennen Sie bereits in der Form der GStrings, die ja ein fester Bestandteil der Programmiersprache Groovy sind. GStrings sind allerdings eine Art von Templates, die nur innerhalb des Programms erzeugt und nicht etwa als Datei eingelesen werden kann.

Da Template-Engines in vielen Situationen nützlich sein können, umfasst die Groovy-Umgebung ein abstraktes Framework für Template-Engines und drei konkrete Ausprägungen für unterschiedliche Anwendungsfälle. Alle eignen sich dazu, Texte und Daten zu mischen, wenden dabei aber unterschiedliche Mechanismen an. Wir werden hier auf die prinzipiellen Mechanismen eingehen und zeigen, wie Sie die Template-Engines in einem Groovy-Programm generell anwenden können. Auf spezifische Anwendungsfälle kommen wir in Kapitel 6 zu sprechen.

Das Vorgehen

Das Mischen, also das Verknüpfen von laufendem Text mit Daten, läuft immer so ab, dass Sie am Anfang einerseits eine Textvorlage mit eingebetteten Platzhaltern und andererseits die dort einzufügenden Programmdaten haben. Das Ergebnis ist ein neuer Text mit dem Inhalt der Textvorlage und den darin eingefügten Programmdaten. Die einzelnen Schritte sind folgende:

1. Sie besorgen sich eine Instanz der zu verwendenden Template-Engine.
2. Die Template-Engine erzeugt aus der Textvorlage das eigentliche Template. Dies ist ein Objekt einer Klasse, die das Interface `groovy.text.Template` implementiert.

3. Sie erzeugen eine Map, in die alle Daten, die dem Template zur Verfügung gestellt werden sollen, unter einem eindeutigen Namen eingetragen werden. Diese Map entspricht dem *Binding* eines Skripts und wird auch so bezeichnet.
4. In einem Aufruf der Methode `make()` übergeben Sie Ihre Daten dem Template, das den eigentlichen Mischvorgang ausführt und als Ergebnis ein `Writable`-Objekt liefert.
5. Das Endresultat in Textform erhalten Sie erst, wenn Sie das `Writable`-Objekt einem Writer übergeben oder einfach seine `toString()`-Methode aufrufen.

Die Templates sind wieder verwendbar, Sie können also die Schritte 3 bis 5 mehrmals hintereinander mit unterschiedlichen Daten ausführen, ohne jedes Mal ein neues Template erzeugen zu müssen.

Die Strategien zur Erzeugung des gemischten Texts können sehr unterschiedlich sein. So kann in einem Fall der gesamte Mischvorgang schon beim `make()` des Templates stattfinden, sodass das `Writable`-Objekt schon den kompletten Text enthält, der dann am Ende nur noch auszugeben ist. Es kann aber auch sein, dass der Mischvorgang überhaupt erst beim Herausschreiben des `Writable`-Objekts geschieht. Letzteres kann sinnvoll sein, wenn der Ergebnistext sehr lang ist, sodass es effizienter ist, wenn er gar nicht erst als kompletter String im Speicher gehalten, sondern gleich beispielsweise in eine Datei geschrieben oder über das Netzwerk versendet wird.

Wir wollen das Vorgehen an einem sehr einfachen Beispiel erläutern. Angenommen, wir programmieren ein einfaches Telefonverzeichnis. Jeder Eintrag in dem Verzeichnis wird im Groovy-Programm durch ein Objekt der folgenden simplen Klasse repräsentiert:

```
class Kontakt {
    def name
    def nummern
}
```

Die Property `name` enthält den Namen einer Person und die Property `nummern` eine Liste mit Strings, in denen jeweils eine Telefonnummer und gegebenenfalls noch eine zusätzliche Bemerkung stehen. Das gesamte Verzeichnis ist wiederum eine Liste aus `Kontakt`-Objekten, die wir in einer Variablen namens `kontaktliste` gespeichert haben. Wir fangen klein an, daher hat unsere Kontaktliste erst einmal nur drei Einträge, wobei es zu einem Eintrag noch gar keine Nummern gibt.

```
kontaktliste = [
    new Kontakt(name:'Klaus Schulz', nummern:['029281 dienstl.','028277 privat']),
    new Kontakt(name:'Jens Krause'),
    new Kontakt(name:'Erika Meier', nummern:['02827'])
]
```

Da sich Kontakte, wie wir wissen, schnell ändern können, haben wir auch den Stand der Liste als Datum in einer Variablen namens `stand`. Im Beispiel ist es einfach das heutige Datum.

```
stand = new Date()
```

Dieses Telefonverzeichnis wollen wir nun formatiert ausgeben. Ohne Template-Engine würden wir dies etwa so bewerkstelligen:

```
df = java.text.DateFormat.getDateInstance()
println " TELEFONVERZEICHNIS vom ${df.format(stand)}"
kontaktliste.each { kontakt ->
    println kontakt.name
    kontakt.nummern.each { println "    $it" }
}
```

Das Ergebnis sieht dann so aus:

```
TELEFONVERZEICHNIS vom Wed May 23 11:12:01 CEST 2007
Klaus Schulz
    029281 dienstl.
    028277 privat
Jens Krause
Erika Meier
    02827
```

Das gleiche Ergebnis können wir auch mit einem Template erzielen, dann brauchen wir den Text nicht programmgesteuert zusammenzusetzen.

Die Template-Syntax

Zunächst wird eine Textvorlage benötigt. Die Form, die Textvorlagen haben müssen, ähnelt sehr den verbreiteten Mustern, wie sie etwa von JSP oder Velocity bekannt sind. Wenn Sie bereits mit einem vergleichbaren Werkzeug gearbeitet haben, werden Sie auch mit den Groovy-Templates keine Schwierigkeiten haben.

Prinzipiell gibt es zwei Arten von Platzhaltern: solche, die dem Einfügen von Daten dienen, und solche, mit denen bestimmte Aktionen ausgelöst werden, zum Beispiel die wahlweise oder wiederholte Ausführung eines Template-Abschnitts. In Groovy werden an diesen Stellen in dem einen Fall die Inhalte von Variablen oder die Ergebnisse von Ausdrücken eingefüllt, im anderen Fall können es beliebige Codeschnipsel sein – die allerdings im Zusammenhang einen Sinn ergeben müssen.

Für das obige Telefonverzeichnis könnte die Textvorlage so aussehen:

```
<% df = java.text.DateFormat.getDateInstance() %>
TELEFONVERZEICHNIS vom <%= df.format(stand) %>
<% liste.each { eintrag -> %>
$eintrag.name
<% eintrag.nummern.each { nummer -> %>
    $nummer
<% } } %>
```

In diesem Beispiel sollen anstelle der durch Dollarzeichen markierten Platzhalter die Inhalte der gleichnamigen Variablen eingetragen werden. Zwischen den spitzen Klammern mit Prozentzeichen befinden sich Teile von Programmcode, die ausgeführt werden

sollen. Hier benötigen wir die Codeabschnitte, um eine `DateFormat`-Instanz zu initialisieren und um die Schleifen formulieren zu können.

Sie sehen schon an diesem sehr einfachen Beispiel, dass die Mischung aus Vorlagentext und Programmlogik nicht unbedingt übersichtlich ist. Insbesondere die korrekte Anordnung der schließenden Klammern ist schwer zu erkennen. Die Anwendung der eingebauten Template-Engines ist daher nur in sehr einfachen Situationen zu empfehlen. Außerdem sollten Sie immer darauf achten, dass Programmlogik und Anzeige sauber getrennt bleiben. Im obigen Beispiel könnte man daher überlegen, die Formatierung des Datums im Programm außerhalb der Vorlage vorzunehmen.

Tabelle 4-2 enthält eine kurze Übersicht der bei den mitgelieferten Template-Engines vorgesehenen Platzhalter.

Tabelle 4-2: Verschiedene Arten von Platzhaltern

Platzhalter	Bedeutung
`$variable` `$variable.property`	Wird durch den Inhalt der Variablen ersetzt. Anstelle des einfachen Variablennamens kann auch ein Ausdruck verwendet werden, dieser darf aber nur Namen und Punkte zur Verknüpfung enthalten.
`${ausdruck}`	Wird durch das Ergebnis des zwischen den geschweiften Klammern stehenden Groovy-Ausdrucks ersetzt. Die geschweiften Klammern sind auch nötig, wenn auf einen Variablennamen direkt ein Buchstabe oder eine Zahl folgen sollen: `${variable}test`
`<%= ausdruck %>`	Wird ebenfalls durch das Ergebnis des Ausdrucks ersetzt. Im Unterschied zu dem Dollar-Platzhalter kann dieser Ausdruck über mehrere Zeilen hinweg reichen.
`<% programmcode %>`	Ein Stück Programmcode, das direkt ausgeführt wird, ohne dass ein Wert in den Ergebnistext einzufügen ist. Kann ebenfalls mehrere Zeilen umfassen.

Arbeiten mit der SimpleTemplateEngine

Wie kommen aber nun Textvorlage und Daten zusammen? Das Vorgehen ist bei allen Template-Engines in Groovy gleich; am Beispiel der `SimpleTextEngine` zeigen wir, wie die einzelnen Schritte vonstatten gehen.

Holen Sie sich zunächst eine Instanz der `TemplateEngine`, die Sie verwenden möchten, in diesem Fall also der `SimpleTemplateEngine`.

```
engine = new SimpleTemplateEngine()
```

Lassen Sie die Template-Engine ein Template aus dem Vorlagentext erzeugen.

```
template = engine.createTemplate(textvorlage)
```

Legen Sie eine Map für die Daten an, die Sie dem Template zum Mischen übergeben möchten.

```
daten = [liste:kontaktliste, stand:new Date()]
```

Stoßen Sie den eigentlichen Mischvorgang an.

```
ergebnis = template.make(daten)
```

Das Ergebnis ist ein `Writable`-Objekt. Sie können es beispielsweise an einen Writer übergeben, um es in eine Datei zu schreiben.

```
new File('Telefonliste-Ergebnis.txt').withWriter { writer ->
    writer.write(ergebnis)
}
```

Oder geben Sie einfach das Ergebnis auf der Konsole aus.

```
println ergebnis
```

Wenn Sie sich das Ergebnis ansehen, werden Sie möglicherweise feststellen, dass es nicht ganz Ihren Erwartungen entspricht. Das Problem besteht darin, dass die Template-Engine zwar den in `<%...%>` eingeschlossenen Groovy-Code entfernt, den Rest des Texts aber gänzlich unverändert lässt. Das hat zur Folge, dass dort, wo ganze Zeilen aus Groovy-Code bestehen, möglicherweise unerwünschte Leerzeilen bestehen bleiben. Hier ist dies auch der Fall:

```
TELEFONVERZEICHNIS vom 23.05.2007

Klaus Schulz

    029281 dienstl.

    028277 privat

Jens Krause

Erika Meier

    02827
```

Um dies zu verhindern, bleibt Ihnen nur eine Möglichkeit – den Vorlagentext so schreiben, dass die Groovy-Codeschnipsel keine ganzen Zeilen einnehmen, etwa so:

```
<% df = java.text.DateFormat.getDateInstance()
%>TELEFONVERZEICHNIS vom <%= df.format(stand) %>
<% liste.each { eintrag ->
%>$eintrag.name
<% eintrag.nummern.each { nummer ->
%>    $nummer
<% } } %>
```

Dies macht die Vorlage aber nicht gerade leichter lesbarer.

Wenn der in der Vorlage eingebettete Groovy-Code Fehler enthält, sind diese unter Umständen sehr schwer zu finden. Wie erwähnt, wandelt die `SimpleTemplateEngine` die Vorlage in ein Groovy-Skript um, das beim Aufruf der `make()`-Methode des Templates ausgeführt wird. Wenn dabei Fehler auftreten, verweisen die angegebenen Zeilennummern auf dieses intern generierte Skript – und helfen Ihnen daher herzlich wenig.

Um fehlerhafte Vorlagentexte debuggen zu können, bietet die `SimpleTemplateEngine` die Möglichkeit, sie in einem »gesprächigen« Modus laufen zu lassen. Dazu müssen Sie die Engine mit einem `true` als Argument instantiieren.

```
engine = new SimpleTemplateEngine(true)
```

Dann gibt sie bei jedem Erzeugen eines Templates den generierten Groovy-Code auf der Konsole aus. Den Fehler hier zu finden ist jetzt wesentlich einfacher.

Sehen wir uns das kurze Skript, mit dem wir das Telefonverzeichnis erstellen, noch einmal im Zusammenhang an. Beispiel 4-6 enthält das entsprechende Listing.

Beispiel 4-6: Skript zum Mischen von Daten

```
/**
 * Skript zum Erzeugen einer Telefonliste mit SimpleTemplateEngine
 */

// Die Klasse Kontakt enthält die Telefonnummern zu einer Person
class Kontakt {
    def name
    def nummern
}

// Liste der Kontakte initialisieren
kontaktliste = [
    new Kontakt(name:'Klaus Schulz', nummern:['029281 dienstl.','028277 privat']),
    new Kontakt(name:'Jens Krause'),
    new Kontakt(name:'Erika Meier', nummern:['02827'])
]

// Die Textvorlage anlegen
textvorlage = '''
<% df = java.text.DateFormat.getDateInstance()
%>TELEFONVERZEICHNIS vom <%= df.format(stand) %>
<% liste.each { eintrag ->
%>$eintrag.name
<% eintrag.nummern.each { nummer ->
%>    $nummer
<% } } %>'''
```

Beispiel 4-6: Skript zum Mischen von Daten (Fortsetzung)

```
// Template-Engine instantiieren
engine = new groovy.text.SimpleTemplateEngine()
// Template aus Textvorlage erzeugen
template = engine.createTemplate(textvorlage)
// Binding erstellen
daten = [liste:kontaktliste, stand:new Date()]
// Mischvorgang auslösen
ergebnis = template.make(daten)
// Ergebnis ausgeben
new File('Telefonliste-Ergebnis.txt').withWriter { writer ->
    writer.write(ergebnis)
}
```

Achten Sie darauf, dass Sie, wenn Sie einen Vorlagentext wie hier als String-Literal definieren, dieses nicht in doppelte Anführungszeichen einschließen ("**...**" oder """**...**"""). In diesem Fall wird der Groovy-Compiler nämlich den Text als `GString` betrachten und versuchen, die Platzhalter wie `$nummer` schon beim Kompilieren zu ersetzen. Dies ist natürlich nicht erwünscht, denn das Mischen von Text und Daten soll dem Template vorbehalten bleiben. Einfache Anführungszeichen um einen String verhindern, dass dieser als `GString` interpretiert wird.

Weitere Skript-Engines in Groovy

Wie erwähnt, liefert die Groovy-Umgebung noch zwei weitere Template-Engines aus. Eine von ihnen, die `XmlTemplateEngine`, ist auf das Verarbeiten und Generieren von XML-Dokumenten spezialisiert und wird in Kapitel 6 näher behandelt.

Schließlich gibt es noch die `GStringTemplateEngine`. Sie kann in genau der gleichen Weise verwendet werden. Sie brauchen im obigen Beispiel nur die Zeile

```
engine = new groovy.text.SimpleTemplateEngine()
```

durch diese Zeile zu ersetzen:

```
engine = new groovy.text.GStringTemplateEngine()
```

Das Ergebnis sieht genau so aus wie vorher. Die Templates beider Engines generieren `Writable`-Objekte, die denselben Ergebnistext über einen Writer ausgeben können. Allerdings unterscheidet sich die Implementierung gravierend. Während die `SimpleTemplateEngine` aus dem Vorlagentext ein Groovy-Skript erzeugt, das den Ergebnistext Zeile für Zeile in den Writer schreibt, baut die `GStringTemplateEngine` ein spezielles `Closure`-Objekt, das das gesamte Template als `GString` in einem Stück ausgibt. Der Unterschied kommt insbesondere dann zum Tragen, wenn das Ergebnis des Mischvorgangs nicht zeilenweise organisiert ist, denn es entfällt die Notwendigkeit, zunächst eine gesamte Zeile

im Speicher zusammenzubauen, bevor sie ausgegeben werden kann. Ein Beispiel dafür ist die Übertragung von Streaming-Daten im Internet, die vom Empfänger ja schon während des laufenden Übertragungsvorgangs dargestellt werden sollen. Zusammengefasst kann man sagen, dass sich `SimpleTemplate` bei umfangreichen Ausgaben anbietet, da die Verarbeitung inkrementell geschieht, und `GStrings` bei kleineren Strings, da der Zwischenschritt der Generierung von Groovy-Code entfällt.

Mit der Closure, dem Builder und der Template-Engine haben Sie drei Konzepte kennengelernt, die in in der Groovy-Programmierung generell eine große Rolle spielen und die in vielen Teilen der Groovy-Bibliothek angewendet und umgesetzt werden. Ihnen werden wir in den folgenden Kapiteln immer wieder begegnen.

KAPITEL 5

Wie Groovy das JDK erweitert

Die zu Groovy gehörenden Bibliotheken umfassen eine Reihe von Klassen, die in sehr enger Beziehung mit der Sprache selbst stehen. Dazu gehören zum einen die zusätzlichen, von der Sprache direkt unterstützten Typen wie Closures und GStrings, aber auch die vielen Erweiterungen zu den standardmäßigen Java-Typen, beispielsweise Strings, reguläre Ausdrücke, Wrapper für primitive Typen, Zahlen beliebiger Größe usw., die durch Groovy viel weitgehender unterstützt werden, als dies bei Java der Fall ist. Diese Erweiterungen werden bisweilen als »Groovy-JDK« oder als »Groovy Development Kit« (GDK) bezeichnet.

Zunächst beschäftigen wir uns detailliert mit den Groovy-eigenen Mechanismen, mit deren Hilfe vorhandene Klassen gewissermaßen von außen mit zusätzlichen Funktionalitäten versehen werden; wir nennen sie in diesem Buch »vordefinierte Methoden«. Dann gehen wir näher auf die verschiedenen Java-Typen ein, für die diese neuen Methoden gedacht sind, und sehen uns an, wie man mit diesen Typen in Groovy-Programmen umgehen kann.

Wir werden in diesem Kapitel nur auf die relativ häufig benutzten vordefinierten Methoden sowie auf jene eingehen, die einer Erläuterung bedürfen. Eine vollständige Auflistung all dieser Methoden finden Sie in Anhang A.

Vordefinierte Methoden

Überall in diesem Buch kommen in den Beispielprogrammen Aufrufe einer Methode `println()` vor, die offensichtlich für die Klassen, in denen sich diese Aufrufe befinden, nicht definiert ist. In Kapitel 3 haben Sie erfahren, dass in Groovy die meisten Operatoren durch spezielle Methoden implementiert werden, beispielsweise der Plusoperator (+) durch die Methode `plus()`. Trotzdem können wir zwei Integer-Zahlen addieren, obwohl

die Klasse `Integer` keine Methode `plus()` besitzt. Wir wollen uns ansehen, wie Groovy dies zustande bringt und wie wir diesen Mechanismus in eigenen Programmen nutzen können.

Das Prinzip

Wenn Groovy versucht, einen Methodenaufruf bei einem gegebenen Objekt durchzuführen, sucht es genau wie Java diejenige Methode, die für die Klasse des Objekts definiert oder von einer Basisklasse geerbt ist und am besten zu den übergebenen Parametern passt (allerdings findet die Zuordnung von Aufruf und aufrufbaren Methoden bei Java bereits beim Kompilieren statt; wenn es eine solche Zuordnung nicht gibt, führt dies zu einem Fehler). Wenn Groovy nun keine passende Methode findet, sieht es anschließend in einer zur Laufzeitumgebung von Groovy gehörenden Klasse `org.codehaus.groovy.runtime.DefaultGroovyMethods` nach – im statischen Kontext ist es `DefaultGroovyStaticMethods` im selben Package –, ob sich darin eine statische Methode gleichen Namens befindet, deren erster Parameter von einem Typ ist, mit dem das Objekt zuweisungskompatibel ist, und deren restliche Parameter mit den Argumenten des Methodenaufrufs übereinstimmen. Und wenn es eine solche findet, ruft es diese anstelle einer für das Objekt definierten Methode auf.[1]

So enthält `DefaultGroovyMethods` beispielsweise die folgende Methode:

```
public static void println(Object self, Object value) {
    System.out.println(InvokerHelper.toString(value));
}
```

Wenn Sie nun in Ihrem Programm irgendwo einen Aufruf wie `println('text')` haben und in der betreffenden Klasse eine solche Methode nicht bereits definiert ist, findet Groovy die obige Methode, denn in welcher Klasse sich der Aufruf auch immer befindet, muss diese doch zuweisungskompatibel zu `Object` sein, und der zweite Parameter der Methode passt zum ersten und einzigen Argument des Aufrufs. Diese `println()`-Methode macht nun nichts weiter, als das übergebene Objekt mithilfe einer Hilfsklasse in einen String umzuwandeln und an die Standardausgabemethode `System.out.println()` weiterzureichen.

Ein weiteres Beispiel ist die Methode `plus()`, die in `DefaultGroovyMethods` zu finden ist und es ermöglicht, dass der Plusoperator auch bei den numerischen Standardtypen funktioniert:

```
public static Number plus(Number left, Number right) {
    return NumberMath.add(left, right);
}
```

1 Die Klasse `DefaultGroovyMethods` enthält inzwischen weit über 300 Methoden, und ein Refactoring wird bereits seit einiger Zeit diskutiert. Eine neue Implementierung wird vermutlich die vordefinierten Methoden auf verschiedene Klassen verteilen, die über eine Namenslogik mit den korrespondierenden Typen verknüpft sind.

Wenn Sie nun in Ihrem Code einen Ausdruck wie `a+b` mit zwei Integer-Zahlen haben, wandelt Groovy diesen in einen Methodenaufruf `a.plus(b)` um; eine entsprechende Methode ist in der Klasse `Integer` nicht zu finden, aber `DefaultGroovyMethods.plus(a,b)` ist möglich, und somit wird ersatzweise diese Methode ausgeführt.

Die vordefinierten Methoden kommen immer nur dann ins Spiel, wenn nicht das aktuelle Objekt selbst, bei dem die Methode aufgerufen werden soll, oder eines seiner Vorfahren eine zum Aufruf passende Methode implementiert. Folgendes Beispielskript soll dies demonstrieren.

```
class EineKlasse {
}
k = new EineKlasse()
k.println('Hallo')
```

Wenn Sie dieses Skript laufen lassen, wird der Text »Hallo« ausgegeben, da die Klasse `EineKlasse` selbst keine Methode `println()` implementiert, es aber eine passende vordefinierte Methode gibt. Bauen wir die Klasse nun so um, dass sie eine Implementierung von `println()` enthält:

```
class EineKlasse {
    def println(text) {
        System.out.println("EineKlasse: $text")
    }
}
k = new EineKlasse()
k.println('Hallo')
```

Unsere eigene `println()`-Implementierung funktioniert ähnlich wie das Original, fügt aber etwas vor dem ausgegebenen Text ein. Starten Sie dieses Skript, und es erscheint tatsächlich der Text »EineKlasse: Hallo« auf der Konsole.

Dieses Verhalten entspricht weitgehend dem Überschreiben von Methoden in abgeleiteten Klassen. So können Sie auch eine der Elternklasse zugeordnete vordefinierte Methode mit dem Präfix `super` adressieren. Der Vorteil dieses Vorgehens ist, dass auch manche Klassen der Java-Standardbibliothek Methoden aufweisen, die mit korrespondierenden vordefinierten Methoden übereinstimmen. In solchen Fällen wird die spezifische, in der Klasse definierte Methode aufgerufen und nicht die von Groovy vordefinierte Variante. Ein Beispiel hierfür ist die Methode `iterator()`, die von allen `Collection`-Klassen implementiert wird, gleichzeitig aber auch für alle Objekte vordefiniert ist. Es ist also, als hätten diese Standardklassen die vordefinierte Methode »überschrieben«.

Vordefinierte Methoden für alle Objekte

Es gibt zahlreiche Methoden, die für verschiedene Standardklassen und -Interfaces vordefiniert sind. Wir wollen uns zunächst denjenigen zuwenden, die für den Typ `Object` verfügbar sind, also überall im Programm verwendet werden können. Manche von ihnen sind für bestimmte Typen überladen.

`Object asType (Class zielTyp)`

Wandelt das aktuelle Objekt in eine Instanz des genannten Typs – sofern dies möglich ist. Die Methode implementiert das Schlüsselwort `as` zur Typanpassung und wird für viele spezifische Typen überschrieben.

`String dump()`

Generiert einen zusammenfassenden String aus Klassennamen, Hashcode und Feldern des Objekts, der besonders für Debugging-Zwecke geeignet ist.

`ProxyMetaClass getMetaClass()`

Ordnet dem aktuellen Objekt eine `ProxyMetaClass`-Instanz zu, falls dies nicht bereits geschehen ist, und übergibt diese als Ergebnis. Auf diesem Weg kann man einer Klasse bequem neue Methoden zuordnen. Näheres dazu in Kapitel 7.

`Map getProperties ()`

Liefert eine Map mit den Namen und Werten aller für das Objekt definierten Properties.

`Object identity (Closure c)`

Ruft die Closure mit dem aktuellen Objekt selbst als Argument auf. Objektreferenzen in der Closure werden ebenfalls über das aktuelle Objekt aufgelöst, da es gleichzeitig als Delegate angemeldet ist. *`objekt`*`.identity { closure }` ist also ungefähr gleichbedeutend mit `closure.delegate=`*`objekt`*`; closure.call (`*`objekt`*`)`. Die Konstruktion ähnelt der `with`-Anweisung in Basic; man kann sie verwenden, wenn man auf viele Member eines Objekts zugreifen will. Ein typischer Anwendungsfall sind Plausibilitätsprüfungen an einem Domain-Objekt:

```
def validatePerson (person) {
    person.identity {
        if (vorname==null || vorname.length()<2) {
           throw new ...
        }
    // usw.
    }
}
```

`String inspect ()`

Generiert einen String, dessen Inhalt die Form hat, mit der in einem Groovy-Programm ein Objekt mit demselben Inhalt wie das aktuelle Objekt gebildet werden könnte. Wenn dies nicht möglich ist, liefert die Methode dasselbe Ergebnis wie `toString()`.

Beispiel: Das Ergebnis von `[1,2,3].inspect()` ist ein String mit dem Inhalt `"[1,2,3]"`.

`Boolean is (Object obj)`

Prüft, ob das aktuelle Objekt identisch mit dem als Argument übergebenen Objekt ist. Diese Methode wird benötigt, da der Gleichheitsoperator (`==`) in Groovy die `equals()`-Methode aufruft und nicht, wie in Java, auf Objektidentität prüft.

`Boolean isCase (Object switchObjekt)`

Prüft, ob das als Argument übergebene Switch-Objekt in einer `switch-case`-Verzweigung ein »Fall« des aktuellen Objekts ist. Wenn diese Methode nicht überladen ist, gibt sie das Ergebnis eines Aufrufs von `equals(switchObject)` zurück. Dass diese Methode für die Klasse `Object` definiert ist, bedeutet, dass jedes beliebige Objekt als `case` verwendet werden kann.

`void print (Object obj)`

Ruft `System.out.print (obj)` auf.

`void print (PrintWriter writer)`

Gibt eine String-Repräsentation des aktuellen Objekts über die `print()`-Methode des angegebenen `PrintWriter` aus.

`void printf (String format, Object[] args)`

Ruft `System.out.printf(format, args)` auf. Funktioniert nur ab Java 5.0.

`void println (Object obj)`

Ruft `System.out.println (obj)` auf.

`void println (PrintWriter writer)`

Gibt eine String-Repräsentation des aktuellen Objekts über die `println()`-Methode des angegebenen `PrintWriter` aus.

`void println ()`

Gibt einen Zeilenvorschub über `System.out.println()` aus.

`void putAt (String name, Object wert)`

Ermöglicht den schreibenden Zugriff auf die Properties eines Objekts über den Property-Namen wie auf den Index einer Map.

`static void sleep (long milliseconds, Closure onInterrupt=null)`

Unterbricht die Ausführung für die angegebene Anzahl von Millisekunden. Unterbrechungen werden während dieser Zeit intern abgefangen; man kann also davon ausgehen, dass die angegebene Zeit ungefähr eingehalten wird. Wenn eine Closure angegeben ist, wird sie im Fall einer Unterbrechung mit der `InterruptedException` als Argument aufgerufen.

`void use (Class[] kategorien, Closure c)`

`void use (List kategorien, Closure c)`

Die beiden Methoden bewirken die Ausführung der übergebenen Closure, dabei werden bei allen innerhalb der Closure (auch indirekt) ausgelösten Methodenaufrufen die in den Kategorienklassen definierten Methoden als vordefinierte Methoden

berücksichtigt (Näheres dazu in Kapitel 7). Die beiden Methoden unterscheiden sich darin, dass bei der einen eine variable Anzahl einzelner und bei der anderen eine Liste von Kategorienklassen angegeben werden kann.

Vordefinierte Methoden zur Iteration über Objektmengen

Es gibt eine Reihe vordefinierter Methoden, die speziell für Container, Files, Streams und andere Objekte vorgesehen sind, die den Zugriff auf ein oder mehrere Elemente organisieren. Diese iterativen Methoden bilden mithilfe eines Aufrufs von `iterator()` einen Iterator über den Inhalt des aktuellen Objekts und führen eine bestimmte Aktion für jedes der vom Iterator gelieferten Elemente aus. Die meisten von ihnen rufen eine als Argument angegebene Closure auf und reagieren in unterschiedlicher Weise auf das Ergebnis des Closure-Aufrufs. Alle diese Methoden stützen sich allein auf das Vorhandensein der `iterator()`-Methode bei dem aktuellen Objekt, sie setzen aber nicht die Implementierung eines bestimmten Interface voraus. Tatsächlich funktionieren sie prinzipiell mit beliebigen Objekten, da bereits der Klasse `Object` eine vordefinierte `iterator()`-Methode zugeordnet ist:

```
Iterator iterator ()
```

Sie bildet einen Iterator über im Objekt enthaltene Elemente. Im einfachsten Fall ist es nur ein Iterator über das aktuelle Element selbst. Die Methode wird nicht nur in den weiter unten aufgeführten vordefinierten iterierenden Methoden verwendet, sondern auch in der Implementierung der Groovy-eigenen `for`-Schleife.

```
groovy> for (element in new Object()) { println element }
java.lang.Object@14807d9
```

Die `for`-Schleife in dem folgenden Skript gibt nacheinander die Werte der drei Properties (100, »alpha« und das aktuelle Datum) aus, da die `iterator()`-Methode der Klasse `K` einen entsprechenden Iterator liefert.

```
class K {
    def a = 100
    def b = 'alpha'
    def c = new Date()
    def iterator() {
        this.properties.values().iterator()
    }
}
for (element in new K()) {
    println element
}
```

Für zahlreiche Typen ist `iterator()` so überschrieben, dass sie eine für den jeweiligen Typ sinnvollen Iterator liefert. Die entsprechenden Methodendefinitionen befinden sich teilweise in der jeweiligen Klasse selbst und teilweise in `DefaultGroovyMethods`. Tabelle 5-1

Tabelle 5-1: Iterator-Elemente verschiedener Typen

Typ des Objekts	Elemente des Iterators
`null`	Keine (leerer Iterator).
Beliebiges Array	Inhaltselemente des Arrays, die, wenn es ein Array aus primitiven Werten ist, in die korrespondierenden Wrapper-Typen umgewandelt werden.
`java.lang.String`	Im String enthaltene Zeichen.
`java.util.Collection`	In der `Collection` enthaltene Objekte.
`java.util.Enumeration`	Von der `Enumeration` gelieferte Objekte.
`java.util.Iterator`	Die Elemente des Iterators selbst.
`java.util.Map`	`Map.Entry`-Objekte.
`java.util.regex.Matcher`	Gefundene Teilsequenzen (Ergebnisse des Aufrufs von `group()`).
`java.io.File`	Zeilen der Datei.
`java.io.InputStream`	Im Stream enthaltene Bytes.
`java.io.Reader`	Eingabezeilen.
`org.w3c.dom.NodeList`	In der `NodeListe` enthaltene `Node`-Objekte.

Alle folgenden vordefinierten Methoden durchlaufen die Elemente des Iterators.

`Boolean any (Closure c)`

Prüft, ob der Aufruf der Closure bei mindestens einem der Elemente das Ergebnis `true` liefert.

Beispiel: `[1,2,3].any {it==2 }` liefert `true`, denn eines der Elemente ist gleich 2.

`List collect (Closure c)`

Sammelt alle Ergebnisse der Closure-Aufrufe in einer Liste.

Beispiel: `[1,2.3].collect {it * 2 }` liefert als Ergebnis die Liste `[2,4,6]`.

`List collect (Collection coll, Closure c)`

Fügt alle Ergebnisse der Closure-Aufrufe der angegebenen Collection hinzu.

Beispiel: `[1,2,3].collect (['a','b']) { it.toString() }` liefert die verlängerte Liste `['a','b','1','2','3']`.

`void each (Closure c)`

Ruft die Closure für jedes Element auf, ohne den Ergebniswert zu berücksichtigen.

Beispiel: `[1,2,3].each { println it }` gibt alle Elemente der Reihe nach aus.

`void eachWithIndex (Closure c)`

Ruft die Closure für jedes Element auf, ohne den Ergebniswert zu berücksichtigen. Dabei wird der Index des Elements, bei 0 beginnend, der Closure als zweites Argument übergeben.

Beispiel: `[1,2,3].eachWithIndex {elem, index -> println "Nr. $index = $elem"}` gibt alle Elemente in einer nummerierten Liste aus.

`Boolean every (Closure c)`

Prüft, ob die Closure für alle Elemente `true` liefert. Folgende Anweisung gibt `true` zurück, weil alle Elemente größer als 0 sind:

```
[1,2,3].every { it > 0 }
```

`Object find (Closure c)`

Liefert das erste Element zurück, für das der Closure-Aufruf `true` ergibt. Wenn dies bei keinem der Elemente der Fall ist, wird `null` zurückgegeben.

Beispiel: `[1,2,3].find { it > 1 }` gibt das Element `2` zurück, da bei ihm als Erstes die Bedingung erfüllt ist.

`List findAll (Closure c)`

Sammelt alle Elemente, für die der Closure-Aufruf `true` ergibt, in einer Liste. Wenn dies bei keinem der Elemente der Fall ist, wird eine leere Liste zurückgegeben.

Beispiel: `[1,2,3].findAll { it > 1 }` gibt die Liste `[2,3]` zurück, da bei diesen beiden Elementen die Bedingung erfüllt ist.

`Integer findIndexOf (Closure c)`

Liefert den Index des ersten Elements zurück, für das der Closure-Aufruf `true` ergibt, beginnend mit 0. Wenn die Closure bei keinem der Elemente `true` liefert, wird die Anzahl der Elemente zurückgegeben.

Beispiel: `[10,20,30].findIndexOf { it > 10 }` gibt den Wert `1` zurück, da bei dem zweiten Element als Erstes die Bedingung erfüllt ist.

`List grep (Object obj)`

Ermittelt einen Iterator über das aktuelle Objekt und liefert alle Elemente daraus, die mit dem als Argument übergebenen Objekt übereinstimmen, als Liste zurück. Ob es eine Übereinstimmung gibt, wird durch einen Aufruf von `obj.isCase(`*`element`*`)` ermittelt.

Beispiel: `[10,20,30].grep(15..25)` liefert die einelementige Liste `[20]`, denn nur der Wert `20` ist im Wertebereich `15..25` enthalten.

Das Objekt darf durchaus auch ein Regulärer Ausdruck sein, dann wird die Nähe zum gleichlautenden Unix-Befehl noch deutlicher. Beispiel:

```
['Januar', 'Februar', 'März', 'April', 'Juni', 'Juli'].grep(~/.*r.*/)
```

Zahlen und Arithmetik

Während sich der normale Umgang mit numerischen Werten und Rechenoperationen nicht wesentlich von Java unterscheidet, gibt es doch einige gravierende Unterschiede, die sich daraus ergeben, dass Groovy grundsätzlich nur mit Objekten rechnet und nicht mit primitiven Typen und dass standardmäßig `BigInteger` und `BigDecimal` anstelle von `Float` und `Double` verwendet werden. Außerdem steht durch Groovys Laufzeitbibliothek eine Reihe zusätzlicher oder vereinfachter Funktionalitäten zur Verfügung.

Numerische Datentypen

Tabelle 5-2 zeigt eine Übersicht der vordefinierten numerischen Datentypen (alle sind vorzeichenbehaftet).

Tabelle 5-2: Numerische Datentypen in Groovy

Klasse	Suffix	Bedeutung	Beispiel-Literal
`java.lang.Byte`		8-Bit-Ganzzahl	`(Byte)100`
`java.lang.Short`		16-Bit-Ganzzahl	`(Short)100`
`java.lang.Integer`		32-Bit-Ganzzahl	`100`
`java.lang.Long`	`L`, `l` (kleines L)	64-Bit-Ganzzahl	`100L` `10000000000`

`java.lang.Float`	`F`, `f`	32-Bit-Fließkommazahl	`100F`
			`100.0f`
			`1e2`

`java.lang.Double`	`D`, `d`	64-Bit-Fließkommazahl	`100D`
			`100.0d`
			`1e2d`

`java.math.BigInteger`	`G`, `g`	Ganzzahl beliebiger Länge	`100G`
			`1e2`
			`10000000000000000000`

`java.math.BigDecimal`	`G`, `g`	Dezimalzahl beliebiger Länge	`100.0`
			`100.0G`
			`1.0000e2`

Der sicher bedeutendste Unterschied zu Java besteht darin, dass Groovy sehr große und gebrochene Zahlen standardmäßig als `BigInteger`- und `BigDecimal`-Objekte speichert und auch die Anwendung der üblichen arithmetischen Operationen mit solchen Werten ermöglicht. Das hat den Vorteil, dass es keine Genauigkeitsprobleme gibt, die bei Fließkommazahlen bisweilen zu überraschenden Ergebnissen führen. So ergibt beispielsweise die Addition der Zahlen 1.7 und 1.9 in Java, mit `double`-Zahlen berechnet, das Ergebnis 3.5999999999999996. In Groovy, mit `BigDecimal` berechnet, erhalten wir den eher erwartungskonformen Wert 3.6.

Auch Groovy-Fließkommazahlen haben allerdings eine Rundung entsprechend der Java-Klasse *BigDecimal*, z.B. `2/3 == 0.6666666667`. Die Genauigkeit kann über `scale` abgefragt werden: `(2/3).scale == 10`

Es ist auch weiterhin möglich, Fließkommazahlen zu benutzen, nur müssen bei den Literalen die entsprechenden Suffixe (`F`, `f`, `D`, `d`) angebracht oder die Variablen entsprechend typisiert werden. Beide folgenden Variablen werden mit einem `Double`-Wert initialisiert.

```
def x = 100d
Double y = 100
```

Berechnungen mit `BigInteger` und `BigDecimal` gelten als relativ langsam, daher kann es sinnvoll sein, in rechenintensiven Programmen, bei denen der Genauigkeitsverlust nicht wesentlich ist oder durch geeignete Maßnahmen abgefangen wird, lieber bei Fließkommazahlen zu bleiben.

Typanpassungen bei arithmetischen Operationen

Wenn mehrere numerische Werte in Rechenoperationen miteinander verknüpft werden, stellt sich die Frage, mit welcher Genauigkeit die jeweilige Operation ausgeführt wird und von welchem Typ das Ergebnis ist. Java hat die einfache Regel, dass bei Operationen zwischen verschiedenen Typen die Berechnung in dem genaueren der beiden Typen ausgeführt wird und das Ergebnis auch diesen Typ hat – mit der einzigen Ausnahme, dass ganzzahlige Operationen mindestens in `Integer`-Genauigkeit, also mit mindestens 32 Bit, ausgeführt wird. Das führt zu kurios erscheinenden Situationen; beispielsweise hat das Ergebnis der Berechnung `1/3` den Wert `0`, da die Division zwischen zwei Integer-Zahlen eine ganzzahlige Division ist.

Groovy verhält sich auch beim Dividieren erwartungskonformer, da es das Ergebnis einer Division zwischen zwei ganzen Zahlen in einen `BigDecimal`-Wert speichert. Wenn Sie eine ganzzahlige Division wie in Java benötigen, müssen Sie eine Typumwandlung vornehmen (z.B. `1/3 as Integer`). Einige andere Probleme, die es bei arithmetischen Operationen in Java gibt, bleiben uns aber auch in Groovy erhalten.

Tabelle 5-3: Genauigkeit arithmetischer Operationen

Operation	Typ der Operanden	Typ des Ergebnisses
/ (Division)	Mindestens einer der Operanden ist `Float` oder `Double`.	`Double`
	In allen übrigen Fällen.	`BigDecimal`

Tabelle 5-3: Genauigkeit arithmetischer Operationen (Fortsetzung)

Operation	Typ der Operanden	Typ des Ergebnisses
+ (Addition) - (Subtraktion)	Mindestens einer der Operanden ist `Float` oder `Double`.	`Double`
* (Multiplikation)	Mindestens einer der Operanden ist `BigDecimal`.	`BigDecimal`
	Mindestens einer der Operanden ist `BigInteger`.	`BigInteger`
	Mindestens einer der Operanden ist `Long`.	`Long`
	In allen übrigen Fällen.	`Integer`

** (Potenz)	Beliebig (der Typ des Ergebnisses wird in Abhängigkeit von der Größe und Genauigkeit des Ergebnisses ermittelt).	`Integer`
		`Long`
		`Double`

Die Tatsache, dass der Ergebnistyp bei Additionen, Subtraktionen und Multiplikationen allein aufgrund der beteiligten Operandentypen bestimmt wird, bringt es mit sich, dass man die gleiche Vorsicht vor Überlaufen walten lassen muss wie bei Java auch. Auf die Möglichkeit, den Ergebnistyp anhand der Größe und der Genauigkeit des Ergebnisses zu wählen, wie es etwa bei Ruby der Fall ist, hat man aus Performancegründen verzichtet.

Das ist insbesondere dann tückisch, wenn es in Zwischenwerten einen unerwarteten Überlauf geben kann:

```
groovy> println 123456789/100*100
groovy> println 123456789*100/100
123456789.00
-5392229.88
```

Beide Berechnungen sollten den Wert 123456789 liefern. Leider kommt es aber bei der zweiten Formel zu einem Überlauf nach der Multiplikation, der zu einem völlig unsinnigen Ergebnis führt. Würde Groovy das Zwischenergebnis, das in einem Integer nicht mehr unterzubringen ist, in einer `Long`- oder einer `BigInteger`-Variablen speichern, wäre das Ergebnis korrekt. Immerhin ist das Verhalten aber besser als das von Java, wo bei der ersten Berechnung aufgrund der Ganzzahldivision zwei Stellen verloren gingen und das Ergebnis 123456700 wäre.

Hier müssen Sie also weiterhin selbst darauf achten, ob die von Ihnen gewählten Datentypen für die zu erwartenden Werte auch ausreichend, und im Zweifelsfall gleich mit `BigInteger` oder `BigDecimal` rechnen, was Groovy Ihnen ja sehr leicht macht.

Vordefinierte numerische Methoden

Das GDK enthält eine Reihe von vordefinierten Methoden für den Typ `Number` bzw. Spezialisierungen für einzelne numerische Typen. Viele von ihnen dienen dazu, die numerischen Operatoren zu implementieren. Von den übrigen vordefinierten Methoden wollen wir einige wichtigere hier kurz erläutern.

`Number abs()`

Bildet den Absolutwert und liefert das Ergebnis als neue Instanz. Wird von den Klassen `BigInteger` und `BigDecimal` selbst implementiert.

`void downto (Number n, Closure c)`

Ruft die Closure einmal mit dem aktuellen Objekt auf und danach mit jeweils um 1 verminderten Werten, bis die als Argument angegebene Zahl erreicht ist. Der letzte übergebene Wert ist also größer oder gleich `n`. Der Wert von `n` muss kleiner oder gleich dem aktuellen Objekt sein.

Beispiel: `1.1.downto(-3) { println it }` gibt nacheinander die Werte `1.1`, `0.1`, `-0.9`, `-1.9` und `-2.9` aus.

`Number intdiv (Number n)`

Führt eine ganzzahlige Division des aktuellen Objekts durch das angegebene Argument durch und liefert das Ergebnis als neue Instanz. Beide Werte müssen von einem ganzzahligen Typ (also weder `Float` noch `Double` noch `BigDecimal`) sein.

Beispiel: `4.intdiv(3)` hat das Ergebnis `1`.

`Number multiply (Number n)`

Implementiert den Operator *. Multipliziert das aktuelle Objekt mit dem Argument und liefert das Ergebnis als neues Objekt zurück.

`void step (Number bis, Number schrittweite, Closure c)`

Durchläuft in einer Schleife die Werte vom aktuellen Objekt bis zu dem des ersten Arguments (exklusive), mit der Schrittweite des zweiten Arguments. Bei jedem Schritt wird die Closure aufgerufen.

Beispiel: `10.step(-10,-5) {println it}` gibt die Werte `10`, `5`, `0` und `-5` aus.

`void times (Closure c)`

Durchläuft in einer Schleife die Werte von 0 bis zum aktuellen Objekt mit der Schrittweite 1 und ruft bei jedem Schritt die Closure mit dem jeweiligen Wert auf.

Beispiel: `5.times { println it }` gibt die Werte `0`, `1`, `2`, `3` und `4` aus.

`BigDecimal toBigDecimal ()`

`BigInteger toBigInteger ()`

`Double toDouble ()`

`Float toFloat ()`

`Integer toInteger ()`

`Long toLong ()`

Diese Methoden wandeln das aktuelle Objekt in den durch den Methodennamen bezeichneten Typ um. Überzählige Stellen hinter dem Komma werden ohne Rundung abgeschnitten. Auf den möglichen arithmetischen Überlauf beim Umwandeln in kleinere Typen müssen Sie selbst achten.

`void upto (Number n, Closure c)`

Ruft die Closure erst mit dem aktuellen Objekt und danach mit jeweils um 1 erhöhten Werten auf, bis die als Argument angegebene Zahl erreicht ist. Der letzte übergebene Wert ist also kleiner oder gleich `n`. Der Wert von `n` muss größer oder gleich dem aktuellen Objekt sein.

Beispiel: `0.1.upto(3) { println it }` gibt nacheinander die Werte `0.1`, `1.1` und `2.1` aus.

`Number xor (Number n)`

Führt ein bitweises exklusives Oder zwischen dem aktuellen Objekt und dem Argument durch und liefert das Ergebnis als neue Instanz.

Strings und Zeichen

Neben dem Rechnen mit Zahlen machen die String-Operationen normalerweise einen Großteil der alltäglichen Programmierung aus. Groovy bietet auch hier einige Erleichterungen und zusätzliche Möglichkeiten, die das Arbeiten mit Texten und Zeichen einfacher, übersichtlicher und sicherer als bei Java machen. Neben zusätzlichen Formen für String-Literale sind dies vor allem die String-Interpolation mit *GStrings* sowie die direkte Einbindung von regulären Ausdrücken in die Groovy-Sprache.

String-Literale

Ein normales String-Literal wird im Wesentlichen genau so gebildet wie in Java. Die einzige Abweichung besteht darin, dass als Begrenzungszeichen einfache Anführungszeichen dienen.

```
groovy> println '\'Alea iacta est!\' sagt der Lateiner.'
'Alea iacta est!' sagt der Lateiner.
```

Eingebetteten Anführungszeichen muss ein Backslash vorangestellt werden, die bekannten Escape-Sequenzen für Sonderzeichen und Unicode-Zeichen stehen ebenfalls zur Verfügung. Sie können als Begrenzungszeichen auch doppelte Anführungszeichen verwenden, allerdings kennzeichnen Dollarzeichen dann Interpolationen (siehe weiter unten) und müssen ebenfalls mit einem Backslash versehen werden, wenn dies nicht erwünscht ist.

```
groovy> println "Dem \$-Zeichen muss ein \\-Zeichen vorangestellt werden."
Dem $-Zeichen muss ein \-Zeichen vorangestellt werden.
```

Schließlich kann auch noch der Slash (Schrägstrich) als Begrenzungszeichen dienen. In einem solchen String-Literal gilt der Backslash nicht als Escape-Zeichen, daher eignet sich diese Form insbesondere für reguläre Ausdrücke, aber auch beispielsweise für Windows-Pfadangaben:

```
groovy: p = java.util.regex.Pattern.compile(/\s\w+\s/)
groovy: println p
groovy: println (/\C:\Dokumente und Einstellungen\All Users\Dokumente/)
\s\w+\s
\C:\Dokumente und Einstellungen\All Users\Dokumente
```

Zwei Dinge sind bei in Schrägstrichen eingeschlossenen String-Literalen besonders zu beachten: Das Dollarzeichen dient auch hier, sofern es nicht am Ende des Strings steht, zur Kennzeichnung von String-Interpolationen, und die Sequenz `\u` leitet weiterhin eine Unicode-Sequenz ein. Beide können nicht ohne Weiteres mit Backslash-Zeichen außer Kraft gesetzt werden; in der Praxis ist dies aber kaum von Bedeutung.

Es gibt keine speziellen Literale für Einzelzeichen, sie können aber bedenkenlos durch explizite oder implizite Typanpassung gebildet werden:

```
Character zeichenA = 'A'
Character zeichenB = "B"
def zeichenC = (char)'C'
def zeichenD = 'D' as Character
def zeichenE = 'E'.toCharacter()
```

String-Literale, die in einfache oder doppelte Anführungszeichen eingeschlossen sind, können auch über mehrere Zeilen reichen. Dazu müssen die Begrenzungszeichen nur verdreifacht werden. Ansonsten unterscheiden sie sich nicht von ihren einzeiligen Varianten.

```
a = '''Dies ist ein String,
der über mehrere
Zeilen reicht.'''
b = """Dieser Text
reicht ebenfalls
über mehrere Zeilen"""
```

GDK-Erweiterungen für String und StringBuffer

Die Groovy-Standardbibliothek enthält eine Fülle von Erweiterungen für die Klasse `String`. Beachten Sie, dass alle Operatoren und Methoden für Strings nie den String selbst modifizieren, sondern gegebenenfalls eine neue Instanz liefern. Methoden für `StringBuffer`, der im Gegensatz zu `String` veränderlich ist, können dagegen auch Veränderungen am aktuellen Objekt vornehmen. Die ab Java 5.0 eingeführte Klasse `StringBuilder` ist in Groovy 1.0 noch nicht berücksichtigt.

Operatoren und Methoden zur Textmanipulation

Während Java nur einen Operator für Strings kennt, nämlich das +-Zeichen zum Verketten, gibt es in Groovy einige weitere Operatoren, die durch korrespondierende vordefinierte Methoden implementiert sind. Tabelle 5-4 zeigt die verfügbaren Operatoren mit Ausnahme der Indexoperatoren, auf die wir weiter unten zu sprechen kommen.

Tabelle 5-4: String-Operatoren in Groovy

Operator	Operandentyp	Funktionalität
`+`	`Object`	Verkettet den links stehenden String mit dem in einen String umgewandelten rechts stehenden Objekt. Beispiel: `'ABC'+123` ergibt `'ABC123'`.
`<<`	`Object`	Verkettet den links stehenden String mit dem in einen String umgewandelten rechts stehenden Objekt; das Ergebnis ist ein `StringBuffer`, mit dem auf gleiche Weise weitere Verkettungen vorgenommen werden können. Beispiel: `'ABC'<<123` ergibt einen `StringBuffer` mit dem Inhalt »ABC123«.
`-`	`Object`	Entfernt das erste Vorkommen des in einen String umgewandelten rechts stehenden Objekts aus dem links stehenden String. Beispiel: `'ABCDABCD'-'BC'` ergibt `'ADABCD'`.
`*`	`Number`	Vervielfacht den links stehenden String um die rechts stehende Zahl. Beispiel: `'AB' * 3` ergibt `'ABABAB'`.
`++`		Erhöht das letzte Zeichen im String um 1. Beispiel: `'AB'++` oder `++'AB'` ergibt `'AC'`.
`--`		Vermindert das letzte Zeichen im String um 1. Beispiel: `'AB'--` oder `--'AB'` ergibt `'AA'`.

Daneben gibt es noch eine Reihe zusätzlicher Methoden, die das Manipulieren von Zeichenketten etwas vereinfachen.

Die Methoden `padLeft()`, `padRight()` und `center()` verlängern den aktuellen String auf die angegebene Länge, wobei die Füllzeichen im ersten Fall links, im zweiten Fall rechts und im dritten Fall beidseitig hinzugefügt werden. Wenn der String bereits so lang wie gewünscht oder länger ist, wird er unverändert zurückgegeben.

```
groovy> '|' + 'ABC'.padLeft(5) + '|'
===> |  ABC|
groovy> '|' + 'ABC'.padRight(7,'-.') + '|'
===> |ABC-.-.|
groovy> '|' + 'ABC'.center(7) + '|'
===> |  ABC  |
```

Die Methode `reverse()` kehrt die Reihenfolge der Zeichen im aktuellen String um.

```
groovy> 'Nur du Gudrun'.reverse()
===> nurduG ud ruN
```

Die Methode `replaceAll()` durchsucht den aktuellen String anhand eines regulären Ausdrucks und ruft für jedes Vorkommen eine Closure auf. Der Closure werden als Aufrufargumente die Werte der gefundenen Match-Gruppen übergeben. Das folgende Beispiel setzt vor alle Großbuchstaben einen Unterstrich und ersetzt alle Kleinbuchstaben durch Großbuchstaben.

```
groovy> 'dateOfMonth'.replaceAll(/([A-Z])?([a-z]*)/) {
     a0,a1,a2 -> println a0; return (a1==null ? '' : '_'+a1)+a2.toUpperCase() }
===> DATE_OF_MONTH
```

String-Inhalt prüfen

Die Methode `count()` prüft, wie oft ein Teilstring in dem aktuellen String enthalten ist. Überlappungen werden dabei mitgezählt.

```
groovy> 'ABCABC'.count 'BC'
===> 2
groovy> 'AAAA'.count 'AA'
===> 3
```

Die Methode `eachMatch()` nimmt einen String als regulären Ausdruck an und ruft für jede Übereinstimmung im aktuellen String eine Closure auf. Die Closure erhält als Argument ein String-Array mit allen Match-Gruppen. Das folgende Beispiel sucht im aktuellen String alle Zeichenfolgen, die aus einem Groß- und einem oder mehreren Kleinbuchstaben bestehen. Das ausgegebene Array besteht jeweils aus dem gesamten Match und den beiden darin enthaltenen Match-Gruppen.

```
groovy> 'Alles leere Worte!'.eachMatch (/([A-Z])([a-z]+)/) { println it }
{"Alles", "A", "lles"}
{"Worte", "W", "orte"}
```

String und StringBuffer als Liste von Zeichen

Eine weitere Gruppe von vordefinierten Methoden ermöglicht es, einen `String` oder `StringBuffer` wie eine Liste oder ein Array aus einzelnen Zeichen zu behandeln. Über den Indexoperator können Sie lesend auf einzelne Zeichen oder mehrere Zeichen zuzugreifen.

```
groovy> text = 'YOGAVORGIRLS'
groovy> println text [0]                        // Zugriff über Index
groovy> println text [-1]                       // Negativer Index
groovy> println text [0..3]                     // Indexbereich
groovy> println text [2,6,1,5,4,0,7,9,3,8,10,11] // Indexliste
Y
S
YOGA
AGOY
GROOVYGRAILS
```

Das obige Beispiel funktioniert in gleicher Weise mit Objekten vom Typ `String`, `StringBuffer`, `Character`-Array und `Character`-Liste, allerdings sind die Ergebnistypen bei Listenobjekten und Arrays für Indexbereiche und Indexlisten wiederum Listen und keine

Strings. Sie können dies ausprobieren, indem Sie die erste Zeile im obigen Beispiel durch folgende Varianten ersetzen:

```
groovy> text = new StringBuffer('ODRACONIANDEVIL')
groovy> text = ['Y','O','G','A','V','O','R','G','I','R','L','S']
groovy> text = ['Y','O','G','A','V','O','R','G','I','R','L','S']
              as Character[]
```

Außerdem sind die oben aufgeführten vordefinierten Methoden `contains()`, `count()`, und `reverse()` sowie der <<-Operator in gleicher Weise für Strings wie für Listen definiert, so dass auch in dieser Hinsicht eine ähnliche Behandlung möglich ist.

```
groovy> text = 'NERHEGEB'
groovy> println text.reverse()
```

Interpolieren mit GStrings

Wenn Sie ein String-Literal mit doppelten Anführungszeichen einkleiden, haben Sie die Möglichkeit, einzelne Werte oder ganze Groovy-Ausdrücke einzufügen. Eine solche als String-Interpolation bezeichnete Operation wird immer durch ein Dollarzeichen ($) eingeleitet. Ein einzelner Wert kann aus einem oder mehreren mit Punkten verbundenen Namen bestehen.

```
groovy> zeit = new Date()
groovy> println "Aktuelle Zeit: $zeit"
groovy> println "Aktuelles Datum: $zeit.day.$zeit.month.$zeit.year"
Aktuelle Zeit: Sat Jan 06 22:52:56 CET 2007
Aktuelles Datum: 6.0.107
```

Komplexere Ausdrücke oder Variablen im String, denen direkt ein Zeichen oder eine Zahl folgen soll, müssen zusätzlich in geschweifte Klammern ({}) eingeschlossen werden. Im zweiten Fall kann es sich durchaus um ein größeres Programmstück handeln; bei mehrzeiligen Strings (die bei Groovy mit drei Anführungszeichen abgegrenzt sein müssen) können sie auch über mehrere Zeilen gehen.

```
groovy> zeit = new Date()
groovy> println "Aktuelle Zeit: ${zeit.toString()}"
groovy> println "Aktuelles Datum: ${zeit.day}.${1+zeit.month}.${1900+zeit.year}"
Aktuelle Zeit: Sat Jan 06 22:55:27 CET 2007
Aktuelles Datum: 6.1.2007
```

Wenn Sie in einem solchen String ein Dollarzeichen ausgeben möchten, müssen Sie es mit einem vorangestellten Backslash versehen. Das gilt auch für Strings, die in Schrägstriche eingeschlossen sind. In Strings mit einfachen Anführungszeichen sind Interpolationen nicht möglich.

```
groovy> betrag = 500
groovy> println "Zahlen Sie \$ $betrag"
groovy> println (/Zahlen Sie \$ $betrag/)
groovy> println 'Zahlen Sie \$ $betrag'
Zahlen Sie $ 500
```

```
Zahlen Sie \$ 500
Zahlen Sie $ $betrag
```

Diese Möglichkeit, Text und Programmstücke beliebig zu mischen, erweist sich an vielen Stellen als außerordentlich nützlich. Allerdings gibt es auch hier einige Besonderheiten zu beachten – und es zeigen sich einige zusätzliche Möglichkeiten, die man dem auf den ersten Blick sehr simpel aussehenden Konstrukt zunächst nicht ansieht.

Die Klasse GString

Sobald der Groovy-Compiler einen String mit interpoliertem Ausdruck vorfindet, erzeugt er nicht eine String-Instanz, sondern ein Objekt einer von `groovy.lang.GString` abgeleiteten Klasse; im Grunde haben wir es hier also nicht mit einem Literal, sondern mit einer Art Konstruktor zu tun.

Die interpolierten Ausdrücke werden zwar zur Zeit der Erzeugung des Objekts ausgewertet, aber nicht mit den umgebenden konstanten Teilen des Strings verknüpft. Vielmehr werden diese Ergebnisse der ausgewerteten Ausdrücke und diese konstanten String-Teile innerhalb des `GString`-Objekts aufbewahrt und können auch einzeln mithilfe der Listen-Properties `strings` und `values` ausgelesen werden.

```
groovy> x = 123; y = -200
groovy> gstring = "Die Summe von $x und $y ist ${x+y}"
groovy> println gstring
groovy> println "Strings: " + gstring.strings
groovy> println "Werte: " + gstring.values
Die Summe von 123 und -200 ist -77
Strings: {"Die Summe von ", " und ", " ist ", ""}
Werte: {123, -200, -77}
```

Wie Sie sehen, enthält `strings` die vier Teilstrings, von denen die Ausdrücke umgeben sind, wobei der letzte leer ist, und `values` die drei ausgewerteten Integer-Ausdrücke.

Diese Möglichkeit, die Bestandteile eines `GString` nachträglich auszuwerten, bietet einige interessante Möglichkeiten. Beispielsweise ermöglicht die Groovy-Standardbibliothek mithilfe von GStrings äußerst komfortable SQL-Zugriffe auf Datenbanken (siehe Kapitel 6).

Beachten Sie aber, dass `GStrings` aufgrund dieser eingebetteten Werte anders als normale Java-Strings nicht unveränderbar sind. Das wird sofort klar, wenn man bedenkt, dass die Ergebnisse der eingefügten Ausdrücke unter Umständen noch nachträglich modifiziert werden können.

```
groovy> wort = new StringBuffer('SINN')
groovy> satz = "Das Wort lautet: $wort"
groovy> println satz
groovy> wort.insert(0,'UN')
groovy> println satz
Das Wort lautet: SINN
Das Wort lautet: UNSINN
```

Hier ist die Variable `wort` ein `StringBuffer`, also ein veränderbares Objekt. Indem wir es, nachdem es in den `GString` `satz` eingebunden worden ist, noch einmal verändern, beeinflussen wir auch den Inhalt des `GString`. In vielen Fällen sind `GString`s Objekte mit kurzer Lebenserwartung. Wenn Sie jedoch durch String-Interpolation einen String erzeugen wollen, dem noch ein längeres Leben bevorsteht und der nicht veränderlich sein soll, sollten Sie ihn sicherheitshalber gleich mit `toString()` oder durch Typanpassung in einen ganz normalen Java-String umwandeln. Auch wenn Sie das Ergebnis als Schlüssel in einer Map benutzen möchten, ist dies zu empfehlen. Eine nachträgliche Veränderung der Zutaten kann dem String dann nichts mehr anhaben.

```
groovy> wort = new StringBuffer('SINN')
groovy> satz = "Das Wort lautet: $wort" as String // satz ist String-Objekt
groovy> println satz
groovy> wort.insert(0,'UN')
groovy> println satz
Das Wort lautet: SINN
Das Wort lautet: SINN
```

Die Umwandlung in ein String-Objekt kann auch aus Effizienzgründen vorteilhaft sein. Zwar können fast alle String-Methoden und -Operationen in gleicher Weise auch auf `GString`-Instanzen angewendet werden, da der `GString` alle Methodenaufrufe, die nicht von ihm selbst ausgeführt werden können, an einen ad hoc mit `toString()` gebildeten String weiterleitet. Sie können sich aber leicht vorstellen, dass dieses Neuerzeugen von Strings nicht besonders effizient ist, wenn es häufiger vorkommt.

Reguläre Ausdrücke

Groovy bietet wie viele Skriptsprachen eine direkte Unterstützung für die Arbeit mit regulären Ausdrücken durch spezielle Operatoren. Auch die weiter oben schon eingeführte Möglichkeit, String-Literale zu bilden, in denen der Backslash nicht als Escape-Zeichen ausgewertet wird, macht den Umgang mit regulären Ausdrücken wesentlich angenehmer.

Der unäre Operator ~ wandelt einen String in ein `Pattern`-Objekt um. Jede der folgenden drei Anweisungen erzeugt das gleiche Pattern, das sich zum Prüfen des Formats einer Telefonnummer mit optionaler Vorwahl und optionaler Durchwahl eignet.

```
groovy> p = java.util.regex.Pattern.compile('(\\(0\\d+\\))?(\\d+\\s)*\\d+(-\\d+)?')
groovy> println p
groovy> p = ~'(\\(0\\d+\\))?(\\d+\\s)*\\d+(-\\d+)?'
groovy> println p
groovy> p = ~/(\(0\d+\))?(\d+\s)*\d+(-\d+)?/
groovy> println p
(\(0\d+\))?(\d+\s)*\d+(-\d+)?
(\(0\d+\))?(\d+\s)*\d+(-\d+)?
(\(0\d+\))?(\d+\s)*\d+(-\d+)?
```

Während dieser String-Operator eher praktisch ist, wenn Sie mit Java-Klassen arbeiten, deren Methoden ein `Pattern`-Objekt erwarten, arbeiten Sie Groovy-intern besser mit zwei speziellen Operatoren, die Ihnen das Arbeiten mit regulären Ausdrücken noch weiter erleichtern.

Der Match-Operator `==~` führt einen Mustervergleich zwischen zwei Strings durch, wobei der zweite String einen regulären Ausdruck enthält.

```
groovy> println '(0123)45 67-' ==~ /(\(0\d+\))?(\d+\s)*\d+(-\d+)?/
groovy> println '(0123)45 67-1' ==~ /(\(0\d+\))?(\d+\s)*\d+(-\d+)?/
false
true
```

Dies kann man mit der normalen `matches()`-Methode der String-Klasse fast genau so bequem haben. Ein wirklicher Fortschritt gegenüber Java ergibt sich aus den Möglichkeiten, die Groovy für das Arbeiten mit Match-Ergebnissen bietet.

Der spezielle Find-Operator `=~` liefert ein `java.util.regex.Matcher`-Objekt, mit dem einzelne mit einem regulären Ausdruck übereinstimmende Teilstrings lokalisiert werden können. Für die weitere Verarbeitung stehen verschiedene vordefinierter Methoden und Operatoren zur Verfügung. Im folgenden Beispiel suchen wir nach Teilstrings, die aus einem oder mehreren Buchstaben und optional nachfolgenden Ziffern bestehen.

```
groovy> matcher = 'A23 b35 ZZ X14 33' =~ /([a-zA-Z]+)+(\d*)/
```

Nun können wir beispielsweise die verschiedenen Fundstellen mit einer `for`-Schleife durchlaufen:

```
groovy> for (match in matcher) { print "[$match] " }
[A23] [b35] [ZZ] [X14]
```

Auf die einzelnen Fundstellen kann man auch wie auf die Elemente eines Arrays zugreifen, das Ergebnis ist hier aber jeweils ein Array mit den verschiedenen Match-Gruppen; hier sind dies der gesamte gefundene Teilstring und jeweils die darin enthaltenen Buchstaben und gegebenenfalls Ziffern.

```
groovy> println matcher[1]
groovy> println matcher[2]
["b35", "b", "35"]
["ZZ", "ZZ", ""]
```

Schließlich können wir die Suchergebnisse auch mit den verschiedenen Closure-Methoden durchlaufen. Beispielsweise werden einer Closure mit `each()` die einzelnen Match-Gruppen direkt als Argumente übergeben.

```
groovy> matcher.each { x0, x1, x2 -> println "Buchstaben: $x1 / Ziffern: $x2" }
Buchstaben: A / Ziffern: 23
Buchstaben: b / Ziffern: 35
Buchstaben: ZZ / Ziffern:
Buchstaben: X / Ziffern: 14
```

Containertypen

Fast jedes Java-Programm verwendet sie extensiv, und trotzdem werden sie von der Sprache Java in keiner Weise unterstützt: Listen und Maps. Dementsprechend umständlich ist ihre Verwendung, und Programme, die mit diesen Datentypen arbeiten, werden schnell unübersichtlich. Groovy enthält einige syntaktische Erweiterungen, die es Ihnen ermöglichen, mit solchen Containern in einer einfacheren und natürlicheren Weise umzugehen. Dabei wird insbesondere das Anlegen und initiale Füllen von Listen und Maps erleichtert, und man kann in einer Array-ähnlichen Weise auf die Elemente zugreifen. Außerdem stellt Groovy einen eigenen Listentyp namens `Range` zur Verfügung, mit dem sich sehr einfach Wertebereiche durch die Angabe der Unter- und Obergrenze abbilden lassen. Groovy kennt keine typisierten Container, wie es sie in Java seit der Version 1.5 gibt, allerdings fällt dies nicht so stark ins Gewicht, da man in Groovy ohnehin eher mit untypisierten Werten arbeitet und Typecasts in der Regel nicht notwendig sind.

Listen in Groovy

Listen sind in Groovy alle Typen, die das Interface `java.util.List` implementieren. Ein Teil der für Listen verfügbaren Methoden und Operationen kann allerdings auch auf Objekte angewendet werden, die nicht `List`, sondern nur das allgemeinere Interface `java.util.Collection` (für abstrakte, ungeordnete Behältertypen) implementieren.

Listen anlegen

Sie können eine neue Liste anlegen, indem Sie einfach die darin enthaltenen Elemente in eckige Klammern einschließen und durch Kommata getrennt aufführen.

```
groovy> liste = [1, 7.5, 'zwei Worte']
groovy> println liste.dump()
groovy> liste.each { println it }
<java.util.ArrayList@40af55f6 elementData={1, 7.5, "zwei Worte"} size=3 modCount=3>
1
7.5
zwei Worte
```

Die Variable `liste` verweist also auf eine neue Instanz vom Typ `java.util.ArrayList`, die durch diese Art von Konstruktor standardmäßig erzeugt wird. Durch implizite oder explizite Typanpassung können Sie daraus auch ein Array machen (wobei in diesem Fall die Liste natürlich »typenrein« sein oder zumindest aus anpassbaren Typen bestehen muss).

```
groovy> def array1 = ['eins','zwei','drei'] as String[]
groovy> println array1.class.canonicalName
java.lang.String[]
groovy> int[] array2 = [1,2,3]
groovy> println array2.class.canonicalName
int[]
```

Wenn Sie einen anderen Listentyp – oder sogar einen ganz anderen Collection-Typ – mit Werten initialisieren wollen, rufen Sie einfach dessen Konstruktor auf und geben ihm eine Liste als Argument mit.

```
groovy> def set = new HashSet([1, 7.5, 'zwei Worte'])
groovy> println set.class.canonicalName
java.util.HashSet
```

Eine leere Liste definieren Sie einfach durch eine öffnende und eine schließende eckige Klammer ohne Inhalt.

```
groovy> def leereListe = []
groovy> println leereListe.size()
0
```

Die Elemente einer Liste können natürlich auch wiederum Listen sein. Auf diese Weise können Sie auch verschachtelte Listenstrukturen bauen.

```
groovy> verschachtelteListe = [['alpha','beta'],['gamma','delta']]
groovy> println verschachtelteListe.size()
2
```

Operatoren für Listen

Verschiedene Operatoren sind für den Typ `List` (teilweise auch bereits für `Collection`) überladen, sodass sie eine besondere Bedeutung haben.

Mit dem Indexoperator können Sie auf einzelne Elemente oder Gruppen von Elementen einer Liste zugreifen, als wäre sie ein Array. Allerdings ist auch hier einiges mehr möglich als bei einem Array in Java.

```
groovy> liste = ['alpha','beta','gamma','delta']
groovy> println liste[1]
beta
groovy> println liste[-2]    // Negativer Index zählt von hinten
gamma
groovy> println liste[0,2,3] // Bildet neue Liste mit ausgewählten Elementen
["alpha", "gamma", "delta"]
groovy> println liste[0..2]  // Zugriff auf Bereich von Elementen
["alpha", "beta", "gamma"]
groovy> println liste[2..0]  // Abwärts zählender Bereich
["gamma", "beta", "alpha"]
groovy> println liste[1..-1] // Mit negativer Bereichsgrenze
["beta", "gamma", "delta"]
```

Das Gleiche ist natürlich auch schreibend möglich. Wenn Sie in ein Element schreiben, dessen Index noch nicht existiert, wird die Liste automatisch verlängert und mit `null`-Elementen aufgefüllt.

```
groovy> liste = ['alpha','beta','gamma','delta']
groovy> liste[0] = 'ALPHA'
groovy> println liste
["ALPHA", "beta", "gamma", "delta"]
```

```
groovy> liste[1,3] = 'BETA','GAMMA','DELTA'    // Einen Teilbereich ersetzen
groovy> println liste
["ALPHA", "BETA", "GAMMA", "DELTA"]
groovy> liste[6] = 'ETA'                       // Liste wird bei Bedarf verlängert
groovy> println liste
["ALPHA", "BETA", "GAMMA", "DELTA", null, null, "ETA"]
```

Die folgenden »arithmetischen« Operatoren liefern jeweils eine neue `ArrayList`-Instanz.

```
groovy> liste = ['alpha','beta','gamma','delta']
groovy> println liste + 'epsilon'           // Listenelement hinzufügen
["alpha", "beta", "gamma", "delta", "epsilon"]
groovy> println liste + ['epsilon','zeta']  // Liste hinzufügen
["alpha", "beta", "gamma", "delta", "epsilon", "zeta"]
groovy> println liste - 'beta'              // Listenelement entfernen
["alpha", "gamma", "delta"]
groovy> println liste - ['alpha','delta']   // Elemente einer Liste entfernen
["beta", "gamma"]
groovy> println liste * 2                   // Liste vervielfachen
["alpha", "beta", "gamma", "delta", "alpha", "beta", "gamma", "delta"]
```

Der Linksverschiebungsoperator (`<<`) hat fast die gleiche Wirkung wie der Plusoperator (`+`), jedoch wird das Argument dem aktuellen Objekt hinzugefügt und keine neue Liste erzeugt.

```
groovy> liste = ['alpha','beta','gamma','delta']
groovy> liste << 'epsilon'                   // Listenelement hinzufügen
groovy> println liste
["alpha", "beta", "gamma", "delta", "epsilon"]
```

Ein weiterer interessanter Unterschied zwischen + und << offenbart sich, wenn eine Liste hinzugefügt wird: Der +-Operator konkateniert sie, << fügt die Liste insgesamt als ein Element ein:

```
['a', 'b' ] + [1, 2] == ['a', 'b' , 1, 2]
['a', 'b' ] << [1, 2] == ['a', 'b' , [1, 2]]
```

Vordefinierte Methoden

Des Weiteren enthält die Groovy-Laufzeitbibliothek eine Reihe vordefinierter Methoden für Listen bzw. Collections, die die ohnehin umfangreiche Standard-API für diese Typen noch einmal erheblich ausweitet und vor allem um Funktionalitäten erweitert, die für Groovy charakteristisch sind. Einige dieser vordefinierten Methoden, die sich auf die Abarbeitung aller Listen in einer Reihenfolge beziehen, haben wir uns schon weiter oben angesehen.

Die Methode `pop()` entfernt wie bei einem Stack das letzte Element aus der Liste und liefert es als Ergebnis zurück.

```
groovy> liste = ['alpha','beta','gamma','delta']
groovy> println liste.pop()
delta
```

```
groovy> println liste
 ["alpha", "beta", "gamma"]
```

Die Methode `reverse()` liefert eine neue Liste mit allen Elementen in umgekehrter Reihenfolge und lässt das Original unverändert.

```
groovy> liste = ['alpha','beta','gamma','delta']
groovy> println liste.reverse()
["delta", "gamma", "beta", "alpha"]
```

Die Methode `flatten()` löst eine verschachtelte Liste zu einer einzigen flachen Liste auf und liefert diese als neues Objekt zurück, ohne das Original zu verändern.

```
groovy> liste = [['alpha','beta'],['gamma','delta']]
groovy> println liste.flatten()
["alpha", "beta", "gamma", "delta"]
```

Die Methode `count()` stellt fest, wie oft der angegebene Wert in der aktuellen Liste vorkommt.

```
groovy> liste = ['alpha','beta','gamma','alpha']
groovy> println liste.count('alpha')
2
```

Die Methoden `min()` und `max()` suchen das kleinste bzw. größte Objekt in der Liste. Optional können Sie als Argument einen `Comparator` angeben oder eine Closure, die zwei Argumente vergleicht; in Groovy ist die zweite Form sicherlich die näherliegende. Bei einer leeren Liste ist das Ergebnis `null`.

```
groovy> liste = ['gamma','delta','BETA','alpha']
groovy> println liste.min()
BETA
groovy> println liste.min { a,b -> a.compareToIgnoreCase(b) }
alpha
```

Nach dem gleichen Prinzip funktioniert die Methode `sort()`. Sie erzeugt eine neue Liste, in der Elemente sortiert sind. Als Sortierkriterium können Sie auch hier einen `Comparator` oder eine Closure angeben. Das Original der Liste wird dabei nicht verändert.

```
groovy> liste = ['gamma','delta','BETA','alpha']
groovy> liste.sort()
groovy> println liste
["BETA", "alpha", "gamma", "delta"]
groovy> liste.sort { a,b -> a.compareToIgnoreCase(b) }
groovy> println liste
["alpha", "BETA", "gamma", "delta"]
```

Ähnlich ist auch die Methode `unique()`, die doppelte Elemente entfernt. Hier kann optional ein `Comparator` oder eine Closure entscheiden, ob zwei Elemente als gleich anzusehen sind.

```
groovy> liste = ['alpha','beta','alpha','BETA']
groovy> liste.unique()
groovy> println liste
```

```
["alpha", "beta", "BETA"]
groovy> liste.unique { a,b -> a.compareToIgnoreCase(b) }
groovy> println liste
["alpha", "beta"]
```

Mithilfe der Methode `join()` können Sie alle Elemente einer Liste zu einem einzigen String verknüpfen. Das übergebene Argument wird dabei zwischen den Elementen eingefügt.

```
groovy> liste = ['alpha','beta','gamma','delta']
groovy> println liste.join(', ')
alpha, beta, gamma, delta
```

Die Methode `groupBy()` dient dazu, die Elemente einer Liste zu gruppieren. Sie liefert eine Map zurück, deren Schlüssel die Gruppierungskriterien und deren Werte jeweils eine Liste mit den zugehörigen Listenelementen enthalten. Eine Closure liefert jeweils das Gruppierungskriterium.

In dem folgenden Beispiel werden die in der Liste enthaltenen Werte nach den Anfangsbuchstaben gruppiert.

```
groovy> liste = ['Aluminium','Beryllium','Bismut','Blei','Cadmium','Chrom','Eisen']
groovy> gruppierung = liste.groupBy{it[0]}
groovy> for (x in gruppierung) { println "$x.key = $x.value" }
A = ["Aluminium"]
C = ["Cadmium", "Chrom"]
B = ["Beryllium", "Bismut", "Blei"]
E = ["Eisen"]
```

Die Methode `inject()` ruft eine Closure für alle Elemente der Liste auf. Dabei wird als zweites Argument beim ersten Durchlauf ein als Argument für `inject()` angegebener Anfangswert übergeben und in allen weiteren Durchläufen das Ergebnis des vorherigen Durchlaufs. Das Ergebnis des letzten Durchlaufs ist dann das Ergebnis des `inject()`-Aufrufs.

Diese Methode eignet sich beispielsweise für statistische Auswertungen. Das folgende Beispiel berechnet die Summe aller Werte in einer Liste.

```
groovy> werte = [1,7,5,2,9]
groovy> println werte.inject(0) { w, x -> w + x}
24
```

Im konkreten Fall könnte man dies allerdings auch einfacher mit der `sum()`-Methode haben. Diese summiert alle Elemente; dabei wendet sie die `plus()`-Methode an. Das Ergebnis ist also dasselbe, als würden Sie aus der Liste einen Ausdruck bilden, bei dem alle Elemente durch +-Zeichen verknüpft sind. Der Ergebnistyp bleibt dabei entsprechend erhalten. Optional können Sie eine Closure übergeben; in diesem Fall wird nicht die Closure summiert, sondern die Summe der Closure-Ergebnisse.

```
groovy> println ([1,2,3].sum())
6
```

```
groovy> println ([1,2,3.0].sum())
6.0
groovy> println (['A','B','C'].sum())
ABC
groovy> println (['A','B','C'].sum {it.toLowerCase()}
abc
```

Wertebereiche

Relativ häufig hat man in der Programmierung mit einer Reihe von Daten zu tun, die sich durch eine Ober- und eine Untergrenze beschreiben lässt. Die enthaltenen Daten haben immer den Abstand 1. Typischerweise durchläuft man sie in einer Schleife, oder man prüft, ob ein Wert in der so definierten Menge enthalten ist. Groovy hat dafür einen eigenen Datentyp, der auch durch die Sprache direkt unterstützt wird: der Wertebereich (*Range*). Einen geschlossenen Wertebereich, der beide Grenzen enthält, beschreiben Sie durch die Unter- und Obergrenze, die durch zwei Punkte (`..`), die den Range-Operator bilden, verbunden sind. Bei einem rechtsoffenen Wertebereich, bei dem der rechte Wert nicht enthalten ist, steht vor dem linken Wert zusätzlich ein Kleiner-Zeichen (`<`). Die Wertebereiche können aufwärts und abwärts verlaufen. Wenn bei einem geschlossenen Wertebereich Ober- und Untergrenze gleich sind, enthält der Wertebereich nur diesen einen Wert.

```
groovy> geschlossen = 1..3
groovy> geschlossen.each { println it }
1
2
3
groovy> rechtsoffen = 1..<3
groovy> rechtsoffen.each { println it }
1
2
groovy> abwärts = 3..1
groovy> abwärts.each { println it }
3
2
1
groovy> einWert = 2..2
groovy> einWert.each { println it }
2
```

Daran, dass wir in den Beispielen die `each()`-Methode an ihnen aufrufen, erkennen Sie bereits, dass es sich bei den Groovy-Wertebereichen um iterierbare Objekte handeln muss. Alle Wertebereiche implementieren das Interface `Range`, und dieses wiederum ist eine Erweiterung des Interface `List`.

Wertebereiche können aus allen Typen gebildet werden, die sinnvolle Implementierungen für die Methoden `next()`, `previous()` und `compareTo()` bieten. Sie können sich also ohne Weiteres selbst einen Typ bauen, aus dem sich Wertebereiche bilden lassen. Die

Groovy-Standardbibliothek enthält bereits Vordefinitionen für `Character`, `Number`, `String` und `Date` (`java.util` und `java.sql`). Ober- und Untergrenze sollten vom selben Typ sein. Hier einige Beispiele – beachten Sie, dass wegen der niedrigen Priorität des Range-Operators die Wertebereich-Ausdrücke häufig eingeklammert werden müssen.

```
groovy> println 'a'..'c'                    // Character-Wertebereich
["a", "b", "c"]
groovy> println 'aaa'..'aac'                // String-Wertebereich
["aaa", "aab", "aac"]
groovy> println 1.1 .. 5.1                  // BigDecimal-Wertebereich
[1.1, 2.1, 3.1, 4.1, 5.1]
groovy> date = new Date()
groovy> println date .. (date+2)            // java.util.Date-Wertebereich
[Tue Jan 09 22:38:27 CET 2007, Wed Jan 10 22:38:27 CET 2007, Thu Jan 11 22:38:27 CET
2007]
groovy> sqldate = new java.sql.Date(date.time) // java.sql.Date-Wertebereich
groovy> println (sqldate .. (sqldate+2))
[2007-01-09, 2007-01-10, 2007-01-11]
```

Ein häufiger Anwendungsfall für Wertebereiche ist der Ersatz der konventionellen `for`-Schleife von Java, die es in Groovy erst seit Version 1.1 gibt:

```
groovy> for (i in 1..5) { println i }
```

Da ein Wertebereich ein normales Objekt ist, können Sie natürlich auch auf seine Properties und Methoden zugreifen. Allerdings können die Properties nur gelesen werden, da der Wertebereich ein unveränderlicher Typ ist.

```
groovy> w = 1.1 .. 5.1
groovy> println w.from
1.1
groovy> println w.to
5.1
```

Im Allgemeinen können Wertebereiche aus abzählbaren Typen, z.B. `1..3` oder `'a'..<'b'`, genau wie unveränderliche Listen behandelt werden. Beachten Sie, dass der Wertebereich, obwohl er lediglich durch seine Grenzen definiert ist, nur Werte im ganzzahligen Abstand von dem links stehenden Grenzwert enthält

```
groovy> w = (1.3 .. 3.5)
groovy> println w.contains(2.3)
groovy> println w.contains(2.5)
true
false
```

Maps in Groovy

Der letzte Datentyp, den Groovy durch spezielle syntaktische Leckereien unterstützt, ist die Map. Wie bei Listen gibt es auch hier wieder spezielle Operatoren zum Anlegen neuer Objekte und für den bequemen Zugriff auf die Elemente, und die Groovy-Standardbibliotheken halten verschiedene vordefinierte Methoden bereit.

Maps anlegen

Eine Map legen Sie an, indem Sie Schlüssel-Wert-Paare, durch Komma getrennt, zwischen eckigen Klammern ([]) auflisten. Zwischen dem Schlüssel und dem Wert muss dabei jeweils ein Doppelpunkt (:) stehen. Erzeugt wird dabei eine Instanz der Klasse `java.util.HashMap`.

```
groovy> richtungen = ['n':'Nord','s':'Süd','o':'Ost','w':'West']
groovy> println richtungen
["o":"Ost", "n":"Nord", "w":"West", "s":"Süd"]
```

Eine leere Map erzeugen Sie, indem Sie einfach einen Doppelpunkt zwischen eckige Klammern schreiben.

```
groovy> leereMap = [:]
groovy> println leereMap.dump()
<java.util.HashMap@0 table={null} size=0 threshold=0 loadFactor=0.75 modCount=0
entrySet=[] keySet=null values=null>
```

Wenn Sie als Schlüsselwert vor dem Doppelpunkt ein den Groovy-Namensregeln entsprechendes Wort angeben, wird dieses als String-Konstante und nicht als Variablenname angesehen. Das hat den Vorteil, dass Sie sich in den häufigen Fällen, in denen Strings die Schlüssel einer Map bilden, einfach die Anführungszeichen sparen können. Das obige Beispiel funktioniert also auch so:

```
groovy> richtungen = [n:'Nord',s:'Süd',o:'Ost',w:'West']
groovy> println richtungen
["o":"Ost", "n":"Nord", "w":"West", "s":"Süd"]
```

Der Nachteil ist, dass Sie in dem Fall, dass Sie nun tatsächlich Variablen als Schlüsselwerte verwenden möchten, diese erst in Ausdrücke umwandeln müssen, denn Ausdrücke, die nicht nur Variablennamen sind, werden als Schlüsselwerte akzeptiert. Schließen Sie in diesem Fall den Variablennamen einfach in runde Klammern ein, und schon werden die Variablen zu Ausdrücken.

```
groovy> var1 = 'n'; var2='s'; var3='o'; var4='w'
groovy> richtungen = [(var1):'Nord',(var2):'Süd',(var3):'Ost',(var4):'West']
groovy> println richtungen
["o":"Ost", "n":"Nord", "w":"West", "s":"Süd"]
```

Groovy erzeugt aus diesen speziellen Operatoren standardmäßige Instanzen der Klasse `java.util.HashMap`. Wenn Sie einen anderen Map-Typ benötigen, verwenden Sie einfach die `HashMap` als Argument für den Konstruktor.

```
groovy> linkedMap = new LinkedHashMap([n:'Nord',s:'Süd',o:'Ost',w:'West'])
groovy> println linkedMap.getClass().name
java.util.LinkedHashMap
```

Auf Map-Elemente zugreifen

Die folgenden Operatoren und Methoden gelten für alle Klassen, die `java.util.Map` implementieren, sowie für alle von `java.util.Hashtable` abgeleiteten Klassen.

Groovy ermöglicht Ihnen nicht nur wie üblich, mit den Methoden `get()` und `put()` auf die Elemente einer Map zuzugreifen, sondern dafür die Indexoperatoren zu verwenden.

```
groovy> richtungen = [n:'Nord',s:'Süd',o:'Ost',w:'West']
groovy> richtungen['nw'] = 'Nordwest'
groovy> println richtungen['nw']
Nordwest
```

Wenn die Schlüssel wie in diesem Fall Strings sind und den Groovy-Namensregeln gehorchen, können Sie auf die Map-Elemente auch zugreifen wie auf die Properties eines Objekts.

```
groovy> richtungen.oso = 'Ost-Südost'
groovy> println richtungen.oso
Ost-Südost
```

Sogar wenn die Namensregeln nicht eingehalten werden, können Sie diese Notation anwenden. Sie müssen in diesem Fall nur den Elementschlüssel in Anführungszeichen setzen.

```
groovy> richtungen.'o-so' = 'Ost-Südost'
groovy> println richtungen.'o-so'
Ost-Südost
```

Das geht sogar mit interpolierten GStrings.

```
groovy> keyOst = 'o'
groovy> keySüdost = 'so'
groovy> println richtungen."$keyOst-$keySüdost"
Ost-Südost
```

Die Property-Notation für Map-Elemente kann Programme unter Umständen viel übersichtlicher machen. Dies gilt beispielsweise, wenn Sie in Ihrer Map numerische Werte haben, mit denen Sie auch rechnen möchten.

```
groovy> rechteck = [breite:2.0, höhe:4.5]
groovy> rechteck.breite++
groovy> rechteck.fläche = rechteck.breite * rechteck.höhe
===> 13,5
```

Wir wollen Sie nicht ständig mit Codevergleichen zwischen Java und Groovy langweilen, aber hier lohnt es sich doch einmal, ein äquivalentes Programmstück in Java anzusehen:

```
//java
// 1. Zeile
Map rechteck = new HashMap();
rechteck.put("breite",new BigDecimal("2.0"));
rechteck.put("höhe",new BigDecimal("4.5"));
// 2. Zeile
rechteck.put("breite",rechteck.get("breite").add(new BigDecimal(1)));
// 3. Zeile
rechteck.put("fläche",rechteck.get("breite").multiply(rechteck.get("höhe")));
```

Da auch Objekte vom Typ `Properties` das Interface `Map` implementieren, gilt diese Notation ebenfalls für die System-Properties. Wenn die Namen der Properties Punkte oder andere Zeichen enthalten, die nicht den Groovy-Namensregeln entsprechen, müssen sie in Anführungszeichen gesetzt werden.

```
groovy> println System.properties.'java.version'
groovy> println System.properties.'os.name'
1.5.0_11
Windows XP
```

Die Möglichkeit, auf Map-Elemente wie auf Properties eines Objekts zuzugreifen, macht die Programmierung mancher Klassen überflüssig, die nur als Container für Daten dienen sollen und keine spezifische Funktionalität enthalten. In Kapitel 7 zeigen wir Ihnen eine kleine Erweiterung zu den Maps, durch die Sie ihnen sogar eigene Funktionalität verleihen können.

Auf eine Besonderheit müssen wir Sie noch hinweisen: Im Gegensatz zu den meisten Objekten können Sie die Klasse einer Map oder einer `Hashtable` nicht einfach über die Property `class` abfragen.

```
groovy> println "ein String".class
class java.lang.String
groovy> println rechteck.class
null
```

Die Ursache dafür besteht darin, dass aufgrund entsprechender vordefinierter Methoden alle Property-Abfragen bei Maps auf Abfragen nach den Map-Elementen umgelenkt werden. Aus `rechteck.class` macht Groovy also `rechteck.get('class')`, und ein Element namens »class« gibt es in dieser Map nicht. Wenn Sie also in einem Programm den Typ eines Objekts erfahren möchten, ist es in der Regel besser, die Methode `getClass()` anstelle der Property-Abfrage zu verwenden.

```
groovy println rechteck.getClass()
class java.util.HashMap
```

Vordefinierte Methoden

Auch für Maps enthält die Groovy-Laufzeitbibliothek eine Reihe vordefinierter Methoden. Zum Teil dienen sie dazu, die obigen vereinfachten Zugriffsmöglichkeiten zu implementieren, zum Teil stellen sie Ihnen etwas zusätzliche Funktionalität zur Verfügung.

Sicher eine der in Java am häufigsten vermissten Methoden für Maps ist die `get()`-Methode mit Vorgabewert.

```
groovy> richtungen = [n:'Nord',s:'Süd',o:'Ost',w:'West']
groovy> schlüssel = 'oso'
groovy> println richtungen.get(schlüssel,'unbekannte Richtung')
unbekannte Richtung
```

Mithilfe der Methode `subMap()` können Sie eine neue Map bilden, die alle Schlüsselwerte enthält, die sich auch in der übergebenen `Collection` befinden.

```
groovy> liste = ['n','s']
groovy> println richtungen.subMap(liste)
```

Die in Groovy für alle Objekte definierten iterativen Methoden sind für Maps so implementiert, dass sie über Objekte vom Typ `java.util.Map.Entry` iterieren.

```
groovy> richtungen = [n:'Nord',s:'Süd',o:'Ost',w:'West']
groovy> richtungen.each { println it }
o=Ost
n=Nord
w=West
s=Süd
```

Folgendes Beispiel sucht alle Elemente, deren Wert auf »st« endet, und liefert sie als neue Map.

```
groovy> auswahl = richtungen.findAll { it.value ==~ /.*st/ }
groovy> println auswahl
["w":"West", "o":"Ost"]
```

Allerdings gibt es noch einige Besonderheiten. So kann etwa die Methode `each()` auch zwei Parameter haben. Wenn dies der Fall ist, werden ihr bei jedem Durchgang der Schlüssel und der Wert des jeweiligen Map-Elements einzeln übergeben.

```
groovy> richtungen = [n:'Nord',s:'Süd',o:'Ost',w:'West']
groovy> richtungen.each { key, value -> println "Schlüssel=$key, Wert=$value" }
Schlüssel=o, Wert=Ost
Schlüssel=n, Wert=Nord
Schlüssel=w, Wert=West
Schlüssel=s, Wert=Süd
```

Dateien und Datenströme

Ein weiterer großer Block von vordefinierten Methoden hat mit der Eingabe und Ausgabe von Daten bzw. mit der Navigation im Dateisystem zu tun. Sie sind für Dateien sowie für die Eingabe- und Ausgabeströme, Reader und Writer definiert und sorgen dafür, dass manche Ein- oder Ausgabeoperation mit wenig Code erledigt werden kann. Außerdem nehmen sie Ihnen in vielen Fällen eine verpflichtende korrekte Behandlung von Exceptions ab, die die Programmierung mitunter recht umständlich macht.

Navigieren im Dateisystem

Die Klasse `java.io.File` repräsentiert in Java einen Datei- oder Verzeichnisnamen und bietet darüber hinaus einige Methoden zum Konvertieren dieser Namen sowie zur Navigation in Dateisystemen. Die Groovy-Laufzeitbibliothek definiert eine Reihe zusätzlicher Methoden für diese Klasse. Sie erlauben einerseits den direkten Zugriff auf die Daten; darauf kommen wir etwas weiter unten zu sprechen. Daneben gibt es einige Methoden, die das Operieren im Dateisystem auf Groovy-typische Weise mit Closures erlauben.

Angenommen, das Wurzelverzeichnis Ihrer Groovy-Installation liegt ebenfalls unter *C:\java\groovy_1.1*, dann können Sie mit der `File`-Methode `eachFile()` wie im folgenden Beispiel dessen Inhalt in einer Schleife durchlaufen:

```
groovy> f = new File('C:/java/groovy-1.1')
groovy> f.eachFile { println it }
C:\java\groovy-1.1\bin
C:\java\groovy-1.1\conf
C:\java\groovy-1.1\docs
C:\java\groovy-1.1\embeddable
C:\java\groovy-1.1\groovy-1.1.jar
C:\java\groovy-1.1\lib
C:\java\groovy-1.1\LICENSE.txt
```

In Unix-Dateisystemen bleiben dabei die dort üblichen Verzeichniseinträge für das aktuelle und das übergeordnete Verzeichnis (».« und »..«) unbeachtet.

Die Methode `eachDir()` tut im Prinzip dasselbe, beachtet auf ihrem Wege allerdings nur Unterverzeichnisse:

```
groovy> f.eachDir { println it }
C:\java\groovy-1.1\bin
C:\java\groovy-1.1\conf
C:\java\groovy-1.1\docs
C:\java\groovy-1.1\embeddable
C:\java\groovy-1.1\lib
```

Eine Methode, mit der nur echte Dateien (also alles außer Verzeichnissen) gelistet werden, fehlt leider. Weiter unten werden Sie allerdings sehen, dass sich dies mit sehr wenig Mühe ausgleichen lässt.

Eine weitere Variante der Schleife gibt Ihnen die Möglichkeit, einen Vergleichsausdruck für den Dateinamen anzugeben:

```
groovy> f.eachFileMatch (~"...") { println it }
C:\java\groovy-1.1\bin
C:\java\groovy-1.1\lib
```

Das erste Argument der vordefinierten Methode `eachFileMatch()` akzeptiert einen Ausdruck, dessen Typ die Methode `isCase()` implementiert. Diese Methode ist eigentlich für `case`-Verzweigungen vorgesehen und liefert normalerweise dasselbe Ergebnis wie `equals()`. Im obigen Beispiel verwenden wir als erstes Argument einen regulären Ausdruck, bei dem die `isCase()`-Methode einen Mustervergleich durchführt. (Sie erinnern sich: Der Operator `~` vor einem String bewirkt, dass der String in ein `Pattern`-Objekt umgewandelt wird. `~"..."` ist also ein Muster für einen aus drei beliebigen Zeichen bestehenden String.)

Wir hätten stattdessen auch eine Closure verwenden können, bei der `isCase()` die Closure selbst, die ein Boolesches Ergebnis liefern muss, ausführt:

```
groovy> f.eachFileMatch { it.length()==3 } { println it }
C:\java\groovy-1.1\bin
C:\java\groovy-1.1\lib
```

Die letzte Methode dieser Gruppe, `eachFileRecurse()`, durchläuft, wie der Name schon ahnen lässt, den gesamten Verzeichnisbaum unterhalb des angegebenen Wurzelverzeichnisses rekursiv. Das folgende kleine Skript ermittelt die Anzahl der zur Groovy-Installation gehörenden Dateien – ohne Berücksichtigung der Verzeichnisse:

```
groovy> anzahl = 0
groovy> f.eachFileRecurse { if (it.file) anzahl++ }
groovy> println anzahl
854
```

Textdaten

Java unterscheidet bei seinen Ein- und Ausgabeoperationen sorgfältig zwischen Text- und Binärdaten. Während Letztere einfach nur uninterpretierte Rohdaten in Form von Bytes darstellen, werden Textdaten innerhalb der Java VM als 16-bit-Unicode-Character dargestellt. In Dateien werden die Textdaten unterschiedlich kodiert, wobei in Java die gewünschte Kodierung beim Zugriff auf die Datei angegeben wird bzw. eine betriebssystemabhängige Vorbelegung stattfindet. Da bei der Ein- und Ausgabe letztendlich immer nur Rohdaten transportiert werden, ist eine Kodierung bzw. Dekodierung der Textdaten erforderlich. Wenn man bei plattformübergreifender Programmierung in Java nicht darauf achtet, dass dabei immer die gleichen Zeichenkodierungen verwendet werden, findet man schnell anstatt der erwarteten Umlaute und Sonderzeichen umgekehrte Fragezeichen und merkwürdige Sequenzen vor. Das Angenehme an Groovy ist, dass es uns weitgehend von den Kodierungsproblemen befreit und gleichzeitig noch viele Lese- und Schreiboperationen mit wenigen Programmzeilen erledigen lässt.

Auf komplette Textdateien zugreifen

Sehr einfach ist es, wenn Sie eine Textdatei in einem Stück aus einem String heraus schreiben wollen.

```
groovy> text = 'Erste Zeile.\nZweite Zeile.\nDritteZeile.\n'
groovy> new File('Beispiel.txt').write(text)
```

Das Öffnen der Datei, das Schreiben, Schließen und die Kontrolle, dass Fehler richtig behandelt werden, also alles, was die Arbeit mit Dateien immer etwas mühsam macht, übernimmt für Sie die für die Klasse `File` vordefinierte Methode `write()`.

Natürlich lassen sich die Daten auch wieder in einem Rutsch einlesen. Dies geht mit der Methode `getText()`, die wir selbstverständlich auch als Property ansprechen können.

```
groovy> println (new File('Beispiel.txt').text)
Erste Zeile.
Zweite Zeile.
DritteZeile.
```

Alternativ besteht die Möglichkeit, die Datei mittels der Methode `readLines()` nicht in einen einzigen String, sondern in eine Liste von Strings einzulesen, wobei jedes Listenelement eine Zeile enthält.

```
groovy> liste = new File('Beispiel.txt').readLines()
groovy> println liste
["Erste Zeile.", "Zweite Zeile.", "DritteZeile."]
```

Zeichensätze berücksichtigen

Oben haben wir weder beim Lesen noch beim Schreiben den zu verwendenden Zeichensatz angegeben. Wie unter Java üblich, wird dann der Systemzeichensatz verwendet; auf westeuropäischen Arbeitsplatzrechnern ist dies normalerweise *Latin-1* (unter Windows als Cp1252 und unter anderen Betriebssystemen als ISO-8859-1 bezeichnet), Linux verwendet meist UTF-8. Welchen Zeichensatz Sie haben, bekommen Sie leicht mit folgender Groovy-Zeile heraus:

```
groovy> System.properties.'file.encoding'
===> Cp1252
```

Sobald Sie Dateien austauschen wollen oder Programme schreiben, die auf einem Server laufen, müssen Sie auf das richtige Encoding achten, damit Umlaute und Sonderzeichen korrekt behandelt werden. Bei (fast) allen Methoden, die Strings in Bytes und umgekehrt umwandeln, kann daher das Encoding angegeben werden. Wir probieren dies aus, indem wir einen Satz mit vielen Umlauten verwenden und die Unicode-Kodierung UTF-8 anwenden.

```
groovy> text = 'Zwölf Boxkämpfer jagen Viktor quer über den großen Sylter Deich.'
groovy> new File('Beispiel.txt').write(text,'UTF8')
groovy> println (new File('Beispiel.txt').getText('UTF8'))
Zwölf Boxkämpfer jagen Viktor quer über den großen Sylter Deich.
```

Merkwürdigerweise funktioniert dies mit Groovy beim Lesen auch ohne Angabe des Zeichensatzes:

```
groovy> println (new File('Beispiel.txt').text)
Zwölf Boxkämpfer jagen Viktor quer über den großen Sylter Deich.
```

Mit einem normalen Java-Programm würde dies nicht ohne Weiteres funktionieren – statt der Umlaute und des *ß*-Zeichens würden Sie merkwürdige Zeichenkombinationen vorfinden. Der Grund für diesen Unterschied besteht darin, dass Groovy beim Lesen einer Datei von sich aus versucht, das Encoding herauszubekommen, wenn Sie es nicht explizit angeben. Dazu bedient es sich einer Klasse in der Standardbibliothek mit dem Namen `groovy.util.CharsetToolkit`, die mithilfe der ersten vier KByte einer Textdatei deren Zeichensatz ermittelt. Probieren wir dies einmal selbst aus:

```
groovy> ctk = new CharsetToolkit(new File('Pangramm.txt'))
groovy> println ctk.charset
UTF-8
```

Treffer. Das kann natürlich theoretisch schiefgehen, wenn in den ersten 4.000 Zeichen einer Datei keine Umlaute und Sonderzeichen vorkommen, an denen sich der Zeichensatz ermitteln lässt. Die Erfahrung zeigt allerdings, dass dies in der Praxis extrem unwahrscheinlich ist.[2]

Versuchen wir es noch einmal mit der Textdatei aus dem ersten Versuch:

```
groovy> text = 'Erste Zeile.\nZweite Zeile.\nDritteZeile.\n'
groovy> new File('Beispiel2.txt').write(text,'UTF-8')
groovy> println (new CharsetToolkit(new File('Beispiel2.txt')).charset)
windows-1252
```

Obwohl wir die Datei mit UTF-8 geschrieben haben, meint das `CharsetToolkit`, es sei der Zeichensatz »windows-1252« gewesen – ein Synonym für Cp1252. Dies liegt aber daran, dass dieser Text weder Umlaute noch Sonderzeichen enthält und es daher völlig egal ist, welcher Zeichensatz verwendet wird; in dem Fall entscheidet sich das `CharsetToolkit` für den aktuellen Standardzeichensatz.

Das `CharsetToolkit` verfügt übrigens noch über ein paar weitere praktische Methoden in Bezug auf Zeichensätze; in Anhang B ist die Klasse ausführlicher erläutert.

Texte aus anderen Quellen lesen

Die Methode `getText()` ist nicht nur für Dateien, sondern auch für verschiedene andere Datenquellen wie Streams, Reader, Prozesse und auch URLs definiert. Letzteres können wir bequem dafür verwenden, Dateien über das Netzwerk zu laden. Das folgende kurze Beispiel liest eine Webseite und speichert sie als Datei ab.

```
groovy> seite = new URL('http://www.oreilly.de/index.html').text
groovy> new File('oreilly-home.html').write(seite)
```

Öffnen Sie anschließend die Datei mit einem Browser, und Sie finden tatsächlich die deutsche Homepage des O'Reilly Verlags vor – etwas abgemagert, denn es fehlen natürlich das richtige Stylesheet und die Bilddateien, aber deutlich erkennbar (siehe Abbildung 5-1).

Natürlich funktioniert dies nur, wenn Sie am Internet angeschlossen sind. Müssen Sie sich über einen Netzwerk-Proxy verbinden, dürfen Sie nicht vergessen, zuvor die entsprechenden System-Properties zu setzen.

```
groovy> System.properties.proxySet = 'true'
groovy> System.properties.proxyHost = 'proxy.somedomain.org'
groovy> System.properties.proxyPort = '3128'
```

2 Ein Fachgutachter merkt mit Recht an, dass man sich als Mathematiker nicht darauf verlassen sollte, dass, wenn bestimmte Zeichen in einem Teil eines Texts nicht auftauchen, dies auch bei dem Rest so ist. Es sind schon ganze Romane ohne den Buchstaben *e* geschrieben worden (so *La Disparition* von Georges Perec; deutsch *Anton Voyls Fortgang*, auch ohne *e*). Übertragen heißt dies: Natürlich ist es auch möglich, einen langen Text ohne Umlaute und Sonderzeichen zu schreiben – allerdings brauchen wir diesen Text dann auch nicht zu kodieren.

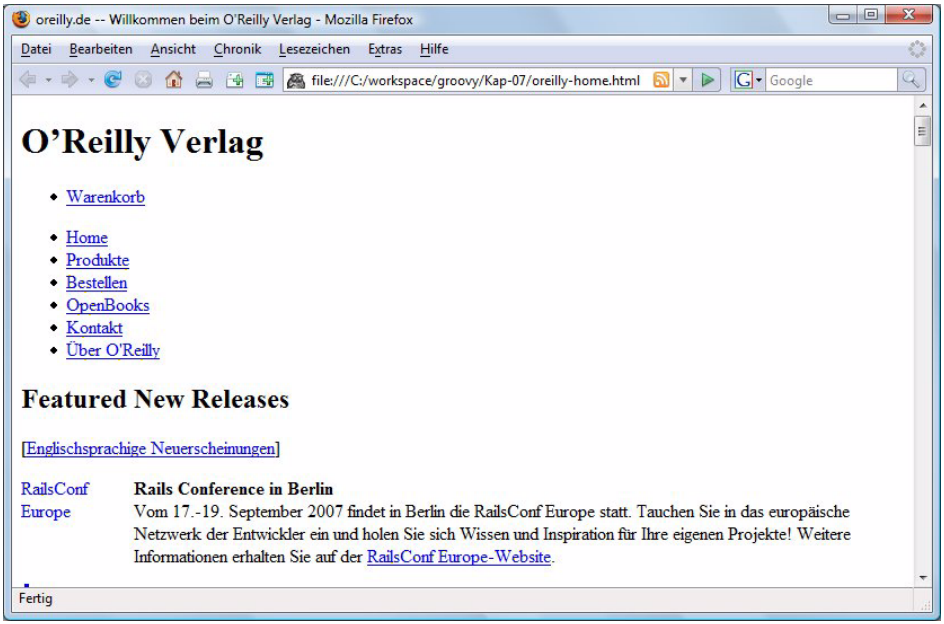


Abbildung 5-1: Die Homepage des O'Reilly Verlags wurde erfolgreich abgespeichert.

Eine weitere mögliche Quelle für Eingabedaten sind Prozesse. Im folgenden Beispiel führt die vordefinierte `String`-Methode `execute()` den Inhalt des Strings als Systembefehl aus; der Rückgabewert ist eine Instanz des Typs `java.lang.Process`. Und dafür gibt es wiederum eine vordefinierte Methode `getText()`, mit der die gesamte Standardausgabe dieses Prozesses ausgelesen werden kann. Das folgende Beispiel liest die aktuelle Netzwerkkonfiguration unter Windows.

```
groovy> ipconfig = "ipconfig".execute().text
groovy> println ipconfig
Windows-IP-Konfiguration
Drahtlos-LAN-Adapter Drahtlosnetzwerkverbindung:
   Verbindungsspezifisches DNS-Suffix:
   Verbindungslokale IPv6-Adresse  . : fe80::1d2c:2980:5be5:f725%10
   IPv4-Adresse  . . . . . . . . . . : 192.168.1.2
   Subnetzmaske  . . . . . . . . . . : 255.255.255.0
   Standardgateway . . . . . . . . . : 192.168.1.1
...
```

Texte zeilenweise lesen und schreiben

In vielen Fällen werden Texte – vor allem, wenn sie länger werden können – nicht im Ganzen, sondern Zeile für Zeile ein- oder ausgegeben. Auch dies vereinfacht Groovy durch entsprechende vordefinierte Methoden zur Klasse `File`.

Wenn Sie eine Datei zeilenweise lesen möchten, benutzen Sie am besten die Methode `eachLine()`. Sie nimmt eine Closure als Argument an, die jede einzelne Zeile des Texts übergeben bekommt. Lassen Sie uns auf diese Weise die dreizeilige Datei von oben auslesen.

```
groovy> zaehl = 0
groovy> new File('Beispiel.txt').eachLine { zeile ->
groovy>    println "${++zaehl} - $zeile"
groovy> }
1 - Erste Zeile.
2 - Zweite Zeile.
3 - DritteZeile.
```

Dies geht auch mit anderen Datenquellen und bietet sich insbesondere dann an, wenn das Lesen der Daten ein längerer Prozess ist, so dass es hilfreich ist, wenn man schon mal einige Zeilen zur Verfügung hat, obwohl der Lesevorgang noch im Gang ist.

Angenommen, Sie wollten mithilfe des Systembefehls `tracert` unter Windows das Routing eines Webseitenabrufs nachvollziehen. Dies zieht sich normalerweise etwas hin, daher kann es hier sinnvoll sein, das Ergebnis zeilenweise einzulesen.

```
groovy> ['tracert','www.google.de'].execute().in.eachLine { println it }
Routenverfolgung zu www.l.google.com [209.85.129.147] über maximal 30 Abschnitte:
  1    35 ms    35 ms    35 ms  209.85.249.180
  2    34 ms    35 ms    35 ms  72.14.232.201
  3     *       35 ms    34 ms  de-cix10.net.google.com [80.81.192.108]
  4    38 ms    35 ms    35 ms  72.14.233.210
  5    34 ms    35 ms    37 ms  fk-in-f147.google.com [209.85.129.147]
```

Tatsächlich erscheinen die einzelnen Stationen Zeile für Zeile einzeln auf dem Bildschirm. Hier sind aber noch ein paar Erklärungen fällig. Die vordefinierte Methode `execute()` kann auch auf String-Arrays angewendet werden; in diesem Fall ist das erste Array-Element der Befehl, und alle weiteren Elemente enthalten die Argumente. Zurückgeliefert wird eine `java.lang.Process`-Instanz. Zu `Process` gibt es aber leider keine vordefinierte Methode `eachLine()`, mit der wir deren Ausgabe direkt zeilenweise auslesen könnten. Wohl aber gibt es eine Methode `getIn()`, die uns die Standardausgabe des Prozesses als `InputStream` gibt, und zu diesem wiederum ist ein `eachLine()` definiert.

Für den umgekehrten Weg – Daten zeilenweise zu schreiben – bietet sich eine Gruppe von vordefinierten Methoden an, die eine Datei oder eine andere Datenquelle öffnen, einer als Argument übergebenen Closure einen Datenstrom übergeben und am Ende die Datei in jedem Fall auch wieder schließen. Innerhalb der Closure können Sie mit dem Datenstrom im Prinzip machen, was Sie wollen; um eine angemessene Behandlung der Ressource, d.h. das Öffnen und Schließen sowie die Fehlerbehandlung, brauchen Sie sich nicht zu kümmern. Diese Art von Methoden gibt es nicht nur für Schreiboperationen, sie bieten sich hier aber besonders an, weil ein Vorgehen wie bei der bequemen Methode `eachLine()` in umgekehrter Richtung nicht anwendbar ist.

```
groovy> new File('Beispiel3.txt').withWriter { out ->
groovy>    'ABC'.each { out.writeLine "Zeile $it" }
groovy> }
groovy> println (new File('Beispiel3.txt').text)
Zeile A
Zeile B
Zeile C
```

Hier bekommt die Closure in `withWriter()` einen `BufferedWriter` übergeben, der in die Datei *Beispiel3.txt* schreibt. Innerhalb dieser Closure schreiben wir mithilfe einer `each()`-Schleife drei Zeilen in den Writer. Das ist auf diese Weise möglich, weil es für den `BufferedWriter` eine vordefinierte Methode `writeLine()` gibt. Nach der Beendigung der Closure schließt `withWriter()` den Writer ordnungsgemäß, sodass wir uns um diese Einzelheiten nicht kümmern müssen.

Filter und Transformationen

Wir möchten Sie gern noch mit ein paar Methoden bekannt machen, die sehr praktisch sind, wenn Sie Textdaten während des Lesens oder Schreibens konvertieren oder bestimmte Zeilen oder Zeichen herausfiltern möchten.

Die für beliebige Reader definierte Methode `transformLine()` nimmt einen Writer und eine Closure als Argument. Sie übergibt der Closure jede Zeile einzeln und schreibt das jeweilige Ergebnis in den Writer. Das folgende Beispiel kopiert eine Textdatei und stellt dabei jeder Zeile eine Zeilennummer in Klammern voran.

```
groovy> reader = new FileReader(new File('Beispiel.txt'))
groovy> writer = new FileWriter(new File('BeispielNummeriert.txt'))
groovy> zaehler = 0
groovy> reader.transformLine (writer) { "(${++zaehler}) $it" }
groovy> println (new File('BeispielNummeriert.txt').text)
(1) Erste Zeile.
(2) Zweite Zeile.
(3) Dritte Zeile.
```

Auch hier brauchen Sie übrigens keine Sorge haben, dass der Reader oder der Writer offen bleiben könnte. Die Methode `transformLine()` schließt beide ordnungsgemäß.

Analog gibt es eine Methode `transformChar()`, sie funktioniert genau so wie `transformLine()`, bekommt aber anstelle der Zeilen jedes einzelne Zeichen als Argument übergeben.

In ähnlicher Weise funktionieren einige Methoden, mit denen Daten zeilenweise gefiltert werden können. Die für `Reader` und `File` definierte Methode `filterLine()` etwa nimmt eine Closure als Argument an, die für jede Zeile entscheidet, ob sie übertragen werden soll oder nicht. In dem folgenden Beispiel werden nur Zeilen durchgelassen, die den Buchstaben »t« genau einmal enthalten.

```
groovy> writer = new FileWriter('BeispielGefiltert.txt')
groovy> new File('Beispiel.txt').filterLine (writer) { it.count('t')==1 }
groovy> println (new File('BeispielGefiltert.txt').text)
Erste Zeile.
Zweite Zeile.
```

Writable-Objekte

Wie viele andere vordefinierte Methoden für Ein- und Ausgabeoperationen gibt es auch die Methode `filterLine()` in verschiedenen Varianten. Das folgende Skript verwendet eine Version der Methode, die für `Reader` definiert ist. Sie erhält aber keinen Writer als Argument, sondern liefert stattdessen ein spezielles Objekt, das `groovy.lang.Writable` implementiert. Dieses Interface ist eine spezielle Schöpfung der Groovy-Bibliotheken, die Schreiboperationen besonders effizient machen soll. Ein `Writable`-Objekt implementiert die Methode `writeTo()`, die einen Writer annimmt und den Inhalt des Objekts in diesen Writer hineinschreibt. Der Vorteil besteht darin, dass der Inhalt des Objekts nicht vor dem Schreiben insgesamt in einen String verwandelt wird, sondern gewissermaßen fließend ausgegeben werden kann.

```
groovy> reader = new FileReader('Beispiel.txt')
groovy> filter = reader.filterLine { it.count('t')==1 }
groovy> writer = new FileWriter('BeispielGefiltert.txt')
groovy> filter.writeTo(writer)
groovy> writer.close()
groovy> println (new File('BeispielGefiltert.txt').text)
Erste Zeile.
Zweite Zeile.
```

Beachten Sie, dass wir hier den Writer mit `close()` selbst schließen müssen und dies nicht einfach einer Groovy-Methode überlassen können. Die vierte Zeile könnten wir mit demselben Ergebnis auch so schreiben:

```
groovy> writer.write(filter)
```

Dies liegt daran, dass wir eine vordefinierte `write()`-Methode für Writer haben, die ein `Writable`-Argument erkennt und entsprechend behandelt, nämlich bei ihm `writeTo()` aufruft.

`Writable`-Objekten begegnen Sie beim Durchstöbern der Groovy-Standardbibliothek des Öfteren. Beispielsweise können Sie auch ein `File`-Objekt direkt zu einem `Writable` machen. Dies ergibt eine einfache Möglichkeit zum Kopieren von Dateien, wie hier von *Beispiel.txt* nach *BeispielKopie.txt*:

```
groovy> new FileWriter('BeispielKopie.txt').withWriter { writer ->
groovy>     writer.write( new File('Beispiel.txt').asWritable() )
groovy> }
groovy> println (new File('BeispielKopie.txt').text)
Erste Zeile.
Zweite Zeile.
Dritte Zeile.
```

Rohdaten

Ähnliche Möglichkeiten wie für Reader und Writer zum Lesen und Schreiben von Textdaten gibt es auch für Input- und Output-Streams zum Lesen und Schreiben unkodierter Rohdaten. Während aber bei einem Reader oder Writer in der Regel mit Zeilen (manchmal auch mit Zeichen) gearbeitet wird, sind es bei normalen Streams immer Bytes und bei serialisierten Daten immer komplette Objekte oder einzelne primitive Datenelemente.

Binärdaten komplett lesen und schreiben

Um eine Datei komplett einzulesen, können Sie die für `File`-Objekte vordefinierte Methode `readBytes()` verwenden. Sie öffnet die Datei, liest sie komplett in ein `byte`-Array ein und sorgt auch wiederum dafür, dass die Datei in jedem Fall geschlossen wird. Im folgenden Beispiel lesen wir die von den obigen Versuchen bekannte Datei *Beispiel.txt* als Rohdaten ein.

```
groovy> daten = new File('Beispiel.txt').readBytes()
groovy> println daten
[69, 114, 115, 116, 101, 32, 90, 101, 105, 108, 101, 46, ...]
```

Für den umgekehrten Weg hat Groovy leider noch keine vergleichbare vordefinierte Methode im Angebot. Allerdings ist es mit den im Folgenden dargestellten Mitteln recht mühelos möglich, Binärdaten zu schreiben.

Auch hierfür stehen nämlich Methoden zur Verfügung, die einen Stream öffnen und einer Closure übergeben. Innerhalb der Closure haben Sie die volle Kontrolle über den Stream, und nach Beendigung der Closure wird der Stream wieder ordnungsgemäß geschlossen.

Das oben eingelesene `byte`-Array wieder in eine Datei zu schreiben gestaltet sich dadurch äußerst simpel:

```
groovy> new File('Beispiel4.txt').withOutputStream { out -> out.write(daten) }
```

Und da dies nur eine Zeile braucht, benötigen wir dafür eigentlich keine spezielle Methode. Wenn wir dies und die zuvor betrachtete Methode zusammenfassen, brauchen wir zum Kopieren einer Binärdatei nur eine Zeile:

```
groovy> file1 = new File('Beispiel.txt')
groovy> file2 = new File('BeispielKopie.txt')
groovy> file2.withOutputStream { it.write(file1.readBytes()) }
```

Diese Lösung kopiert allerdings die gesamte Datei in den Hauptspeicher, bevor sie wieder ausgegeben wird. Wie unten beschrieben wird, ist das bei großen Dateien nicht sinnvoll.

Daten über einen `InputStream` zu lesen ist nicht in einem Rutsch möglich, zumindest wenn die Länge der Datei nicht vorhergesagt werden kann. Versuchsweise lesen wir die Datei einmal in 10-Byte-Portionen ein.

```
groovy> daten = new byte[10]
groovy> new File('Beispiel4.txt').withInputStream { inp ->
groovy>     def len=0
groovy>     while ((len=inp.read(daten)) > 0) { println (daten[0..<len]) }
groovy> }
[69, 114, 115, 116, 101, 32, 90, 101, 105, 108]
[101, 46, 10, 90, 119, 101, 105, 116, 101, 32]
...
```

Hier wenden wir die standardmäßige `read()`-Methode des `InputStream` an, die immer maximal so viele Daten einliest, wie in das als Puffer übergebene `byte`-Array hineinpassen und die Anzahl der gelesenen Bytes zurückgibt. Das ist für Programmierer, die an Groovy gewöhnt sind, zwar relativ umständlich, aber immerhin sparen wir uns das ganze Exception-Handling, das die Arbeit mit Dateien in Java oft recht mühsam macht.

Daten byteweise lesen

Groovy bietet noch ein Bonbon für den Fall, dass einmal ein Stream Byte für Byte gelesen werden muss. Die für `File` vordefinierte Methode `eachByte()` lässt uns eine Closure angeben, die mit jedem Byte der Datei einzeln aufgerufen wird. Sehen wir uns auch dies einmal an, indem wir die Bytes in der Datei zählen.

```
groovy> anzahl = 0
groovy> new File('Beispiel.txt').eachByte { anzahl++ }
groovy> println "Die Datei ist $anzahl Bytes lang."
Die Datei ist 41 Bytes lang.
```

Dasselbe ist möglich, wenn Sie bereits einen geöffneten `InputStream` in der Hand haben.

```
groovy> anzahl = 0
groovy> input = new FileInputStream (new File('Beispiel.txt'))
groovy> input.eachByte { anzahl++ }
groovy> println "Die Datei ist $anzahl Bytes lang."
Die Datei ist 41 Bytes lang.
```

Auch hier können Sie sicher sein, dass der `InputStream` am Ende der Schleife geschlossen wird, und brauchen sich nicht mit dem Abfangen möglicher IO-Exceptions zu beschäftigen – es sei denn, Sie müssen mit dem Auftreten eine Exception rechnen; in diesem Fall können Sie diese natürlich wie gewohnt abfangen, indem Sie die kritischen Zeilen in einen `try-catch`-Block einschließen.

Sie haben in diesem Kapitel einen Überblick darüber bekommen, welche Standard-Java-Klassen durch vordefinierte Methoden in ihrem Funktionsumfang erweitert worden sind, so dass sie – insbesondere unter Anwendung der Groovy-eigenen Möglichkeiten wie Closures und überladbaren Operatoren – in Groovy-Programmen einfacher, übersichtlicher und auch sicherer verwendet werden können, und wie diese Klassen durch die Sprache unterstützt werden. Wir wenden uns nun einigen weiteren Goodies zu, die Sie mit Groovy frei Haus geliefert bekommen.

KAPITEL 6

Highlights der Groovy-Standardbibliothek

Mit Ihrer Groovy-Installation erhalten Sie eine umfangreiche Klassenbibliothek geliefert. Sie wird zum Teil dazu benötigt, Groovy-Programme überhaupt lauffähig zu machen. Sie umfasst aber auch viele Klassen und Interfaces, die dazu dienen, Ihnen das Programmieren generell zu vereinfachen und neue, häufig benötigte Hilfsmittel zu integrieren. Oft ergänzen sie das ohnehin schon sehr breite Angebot an hilfreichen Klassen im JDK, um diese den neuen Möglichkeiten der Sprache Groovy zugänglich zu machen und ihre Benutzung dadurch – zum Teil gravierend – zu vereinfachen. So stellt die Bibliothek in weiten Teilen eine Art Abstraktions- und Integrationsschicht über dem JDK und einigen zusätzlichen APIs dar. Wie Sie sehen, bietet sie aber mit ihren Buildern und Template-Engines auch Neues.

In diesem Kapitel wollen wir Ihnen einen Überblick darüber geben, wie die Groovy-Standardbibliothek aufgebaut ist und in welchem Package was zu finden ist. Dann steigen wir tiefer in vier einzelne Themenbereiche ein, die noch nicht in den übrigen Kapiteln behandelt wurden, und zwar:

- Datenbanken,
- XML-Dokumente,
- Webanwendungen und
- grafische Benutzeroberflächen.

Die Themen sind von besonderem Interesse, weil sie entweder neue oder elegantere Ansätze der Programmierung eröffnen oder weil sie die Möglichkeiten der Sprache auf interessante Weise ausschöpfen. Ergänzend sei auf Anhang B verwiesen, in dem die wichtigsten Klassen und Interfaces der Groovy-Bibliothek ausführlich dokumentiert sind.

Struktur der Groovy-Bibliotheken

Sie werden beim Programmieren mit Groovy öfter einmal in die Situation kommen, eine Klasse für einen speziellen Zweck zu suchen oder sich generell in den Bibliotheken orientieren zu wollen. Wenn Sie einmal einen Blick in die *groovy.jar* werfen, fällt Ihnen sofort auf, dass die darin befindlichen Packages in zwei Bäume zerfallen. Der eine hat die Wurzel *groovy* und der andere die Wurzel *org.codehaus.groovy*.

Grob gesagt, handelt es sich bei den Klassen und Interfaces unter *groovy* um solche, die für Groovy-Programme in irgendeiner Weise direkt sichtbar sind, sei es als Basisklasse von in Groovy geschriebenen Klassen oder als Klassen, die von Groovy-Programmen verwendet werden. Häufig implementieren sie, auch wenn sie in Java geschrieben sind, das Interface `groovy.lang.GroovyObject`, können also wie Groovy-Objekte behandelt werden.

Unter *org.codehaus.groovy* dagegen sind Klassen und Interfaces zu finden, die für die interne Implementierung von Groovy benötigt werden, zum Beispiel für den Groovy-Compiler und für die dynamische Auflösung von Methoden-, Property- und Feldaufrufen. Auf diese Dinge werden Sie in aller Regel als Groovy-Programmierer nicht zugreifen – obwohl dem technisch nichts im Wege steht. Wenn Sie es aus irgendeinem Grund trotzdem tun möchten, müssen Sie allerdings damit rechnen, dass Ihre Programme nach einem Wechsel auf eine neue Groovy-Version nicht mehr funktionieren, da sich die internen Schnittstellen leicht einmal ändern können.[1]

Wir beschränken uns hier also auf die Packages unter *groovy*; Tabelle 6-1 führt die wesentlichen darin enthaltenen Packages mit ihrer jeweiligen Bedeutung auf. Außer den Packages `groovy.lang` und `groovy.util`, die in Groovy-Programmen standardmäßig verfügbar sind, müssen alle anderen Packages wie üblich importiert werden.

Tabelle 6-1: Wichtige Groovy-Standard-Packages

Package	Bedeutung
`groovy.inspect`	Enthält den `Inspector`, mit dem sich alle möglichen Eigenschaften eines Objekts abfragen lassen, sowie ein Skript zur grafischen Anzeige dieser Eigenschaften.
`groovy.lang`	Zentrales Package für alle Typen, die eine grundlegende Bedeutung in Groovy-Programmen haben, analog zum Package `java.lang` für Java. Die enthaltenen Klassen und Interfaces brauchen nicht importiert zu werden.
`groovy.mock`	Enthält eine Klasse für Mock-Objekte, die zum Testen von Groovy-Klassen verwendet werden können.
`groovy.model`	Verschiedene vorgefertigte Modelle, die bei der Programmierung von nach dem *model view controller*-Muster gebauten Anwendungsoberflächen angewendet werden können.

1 Korrekterweise wollen wir hinzufügen, dass sich auch die anderen Schnittstellen ändern können. Groovy ist eine junge Sprache, und da ist noch einiges in Bewegung. Sie können aber davon ausgehen, dass man bei Änderungen an den Klassen und Interfaces in den Packages unterhalb von *groovy* sehr viel vorsichtiger sein wird, um vorhandene Programme nicht mehr als nötig zu beeinträchtigen.

Tabelle 6-1: Wichtige Groovy-Standard-Packages (Fortsetzung)

Package	Bedeutung
`groovy.security`	Enthält eine Permission-Klasse, die für die Zuordnung von Rechten bei Skripten benötigt wird, die keine Codebase im üblichen Sinne besitzen.
`groovy.servlet`	Einige Klassen, mit deren Hilfe Groovy-Skripte und -Templates in Webanwendungen verwendet werden können.
`groovy.sql`	Eine API für den vereinfachten Zugriff auf Datenbanken über JDBC mit Groovy-typischen Mitteln.
`groovy.swing`	Enthält den `SwingBuilder`, mit dem Benutzeroberflächen in einem quasi-deklarativen Stil programmiert werden können.
`groovy.text`	Beherbergt das Template-Framework von Groovy sowie die mit Groovy gelieferten Template-Engines.
`groovy.time`	Einige Klassen für die Arbeit mit Zeiten und Zeitabständen. Derzeit wird noch an einer besseren Unterstützung für Datum und Zeit in Groovy gearbeitet, daher sollten diese Klassen eher nicht verwendet werden.
`groovy.ui`	Alle Klassen, die für den Start von Groovy als Shell oder als Interpreter benötigt werden.
`groovy.util`	Package mit diversen Hilfsklassen, die häufig und für unterschiedliche Zwecke verwendet werden. Die enthaltene Klassen und Interfaces brauchen nicht importiert zu werden.
`groovy.xml`	Klassen, die zur Unterstützung der Arbeit mit XML und HTML dienen.

Auf eine Reihe der in diesen Packages enthaltenen Klassen und Interfaces werden wir im Folgenden näher eingehen; eine Dokumentation der wichtigsten Typen ist außerdem in Anhang B zu finden.

Datenbanken

Kaum eine größere Anwendung kommt ohne eine Datenbank aus, und in aller Regel handelt es sich dabei um ein relationales Datenbank-Management-System (RDBMS), auf das über die standardisierte Abfragesprache SQL zugegriffen wird. Dementsprechend gut ausgebaut ist inzwischen die Unterstützung in Java für die Arbeit mit solchen Datenbanken; für praktisch alle relevanten Produkte gibt es Treiber, die über Java-Schnittstellen verfügen oder ganz in Java geschrieben sind. Für die Treiber gibt es einen Satz fest definierter Interfaces, die als JDBC (*Java Database Connectivity*) bezeichnet werden. Die JDBC-Schnittstellen können natürlich auch in Groovy genau so wie in Java verwendet werden. In den Groovy-Bibliotheken finden sich aber einige hilfreiche Klassen, in denen die Möglichkeiten, die diese Sprache bietet, fantasievoll umgesetzt sind.

Anhand einiger sehr einfacher Beispiele wollen wir Ihnen zeigen, welche prinzipiellen Ansätze dabei verfolgt werden. Die Umsetzung auf komplexere, realitätsnähere Situationen dürfte Ihnen aber nicht schwerfallen, sofern Sie sich mit Datenbankzugriffen unter Java generell etwas auskennen.

Darüber hinaus gibt es auch Persistenz-Ansätze, bei denen die Groovy-Objekte direkt auf Datenbanktabellen abgebildet werden, wie z.B. Hibernate oder JPA-Implementierungen. Seit Groovy 1.1 wird dieser Ansatz durch Annotationen unterstützt. Allerdings führte das innerhalb einer Groovy-Einführung zu weit.

Um mit einer Datenbank arbeiten zu können, müssen Sie natürlich eine solche zur Verfügung haben. Die folgenden sehr einfachen Beispiele sind ein wenig auf das gerade für Experimentierzwecke ausgesprochen populäre Datenbanksystem HSQLDB abgestimmt. Sie können natürlich auch ein anderes Datenbanksystem verwenden, müssen dann aber ein paar Anpassungen beim Herstellen der Datenbankverbindung und eventuell in den Anweisungen zum Erstellen der Tabellen vornehmen. Falls Sie es aber mit HSQLDB versuchen wollen, besorgen Sie sich einfach eine neuere Version von der Website unter *http://hsqldb.org/*. Packen Sie das Archiv auf Ihrem Rechner aus und kopieren Sie die das darin enthaltene Archiv *hsqldb.jar* in Ihr Groovy-Library-Verzeichnis *.groovy/lib* unterhalb Ihres Home-Verzeichnisses. Es wird dann beim nächsten Start von Groovy automatisch in den Klassenpfad eingebunden.

Eine Alternative ist die Open Source-Datenbank Derby, die bei Java 6 als JavaDB mit ausgeliefert wird und ähnlich einfach zu nutzen ist wie HSQLDB.

Die Klasse Sql

Während Sie es in Java beim Arbeiten mit JDBC immer mit einer `java.sql.Connection` zu tun haben, die eine Datenbankverbindung repräsentiert, verwenden Sie in Groovy lieber ein Objekt vom Typ `groovy.sql.SQL`; es kapselt die `Connection` und bietet eine Reihe von für den Programmierer erfreulichen zusätzlichen Möglichkeiten.

Datenbankverbindung herstellen

Wir wollen eine Datenbank mit einigen Grunddaten zu den Staaten der Erde verwenden. Erzeugen Sie dazu ein neues SQL-Objekt unter Angabe der Datenbank-URL, des Datenbankbenutzers und -passworts sowie der Treiberklasse.

```
def sql = groovy.sql.Sql.newInstance(
              'jdbc:hsqldb:file:db/staatendb',  // Datenbank-URL
              'sa',                             // Datenbankbenutzer
              '',                               // Passwort
              'org.hsqldb.jdbcDriver')          // Treiberklasse
```

Die URL besagt, dass wir mit einer HSQLDB-Datenbank in Dateiform arbeiten wollen. Die Datenbankdateien befinden sich unterhalb des aktuellen Arbeitsverzeichnisses in einem Unterverzeichnis *db* und haben alle den Namen *staatendb* mit verschiedenen Endungen.[2] Ist die Datenbank nicht vorhanden, legt HSQDB sie standardmäßig einfach an; wir brauchen also nichts weiter vorzubereiten.

Wenn wir das SQL-Objekt nicht mehr brauchen, schließen wir es wieder und beenden damit die Ressourcen fressende Datenbankverbindung.

```
sql.close()
```

Allgemeine SQL-Befehle

Das `Sql`-Objekt verfügt über eine ganze Reihe von Methoden, mit denen die unterschiedlichsten Datenbankoperationen ausgeführt werden. Die grundlegendste von ihnen ist wohl die Methode `execute()`, mit der Sie irgendeinen SQL-Befehl ausführen können, sofern dieser nicht zum Lesen von Datenbankdaten dient.

Mithilfe eines solchen SQL-Befehls wollen wir erst einmal eine Datenbank anlegen. Die Anweisung dazu ist lang, daher ist es von Vorteil, dass wir in Groovy auch mit mehrzeiligen Strings arbeiten können.

```
sql.execute '''
    DROP TABLE staat IF EXISTS;
    CREATE TABLE staat (
      id INT IDENTITY,
      name VARCHAR(80),
      hauptstadt VARCHAR(80),
      vorwahl SMALLINT,
      feiertag DATE)
'''
```

Die `execute()`-Methode erhält hier einen einzigen String mit zwei SQL-Befehlen. Der erste löscht die Tabelle, sofern sie schon existiert; das ist sehr praktisch zum Üben, alternativ könnte man auch die Dateien manuell löschen. Der zweite Befehl legt die Tabelle neu an und definiert die Felder für einen eindeutigen Primärschlüssel, den Namen des Landes, den Namen der Hauptstadt, die Telefonvorwahlnummer und ein Datum für den Nationalfeiertag oder Unabhängigkeitstag. Der Primärschlüssel wird von der Datenbank automatisch vergeben, darum brauchen wir uns also nicht zu kümmern.

Prepared Statements mit GStrings

In der gleichen Weise könnten wir die Tabelle nun auch mit Daten zu füllen. Das würde etwa so aussehen:

```
sql.execute '''INSERT INTO staat (name,hauptstadt,vorwahl,feiertag)
```

2 Anstelle der Factory-Methode `newInstance()` können Sie auch einen normalen Konstruktor verwenden; in diesem Fall müssen Sie aber eine schon geöffnete Datenbankverbindung oder eine JDBC-Datenquelle angeben.

```
            VALUES ('Afghanistan','Kabul',93,null)'''
sql.execute '''INSERT INTO staat (name,hauptstadt,vorwahl,feiertag)
       VALUES ('Ägypten','Kairo',20,'1922-02-28') '''
```

Das funktioniert, aber es ist umständlich und enthält sehr viel Redundanz. Außerdem muss jede Anweisung trotz ihrer Ähnlichkeit neu zerlegt werden, und es kann zu Fehlern kommen, wenn einer der einzufügenden Texte ein Hochkomma enthält. Bei der SQL-Programmierung arbeitet man daher lieber mit *Prepared Statements*, bei denen der Anweisungstext von den Daten getrennt übergeben und erst vom Datenbanktreiber zusammengesetzt wird. Prepared Statements sind außerdem performanter, wenn sie wieder verwendet werden, und sicherer, weil sie keinen Ansatzpunkt für SQL-Injection-Attacken bieten.

In Groovy geht das sehr elegant mit GStrings, zum Beispiel so:

```
def staatEinfügen(name,hauptstadt,vorwahl,feiertag) {
     sql.execute """INSERT INTO staat (name,hauptstadt,vorwahl,feiertag)
                    VALUES ($name,$hauptstadt,$vorwahl,$feiertag) """
}
```

Hier haben wir also die SQL-Anweisung in einem GString, der die Parameter der Methode referenziert. Damit keine Missverständnisse aufkommen: Die Parameter werden nicht einfach textuell in den Anweisungs-String eingefügt, das würde auch nicht funktionieren, denn die Parameter `name`, `hauptstadt` und `feiertag` müssten dann in Anführungszeichen gesetzt werden. Vielmehr analysiert das SQL-Objekt den GString, erzeugt daraus ein Prepared Statement und übergibt dies zusammen mit den Parametern an die JDBC-Verbindung. Nun wirkt das Einfügen der Daten in der Form von Methodenaufrufen schon viel aufgeräumter:

```
staatEinfügen('Albanien','Tirana',355,'1912-11-28')
staatEinfügen('Algerien','Algier',213,'1962-03-18')
```

Als Variante könnten Sie die Methode auch so definieren, dass einfach nur eine Map als Parameter dient.

```
def staatEinfügen (Map m) {
      sql.execute """INSERT INTO staat (name,hauptstadt,vorwahl,feiertag)
                    VALUES ($m.name,$m.hauptstadt,$m.vorwahl,$m.feiertag)"""
}
```

Dann müssen allerdings die Parameternamen beim Methodenaufruf angegeben werden.

```
staatEinfügen (name:'Andorra',
               hauptstadt:'Andorra la Vella',
               vorwahl:376,
               feiertag:'1278-09-08')
staatEinfügen (name:'Angola',
               hauptstadt:'Luanda',
               vorwahl:244,
               feiertag:'1975-11-11')
```

Durch Ergebnismengen iterieren

In den meisten Fällen sollen die in der Datenbank verstauten Informationen irgendwann auch wieder gelesen werden. Die von JDBC angebotenen Methoden dazu liefern Ergebnismengen zurück, die zwar eine Ähnlichkeit mit Listen haben, trotzdem aber in Java nur umständlich zu behandeln sind. Groovy bietet dazu einen eigenen Ansatz, bei dem die Ergebnismengen über Closures sehr einfach durchlaufen werden können und die Spaltenwerte sich wie Properties verhalten.

Sehen wir uns doch einmal an, was sich inzwischen in unserer Datenbank so angesammelt hat:

```
sql.eachRow('SELECT * FROM staat') {
      println "$it.name, $it.hauptstadt, $it.vorwahl, $it.feiertag" }
```

Wenn Sie schon einmal mit JDBC programmiert haben, werden Sie dies auch ausgesprochen übersichtlich finden. Wir übergeben der Methode `eachRow()` einfach die Select-Anweisung und einen Iterator; Letzterer wird dann mit jeder Zeile des Ergebnisses aufgerufen. Das Ergebnis sieht in diesem Fall so aus:

```
Afghanistan, Kabul, 93, null
Ägypten, Kairo, 20, 1922-02-28
Albanien, Tirana, 355, 1912-11-28
Algerien, Algier, 213, 1962-03-18
Andorra, Andorra la Vella, 376, 1278-09-08
Angola, Luanda, 244, 1975-11-11
```

Natürlich wollen Sie nicht immer alle Daten lesen. Wenn Auswahlkriterien angegeben werden sollen, ist es erneut praktisch, einen GString zu benutzen. In der folgenden Anweisung wollen wir die Staaten heraussuchen, die erst im 20. Jahrhundert selbstständig geworden sind.

```
def startDatum = new java.sql.Date(0,0,01)  // = 1.1.1900
sql.eachRow("SELECT name,feiertag FROM staat WHERE feiertag>=$startDatum") {
                  println "$it.name, $it.feiertag" }
```

Häufig steht von vornherein fest, dass ein Ergebnis aus nicht mehr als einem Datensatz bestehen kann, z.B. wenn man über einen Primärschlüssel sucht. Es ist dann nicht nötig, eine Ergebnismenge zu durchlaufen. Für diesen Zweck gibt es eine Methode `firstRow()`, die nur die erste Zeile der Ergebnismenge liefert, diese aber als Rückgabewert.

Wie heißt noch eben die Hauptstadt von Angola?

```
println sql.firstRow("SELECT * FROM staat WHERE name=?",['Angola'])?.hauptstadt
```

Richtig. Hier müssen wir übrigens mit einem Fragezeichen, dem klassischen SQL-Platzhalter, arbeiten, denn `firstRow()` unterstützt, wie auch einige weitere Methoden, keine GStrings. Dies ist nicht weiter tragisch, denn stattdessen können die Parameter als Liste angegeben werden. Es ist aber auch nicht schwierig, sich eine Kategorienklasse zu bauen, die eine `firstRow()`-Methode mit GString hinzufügt. Kategorieklassen bieten eine bequeme Möglichkeit, vorhandene Typen mit zusätzlicher Funktionalität zu versehen, auf die wir in Kapitel 7 noch näher eingehen werden.

```
class SqlCategory {
    static Object firstRow(groovy.sql.Sql self, GString gstring) {
        def params = self.getParameters(gstring);
        def sqlcode = self.asSql(gstring, params);
        self.firstRow(sqlcode, params);
    }
}
// Einzeilige Abfrage mit GString-Parameter
use (SqlCategory) {
    land = 'Angola'
    println sql.firstRow("SELECT * FROM staat WHERE name=$land")?.hauptstadt
}
```

Wenn übrigens die Ergebnismenge leer ist, liefert `firstRow()` eine Null als Ergebnis. Daher benötigen wir das Fragezeichen am Ende vor der Dereferenzierung von `hauptstadt`; andernfalls würden wir bei einem leeren Ergebnis eine `NullPointerException` erhalten.

Das SQL-Objekt bietet Ihnen noch zahlreiche weitere Methoden, die für das Arbeiten mit Datenbanken erforderlich sind. Dazu gehören `commit()` und `rollback()` für eine eigene Transaktionssteuerung. Näheres dazu im Anhang.

Datasets: Groovy statt SQL

Neben der `SQL`-Klasse hat Groovy in Bezug auf Datenbanken noch etwas anderes zu bieten, das geeignet ist, beim ersten Kontakt Überraschungen auszulösen, weil es die dynamischen Möglichkeiten von Groovy geschickt ausreizt: das sogenannte *Dataset*. Es repräsentiert eine bestimmte Datenmenge in der Datenbank (Tabelle oder View) und ermöglicht einen partiellen Zugriff auf diese Daten für relativ einfache Fälle ohne jeden SQL-Code und ohne Unterstützung durch ein objektrelationales Mapping-Tool. In J2EE ist dieser Ansatz als Data Transfer RowSet Pattern bekannt.

Sie erhalten ein Dataset, indem Sie es unter der Angabe eines Klassennamens und eines Tabellennamens bei Ihrem SQL-Objekt anfordern:

```
def staaten = sql.dataSet('Staat')
```

Als Ergebnis haben Sie ein Objekt der Klasse `groovy.sql.DataSet`, das Ihnen eine Sicht auf die Datenbanktabelle wie auf eine Liste von Zeilen verschafft. `Dataset` ist von der Klasse `SQL` abgeleitet; es verfügt also über dessen gesamte Funktionalität, erweitert diese aber um Datenbankzugriffe ohne SQL-Code.

Um deren Inhalt auszugeben, brauchen wir tatsächlich kein SQL mehr, dazu reicht einfach ein Aufruf der Methode `each()`.

```
staaten.each { println "$it.name, $it.hauptstadt, $it.vorwahl, $it.feiertag" }
```

Richtig beeindruckend wird es, wenn Sie die Daten in der Tabelle filtern möchten. Auch dies können Sie wie bei einer Liste mit einem Aufruf von `findAll()` erledigen. Die folgende Anweisung soll alle Staaten liefern, die in der ersten Hälfte des 20. Jahrhunderts unabhängig geworden sind.

```
def teilmenge = staaten.findAll {
    it.feiertag>='1900-01-01' && it.feiertag<'1950.01-01'
}
teilmenge.each { println "$it.name, $it.feiertag" }
```

In `teilmenge` haben Sie also wiederum ein `DataSet`-Objekt, dessen Inhalt Sie leicht auflisten lassen können. Das Ergebnis sieht erwartungsgemäß so aus:

```
Ägypten, 1922-02-28
Albanien, 1912-11-28
```

Wenn Sie jetzt die naheliegende Vermutung haben, dass dieses Dataset nichts weiter macht, als alle Daten einzulesen und mit der Closure zu filtern, und meinen, dass Sie das auch gekonnt hätten, liegen sie allerdings etwas daneben. Tatsächlich ist es so, dass die Closure nie ausgeführt wird, sondern dass deren zur Laufzeit verfügbare Syntaxstruktur dazu verwendet wird, daraus eine SQL-Anweisung zu bilden. Und diese wird beim `each()`-Aufruf dann angewendet. Als Beleg können Sie sich die SQL-Anweisung sowie die anstelle der Fragezeichen einzufüllenden SQL-Parameter einfach mal ansehen:

```
println teilmenge.sql
println teilmenge.parameters
```

In diesem Fall sehen SQL-Code und Parameter so aus:

```
select * from Staat where feiertag >= ? and feiertag < ?
[1900-01-01,1950-01-01]
```

Natürlich steht Ihnen innerhalb der Closure nicht die ganze Breite der sprachlichen Möglichkeiten von Groovy zur Verfügung. Damit sich der Groovy-Code überhaupt in SQL-Code übersetzen lässt, ist darin nur ein Ausdruck zulässig, der aus einfachen Vergleichen und logischen Verknüpfungen besteht.

Das Dataset ermöglicht Ihnen auch, Veränderungen an der darunter liegenden Tabelle vorzunehmen; derzeit können aber nur Datensätze hinzugefügt werden.

```
staaten.add (name:'Antigua und Barbuda',
             hauptstadt:"Saint John's",
             vorwahl:1268,
             feiertag:'1981-11-01')
```

Das ähnelt dem Aufruf unserer weiter oben selbst gebauten Methode zum Hinzufügen von Datensätzen. Allerdings brauchen wir diese Methode gar nicht zu programmieren, denn das SQL-Statement zum Einfügen von Datensätzen wird im Dataset einfach automatisch aus den in der Form von benannten Parametern übergebenen Feldnamen und Feldinhalten generiert. Eine einfache Abfrage mittels `sql.eachRow()`, wie oben gezeigt, beweist Ihnen, dass der zusätzliche Satz auch tatsächlich sofort in die Datenbank geschrieben worden ist.

Leider sind im Dataset die übrigen Datenbankoperationen wie `update()` zum Zurückschreiben geänderter Datensätze und `delete()` zum Löschen von Datensätzen noch nicht implementiert, obwohl dies eigentlich nicht schwieriger sein sollte als `add()`. Auch macht

es im Vergleich zum generellen Zustand von Groovy einen noch etwas labilen Eindruck. Allerdings ist es ein beeindruckendes Beispiel dafür, was mit einer dynamischen Programmiersprache wie Groovy möglich ist; und innerhalb der nächsten Updates dürfte auch mit einer Vervollständigung und Perfektionierung dieser Klasse zu rechnen sein.

XML

Wenn man bedenkt, dass die Auszeichnungssprache XML kaum zehn Jahre alt ist, kommt es einem doch erstaunlich vor, wie allgegenwärtig sie heutzutage in der Informationstechnik ist, und das völlig unabhängig von den verwendeten Programmiersprachen und Systemplattformen. Da man also in der Programmierung ständig mit XML zu tun hat, ist es auch wichtig, dafür effektive Hilfsmittel zur Hand zu haben. Leider sind die in Java integrierten APIs nach dem W3C-Standard alles andere als angenehm zu handhaben. Groovy beinhaltet daher eine ganze Reihe von Werkzeugen, die den Umgang mit XML erleichtern. Wir wollen uns hier auf diejenigen beschränken, bei denen die dynamischen Möglichkeiten von Groovy am effektivsten eingesetzt werden. Weitere Hinweise dazu finden Sie in Anhang B unter dem Package `groovy.xml`.

XML-Dokumente als Objektstruktur

XML-Dokumente führen bei der Programmierung oft ein Doppelleben: Sie existieren einerseits in der Form einer Textdatei, in der die Auszeichnungen mittels vieler spitzer Klammern eingefügt sind. Andererseits gibt es sie aber auch innerhalb eines Programms als interne Datenstruktur, die die Struktur der verschachtelten Auszeichnungen im Dokument widerspiegelt. Standardmäßig verwendet man in Java dazu das `org.w3c.dom.Document`, das natürlich auch mit Groovy verwendet werden kann und sogar durch eine Kategorienklasse namens `groovy.xml.dom.DOMCategory` handlicher wird. Wir wenden uns hier aber einer anderen Option zu.

Um ein Beispiel zu haben, bleiben wir gleich bei der Geografie und legen eine XML-Datei mit Staaten an. Um der hierarchischen Struktur von XML-Dokumenten Rechnung zu tragen, gruppieren wir sie jetzt aber nach Kontinenten und tragen zu jedem Staat noch ein paar wichtige Städte mit ihren Einwohnerzahlen ein.[3] Wir speichern die Daten einer Datei namens *staaten.xml* im aktuellen Verzeichnis, wie es in Beispiel 6-1 zu sehen ist.

Beispiel 6-1: XML-Datei

```
<? xml version ="1.0" ?>
<staaten>
    <kontinent bez="Afrika">
    <staat bez="Ägypten" vorwahl="20" feiertag="1922-02-28">
```

3 Alle Angaben stammen aus Wikipedia. Wenn Sie Fehler finden, korrigieren Sie diese am besten gleich an Ort und Stelle unter *http://de.wikipedia.org/*.

Beispiel 6-1: XML-Datei (Fortsetzung)

```
        <stadt bez="Kairo" einwohner="7734614"/>
        <stadt bez="Alexandria" einwohner="3811516"/>
        <stadt bez="Gizeh" einwohner="2443203"/>
        <stadt bez="Schubra al-Chaima" einwohner="991801"/>
    </staat>
    <staat bez="Algerien" vorwahl="213" feiertag="1962-03-18">
        <stadt bez="Algier" einwohner="1518083"/>
        <stadt bez="Oran" einwohner="771066"/>
    </staat>
    <staat bez="Angola" vorwahl="244" feiertag="1975-11-11">
        <stadt bez="Luanda" einwohner="2776125"/>
        <stadt bez="Huambo" einwohner="226177"/>
        <stadt bez="Lobito" einwohner="207957"/>
    </staat>
    </kontinent>
    <kontinent bez="Asien">
    <staat bez="Afghanistan" vorwahl="93">
        <stadt bez="Kabul" einwohner="2536300"/>
        <stadt bez="Herat" einwohner="349000"/>
        <stadt bez="Kandahar" einwohner="324800"/>
        <stadt bez="Masar-e Scharif" einwohner="300600"/>
    </staat>
    </kontinent>
    <kontinent bez="Europa">
    <staat bez="Albanien" vorwahl="355" feiertag="1912-11-28">
        <stadt bez="Tirana" einwohner="347801"/>
        <stadt bez="Durrës" einwohner="122034"/>
    </staat>
    <staat bez="Andorra" vorwahl="376" feiertag="1278-09-08">
        <stadt bez="Andorra la Vella" einwohner="22884"/>
    </staat>
    </kontinent>
    <kontinent bez="Amerika">
    <staat bez="Antigua und Barbuda" vorwahl="1268" feiertag="1981-11-01">
        <stadt bez="Saint John's" einwohner="25150"/>
    </staat>
    </kontinent>
</staaten>
```

Um diese Daten nun im Programm nutzbar zu machen, gibt es verschiedene Möglichkeiten. Sie unterscheiden sich nach dem Ergebnis des Parse-Prozesses und nach der Strategie.

XmlParser

Eine sehr einfache Möglichkeit besteht darin, den XML-Parser von Groovy einzusetzen. Es kostet Sie nicht mehr als zwei Zeilen:

```
def parser = new XmlParser()
def staaten = parser.parse(new File('staaten.xml'))
```

Die Klasse `XmlParser` befindet sich im Package `groovy.util`, daher brauchen Sie keinen Pfad anzugeben. Das Ergebnis des `parse()`-Aufrufs ist eine Baumstruktur aus Knotenobjekten. Das heißt, die Variable `staaten` verweist auf ein Objekt der Klasse `groovy.util.Node`, an dem alle weiteren Elemente des Baums »hängen«.

Es handelt sich hier um die gleiche Art von Struktur, die auch programmintern mit einem `NodeBuilder` erzeugt werden kann (siehe Kapitel 3).

Prüfen wir in einer kurzen Schleife, ob alle Staaten aus der XML-Datei angekommen sind.

```
println staaten.kontinent.each { kont ->
    println kont.'@bez'
    kont.staat.each {staat ->
        println "  ${staat.'@bez'} ${staat.'@feiertag'}"
    }
}
```

Das Ergebnis gibt uns recht:

```
Afrika
  Ägypten 1922-02-28
  Algerien 1962-03-18
  Angola 1975-11-11
Asien
  Afghanistan null
Europa
  Albanien 1912-11-28
  Andorra 1278-09-08
Amerika
  Antigua und Barbuda 1981-11-01
```

Zur Erinnerung: Der Property-Zugriff bei `Node`-Objekten liefert immer eine Liste aller untergeordneten Elemente mit dem angegebenen Namen. Um ein Attribut abzufragen, muss dem Attributnamen ein `@`-Zeichen vorangestellt werden, und beides muss in Anführungszeichen gesetzt werden. Wir brauchen an dieser Stelle nicht tiefer in diese Art von Strukturen einzugehen; lesen Sie bei Bedarf einfach noch einmal in Kapitel 3 nach.

XmlNodePrinter

Wenn Sie eine solche Baumstruktur im Programm haben und als XML-Dokument abspeichern möchten, ist dies auch nicht schwierig.

```
np = new XmlNodePrinter()
np.print(staaten)
```

Der `groovy.util.XmlNodePrinter` funktioniert analog zu dem aus Kapitel 4 bekannten `NodePrinter`. Sie übergeben seiner `print()`-Methode einen Wurzelknoten, und er gibt die gesamte Struktur ordentlich formatiert auf der Konsole aus. Wenn Sie den `XmlNodePrinter` mit einem `PrintWriter` als Argument instantiieren, wird das Ergebnis in diesen hineingeschrieben.

```
new File('staaten2.xml').withPrintWriter { writer ->
    new XmlNodePrinter(writer).print(staaten)
}
```

Und schon haben Sie die Daten in einer neuen XML-Datei *staaten2.xml*. Einschränkend ist zu erwähnen, dass der `XmlNodePrinter` keine Kopfzeilen in die XML-Datei schreibt und demzufolge schlecht mit Zeichen umgehen kann, die nicht 7-Bit-ASCII sind, denn dafür bräuchte man ja die Kopfzeile mit der Zeichensatzdeklaration. Wenn man ganz ordnungsgemäße Ausgabedateien benötigt, muss man derzeit wohl doch noch auf die standardmäßigen Mittel der Java-APIs zurückgreifen.

XML on the Fly verarbeiten

Alle Daten eines XML-Dokuments in einer internen Struktur zu halten ist nicht in jedem Fall sinnvoll, insbesondere wenn die Daten sehr umfangreich sind und nur ein kleiner Teil von ihnen von Interesse ist oder eine sequenzielle Verarbeitung möglich ist.

Zum sequenziellen Einlesen von XML-Daten ist der ereignisorientierte SAX-Parser geeignet, der Bestandteil der Java-Bibliotheken ist und dem Groovy nicht viel hinzufügen kann. Für das programmgesteuerte Erzeugen von XML-Dokumenten kann jedoch hervorragend auf das Konzept der Groovy-Builder zurückgegriffen werden.

Der MarkupBuilder

Die Klasse `groovy.xml.MarkupBuilder` ermöglicht es, XML-Dokumente in der semi-deklaratorischen Manier der Groovy-Builder zusammenzusetzen, wie wir sie in Kapitel 4 behandelt haben. Wenn Sie ein Dokument erzeugen möchten, das die Informationen unseres obigen Beispiels *staaten.xml* enthält, können Sie so vorgehen:

```
new groovy.xml.MarkupBuilder().staaten {
    kontinent(bez:'Afrika') {
         staat (bez:'Ägypten',vorwahl:20,feiertag:'1922-02-28') {
                  stadt(bez:'Kairo',einwohner:'7734614')
                  stadt (bez:'Alexandria',einwohner:'3811516')
                  stadt (bez:'Gizeh',einwohner:'2443203')
                  stadt (bez:'Schubra al-Chaima',einwohner:'991801')
            }
            staat (bez:'Algerien',vorwahl:'213',feiertag:'1962-03-18') {
                  stadt (bez:'Algier',einwohner:'1518083')
                  stadt (bez:'Oran',einwohner:'771066')
            }
// Und so weiter ...
    }
}
```

Wieder erscheinen Kontinente, Länder und Städte auf dem Bildschirm. Auch dem `MarkupBuilder` können Sie einen Writer mitgeben (ein `PrintWriter` ist nicht erforderlich), so dass Sie die Daten beispielsweise in eine Datei schreiben können.

```
new File('staaten3.xml').withWriter { writer ->
    new groovy.xml.MarkupBuilder(writer).staaten {
    ...
    }
}
```

Sie können an dem Beispiel leicht erkennen, dass der `MarkupBuilder` einfach Methodenaufrufe durch Elemente gleichen Namens ersetzt und aus den benannten Parametern analoge Elementattribute macht.

Eine interessante Anwendung dieser Vorgehensweise ist die Generierung von XML-Testdaten für JUnit-Tests.

Zwei spezielle Situationen wollen wir noch erwähnen, bei denen die Lösung vielleicht nicht so offensichtlich ist.

Mixed Content

Zum einen gibt es in XML auch reine Textelemente wie etwa der folgende kurze Absatz aus einem DocBook-Dokument:

```
<para>Dies ist ein kurzer Absatz.</para>
```

So etwas können Sie mit dem `MarkupBuilder` darstellen, indem Sie einfach den Text als String der Methode für das Tag übergeben:

```
para "Dies ist ein kurzer Absatz."
```

Schwieriger wird es, wenn eine Mischung aus Texten und Tags auftritt, wie etwas hier:

```
<para>Dies ist ein <emphasis>kurzer</emphasis> Absatz.</para>
```

Da kommen Sie gegenwärtig mit dem einfachen `MarkupBuilder` nicht weiter. Stattdessen bietet es sich an, auf den `StreamingMarkupBuilder` umzusteigen, auf den wir etwas weiter unten zu sprechen kommen.

Spezielle Zeichen in Namen

Zum anderen sind in XML Zeichen in Element- und Attributnamen erlaubt, die in Groovy-Namen nicht vorkommen können, z.B. Bindestriche, Punkte und Doppelpunkte. Zum Glück ist es aber in Groovy immer möglich, Member-Namen als Strings anzugeben, und die dürfen beliebige Zeichen enthalten. Das folgende Beispiel gibt einen Ausschnitt aus einer XSLT-Datei für DocBook-Dokumente aus:

```
builder = new groovy.xml.MarkupBuilder()
builder.'xsl:template' (match:'para') {
    'fo:block' ('xsl:user-attribute-sets':'normal-text') {
     'xsl:apply-templates'()
     }
}
```

Wenn Sie das Skript laufen lassen, erscheint ganz korrekt diese Ausgabe:

```
<xsl:template match='para'>
  <fo:block xsl:user-attribute-sets='normal-text'>
    <xsl:apply-templates />
  </fo:block>
</xsl:template>
```

Der StreamingMarkupBuilder

Neben dem `MarkupBuilder`, den wir eben kennengelernt haben, gibt es eine zweite Klasse, mit der XML-Dokumente in ähnlicher Weise zusammengebaut werden können: der `StreamingMarkupBuilder`. Er ist in erster Linie dafür vorgesehen, XML-Code für die Verarbeitung durch Maschinen zu produzieren, und verfügt über einige zusätzliche Möglichkeiten. So gelingt es uns mit dem `StreamingMarkupBuilder` auch, XML-Code zu produzieren, in dem Text und Elemente gemischt sind. Das Beispiel aus dem obigen Abschnitt »Mixed Content« etwa lässt sich so darstellen:

```
builder = new groovy.xml.StreamingMarkupBuilder()
writableClosure = builder.bind {
    para {
     out << 'Dies ist ein '
     emphasis 'kurzer'
     out << 'Absatz.'
    }
}
println writableClosure
```

Wie Sie sehen, ist das Vorgehen hier etwas anders: Sie rufen eine Methode `bind()` auf, der die XML-Struktur in der Form einer Closure übergeben werden muss. Das Ergebnis ist wiederum eine Closure. Anders als beim `MarkupBuilder` wird nämlich der XML-Code nicht sofort produziert und in einen Writer geschrieben, sondern es entsteht ein Closure-Objekt, das das Interface `Writable` implementiert (siehe Kapitel 5). Erst wenn dieses Objekt herausgeschrieben wird (wie hier in der `println()`-Methode), entsteht der eigentliche XML-Text. Das Ergebnis sieht so aus:

```
<para>Dies ist ein <emphasis>kurzer</emphasis>Absatz.</para>
```

Dieses Resultat sieht nicht so schön strukturiert aus wie beim einfachen `MarkupBuilder`. Dies liegt insofern nahe, als das Ergebnis von einer Maschine gelesen werden soll, die sich für ordentliches Einrücken nicht interessiert und die zusätzliche Leer- und Tabulatorzeichen möglicherweise falsch verarbeitet.

Wie oben zu sehen ist, ermöglicht das Mischen von Text und Elementen eine spezielle Operation mit der Property `out` und dem `<<`-Operator. Es gibt noch einige weitere derartige Operationen, die es insgesamt ermöglichen, eine komplette XML-Datei zu generieren, siehe Tabelle 6-2.

Tabelle 6-2: Spezielle Operationen im StreamingMarkupBuilder

Beispiel	Bedeutung
`comment << "Kommentartext"`	Fügt einen Text als XML-Kommentar ein.
`namespaces << [Präfix:"URI"]`	Definiert XML-Namensräume. In der übergebenen Map werden die Schlüssel als Präfixe und die Werte als zugehörige URIs betrachtet.
`out << "Text"`	Fügt den angegebenen Text mit spezieller Behandlung von XML-Sonderzeichen ein.
`unescaped << "Text"`	Fügt den angegebenen Text ohne spezielle Behandlung von XML-Sonderzeichen ein.
`pi << [PItarget:"Instruction"]`	Fügt eine oder mehrere XML-Verarbeitungsanweisungen (*Processing Instruction*) in die Ausgabe ein.
`mkp.xmlDeclaration()`	Fügt eine vollständige XML-Deklaration als Kopfzeile ein.

Die letzte Zeile der Tabelle 6-2 ist noch erklärungsbedürftig. Im `StreamingMarkupBuilder` können Namensraum-Präfixe wie Properties verwendet werden (d.h. `pre.element()` wird beispielsweise in `<pre:element/>` umgesetzt). Dabei ist `mkp` ein vordefinierter Namensraum mit einer speziellen Funktionalität: So führt der Aufruf `mkp.xmlDeclaration()` nicht dazu, dass das Tag `<mkp:xmlDeclaration/>` eingefügt wird, sondern dass dem XML-Dokument eine XML-Deklarationszeile (z.B. `<?xml version="1.0" encoding="UTF-8"?>`) vorangestellt wird.

In der Groovy-Distribution gibt es eine in Groovy geschriebene Klasse namens `org.codehaus.groovy.tools.DocGenerator`, die eine HTML-Seite mit einer Übersicht aller vordefinierten Methoden generiert und als Beispiel für die Verwendung des `StreamingMarkupBuilder` dienen kann.

Weitere Möglichkeiten

Was man mit Groovy und XML noch alles anfangen kann, übersteigt bei Weitem den Umfang dieses Buchs. Aber zumindest wollen wir als Anregung darauf hinweisen, was es da noch so gibt.

- Der `XmlSlurper` ist eine Alternative zum `XmlParser`. Er baut keine Struktur aus `Node`-Objekten auf, sondern liefert als Ergebnis ein `GPathResult`-Objekt, dessen Daten zwar effizienter ausgelesen werden können, das aber nur lesenden Zugriff erlaubt. Er lässt sich gut mit dem `StreamingMarkupBuilder` zur durchlaufenden Verarbeitung von XML-Datenströmen kombinieren.
- Die Kategorienklasse `DOMCategory` erleichtert etwas die Arbeit mit klassischen DOM-Objektbäumen.
- Der `SAXBuilder` ist eine Builder-Klasse, die XML-Dokumente derart erzeugt, dass sie SAX-Ereignisse auslösen. Ergänzend gibt es auch einen `StreamingSAXBuilder`.
- `DOMBuilder` ist ein Builder, mit dem man DOM-Strukturen erzeugen kann. Auch hierzu gibt es einen `StreamingDOMBuilder`.

Weitere Informationen können der Groovy-Website entnommen werden, die zum Thema XML recht ausführliche, wenn auch englischsprachige, Informationen bietet.

Webanwendungen

Zu den am häufigsten vorkommenden Anwendungsfällen der Java-basierten Technologien gehören Anwendungen, auf die man mit dem Webbrowser zugreift. Die unübersehbare Vielfalt von Web-Entwicklungsplattformen, die es für Java gibt, unterstreicht dies. Eine von ihnen, Grails, basiert auf Groovy und macht sich insbesondere dessen dynamische Eigenschaften zunutze, um eine extrem effiziente Basis für Webentwicklungen zu bieten.

Für einfache Webanwendungen brauchen Sie aber nicht unbedingt auf Grails zurückzugreifen, denn auch Groovy allein hat schon einiges zu bieten, wenn auch auf einer eher elementaren Ebene. Sie haben hier eine interessante Alternative zur üblichen Java-Servlet-Programmierung, bei der die verschiedenen speziellen Stärken dieser Programmiersprache gut zum Tragen kommen.

Webseiten generieren

Wenn es einfach nur darum geht, HTML-Seiten programmgesteuert zusammenzustellen, hat Groovy zwei Möglichkeiten zur Verfügung, die Sie bereits in anderen Zusammenhängen kennengelernt haben.

Mit dem MarkupBuilder

Die Klasse `groovy.util.MarkupBuilder` haben wir schon im Zusammenhang mit dem Generieren von XML-Dokumenten vorgestellt. Dieselbe Klasse eignet sich natürlich auch für den Aufbau von Webseiten, denn den `MarkupBuilder` interessiert in keiner Weise, was für Tags er in seinen Writer schreibt. Die einzige Schwierigkeit könnte darin bestehen, dass er immer wohlgeformtes XML erzeugt, bei dem alle Tags geschlossen werden. Dies kann bei älteren Browsern zu Problemen führen, die beispielsweise bei einem Tag wie `<br/>` den Schrägstrich nicht verdauen können. Zur Sicherheit sollte den mit dem `MarkupBuilder` erzeugten Dokumenten auch immer ein XHTML-Header vorangestellt werden, der deutlich macht, dass hier wohlgeformtes XML folgt.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="de" xml:lang="de">
```

Freilich muss man sich dann auch an die XHTML-Syntax halten, die in einigen Punkten weniger liberal als das normale HTML ist.

Davon einmal abgesehen, ist das Generieren einer HTML-Seite mit dem `MarkupBuilder` recht übersichtlich:

```
def builder = new groovy.xml.MarkupBuilder()
builder.html (xmlns:"http://www.w3.org/1999/xhtml",lang:"de",'xml:lang':"de") {
    head {
        title "Hello-World"
    }
    body {
        t1 "Die Hello-World-Seite"
    }
    p "Die aktuelle Zeit ist: ${new Date()}"
}
```

Wenn wir den `MarkupBuilder` nicht mit einem Reader instantiieren, schreibt er seine Ausgaben einfach in die Konsole, und dort erscheint Folgendes:

```
<html xml:lang='de' xmlns='http://www.w3.org/1999/xhtml' lang='de'>
  <head>
    <title>Hello-World</title>
  </head>
  <body>
    <t1>Die Hello-World-Seite</t1>
  </body>
  <p>Die aktuelle Zeit ist: Mon May 28 15:45:04 CEST 2007</p>
</html>
```

Sie sehen: Alle Tags sitzen schon einmal richtig. Wirklich interessant wird es aber erst, wenn Sie HTML und Daten mischen. Nehmen wir noch einmal die Datenbank mit den Staaten aus dem ersten Abschnitt dieses Kapitels, die wir so abgefragt haben:

```
sql.eachRow('SELECT * FROM staat') {
      println "$it.name, $it.hauptstadt, $it.vorwahl, $it.feiertag" }
```

Mit nicht allzu viel Mühe können wir daraus auch eine Webseite zaubern. Allerdings nutzen wir jetzt die `rows()`-Methode des SQL-Objekts, mit dem wir eine Liste von Ergebnisdaten erhalten, damit wir nicht innerhalb des Builder-Codes eine Closure aufmachen müssen.

```
// Daten einlesen (das SQL-Objekt ist bereits offen)
staaten = sql.rows('SELECT * FROM staat')
// Webseite generieren
new File('staaten.html').withPrintWriter { writer ->
  writer.println '''<?xml version="1.0" encoding="iso-8859-1" ?>
    <!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">'''
  builder = new groovy.xml.MarkupBuilder(writer)
  builder.html (xmlns:"http://www.w3.org/1999/xhtml",lang:"de",'xml:lang':"de") {
    head {
      title "Staaten"
    }
    body {
      h1 "Staaten der Erde"
      table (border:'1') {
```

```
            tr {
               th "Land"
               th "Hauptstadt"
               th "Vorwahl"
               th "Feiertag"
            }
            for (staat in staaten) {
              tr {
                 td staat.name
                 td staat.hauptstadt
                 td staat.vorwahl
                 td (staat.feiertag?staat.feiertag:'-')
} } } } } }
```

Das Ergebnis ist nicht unbedingt preiswürdig, aber korrekter HTML-Code, wie Abbildung 6-1 zeigt.

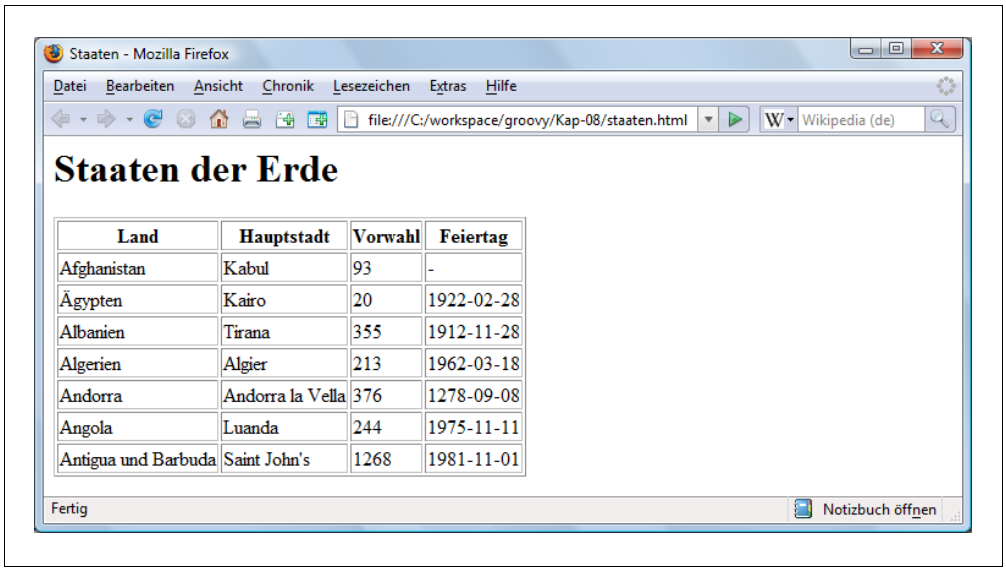


Abbildung 6-1: Ergebnis des MarkupBuilder-Einsatzes

Mit der Template-Engine

Auch die Template-Engine von Groovy lässt sich hervorragend zum Generieren von Webseiten verwenden. Sie eignet sich insbesondere dann, wenn die Gestaltung vom Inhalt getrennt sein soll. In Kapitel 4 haben Sie bereits gesehen, wie Templates prinzipiell in Groovy funktionieren. Nun wollen wir sehen, wie man mit ihrer Hilfe das gleiche Ergebnis wie mit dem `MarkupBuilder` erzielen kann.

Die Textvorlage, die wir benötigen, sieht folgendermaßen aus. Wir speichern sie unter dem Namen *staaten-template.html* ab.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xml:lang='de' xmlns='http://www.w3.org/1999/xhtml' lang='de'>
  <head>
    <title>Staaten</title>
  </head>
  <body>
    <h1>Staaten der Erde</h1>
    <table border='1'>
      <tr>
        <th>Land</th>
        <th>Hauptstadt</th>
        <th>Vorwahl</th>
        <th>Feiertag</th>
      </tr>
<% for (staat in staaten) { %>
      <tr>
        <td>$staat.name</td>
        <td>$staat.hauptstadt</td>
        <td>$staat.vorwahl</td>
        <td>${staat.feiertag?staat.feiertag:'-'}</td>
      </tr>
<% } // Ende der For-Schleife %>
    </table>
  </body>
</html>
```

Der interessante Bereich liegt in der `for`-Schleife, die am Anfang und am Ende durch die Tags `<%...%>` gekennzeichnet ist. Dazwischen durchläuft das Template eine Liste namens `staaten` und legt für jede Datenbankzeile eine HTML-Tabellenzeile an.

Nun müssen wir nur noch dafür sorgen, dass die Vorlage mit den Daten zusammengemischt wird. Das geschieht in dem folgenden Skriptabschnitt. Auch hier gehen wir wieder davon aus, dass das `SQL`-Objekt für die Datenbankverbindung bereits vorhanden ist.

```
// Template-Engine instantiieren
engine = new groovy.text.SimpleTemplateEngine()
// Template aus Textvorlage erzeugen
template = engine.createTemplate(new File('staaten-template.html'))
// Binding für die Daten anlegen
daten = [staaten:sql.rows('SELECT * FROM staat')]
// Template und Daten mischen
ergebnis = template.make(daten)
// Ergebnis in Datei schreiben
new File('staaten2.html').withWriter { writer ->
    writer.write(ergebnis)
}
```

Die frisch generierte HTML-Seite *staaten2.html* unterscheidet sich in keiner Weise von der mit dem `MarkupBuilder` erzeugten Seite, wie sie in Abbildung 6-1 weiter oben dargestellt ist.

Groovy-Servlets

Bis hierhin haben wir nur davon gesprochen, wie man mit Groovy HTML-Seiten erzeugen kann. Das ist ein wenig wie Trockenschwimmen, denn die Seiten sollen in Wirklichkeit natürlich nicht mithilfe von Skripten auf dem PC entstehen, sondern in einem Webserver produziert werden. Nun steht ja mit dem Servlet-Container ein bewährter Standard für den Betrieb von Webservern unter Java zur Verfügung. Und da Groovy auf Java basiert, kann man diese natürlich nutzen.

Vorbereitungen

Um die anschließenden Experimente verfolgen zu können, benötigen Sie irgendeinen Java-basierten Webserver, der standardkonform mit Servlets umgehen kann. Es gibt eine ganze Reihe von solchen Programmen, die frei verfügbar sind, sehr populär ist beispielsweise der Apache-Tomcat-Server. Ebenfalls sehr beliebt ist ein Server namens Jetty. Er ist nicht ganz so schwergewichtig wie Tomcat und reicht für unsere Zwecke völlig aus.

Wenn Sie also noch keinen Servlet-fähigen Webserver haben, laden Sie sich einfach die neueste Jetty-Version von *http://jetty.mortbay.org/jetty/index.html* herunter und entpacken sie in ein beliebiges Verzeichnis. Unterhalb dieses Verzeichnisses finden Sie dann unter anderem ein Verzeichnis namens *webapps*, in dem die Webanwendungen installiert werden. Wenn Sie einen anderen Servlet-Container haben, wird es sicher auch ein vergleichbares Verzeichnis geben. Führen Sie nun folgende Schritte durch.

1. Legen Sie unterhalb von *webapps* ein neues Verzeichnis namens *groovy* an. In diesem Verzeichnis, oder in darunter liegenden Verzeichnissen, werden Sie gleich Skripte und Templates ablegen können, die von Groovy verarbeitet werden sollen.
2. Legen Sie unterhalb von *groovy* ein Verzeichnis *WEB-INF* an. Dieses Verzeichnis enthält interne Informationen und wird vom Webserver dargestellt.
3. Legen Sie unterhalb von *WEB-INF* ein weiteres Verzeichnis *lib* an. Dieses ist für Java-Bibliotheken vorgesehen.
4. Legen Sie in *lib* eine Kopie der Datei *groovy-all-1.1.jar* (oder eine andere Version) aus Ihrer Groovy-Distribution ab.
5. Legen Sie in *WEB-INF* eine einfache Textdatei namens *web.xml* mit dem in Beispiel 6-2 gezeigten Inhalt an. Dies ist der Deployment-Descriptor, der dem Webserver mitteilt, wie mit den Groovy-Servlets umzugehen ist.

Beispiel 6-2: Inhalt der Datei web.xml

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd" >
<web-app>
    <servlet>
    <servlet-name>GroovyServlet</servlet-name>
```

Beispiel 6-2: Inhalt der Datei web.xml (Fortsetzung)

```
    <servlet-class>groovy.servlet.GroovyServlet</servlet-class>
    </servlet>
    <servlet>
      <servlet-name>TemplateServlet</servlet-name>
      <servlet-class>groovy.servlet.TemplateServlet</servlet-class>
    </servlet>
    <servlet-mapping>
    <servlet-name>GroovyServlet</servlet-name>
    <url-pattern>*.groovy</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
    <servlet-name>TemplateServlet</servlet-name>
    <url-pattern>*.html</url-pattern>
    </servlet-mapping>
</web-app>
```

Grob gesagt, enthält die Datei *web.xml* die Information, dass alle Dateien mit der Endung *.groovy* von dem Servlet `groovy.servlet.GroovyServlet` und alle Dateien mit der Endung *.html* von dem Servlet `groovy.servlet.TemplateServlet` behandelt werden sollen. Was das konkret bedeutet, werden Sie in den folgenden Abschnitten gleich sehen.

Prüfen Sie aber vorher noch einmal, ob die Struktur Ihres Verzeichnisses so aussieht wie hier dargestellt:

- *webapps* – Wurzelverzeichnis für alle Webanwendungen
  - *groovy* – Verzeichnis für Ihre Versuche mit den Groovy-Servlets
    - *WEB-INF* – internes Verzeichnis der Webanwendung
      - *lib* – Verzeichnis für Java-Bibliotheken
        - *groovy-all…jar* – Groovy-Bibliothek
      - *web.xml* – Deployment-Deskriptor

Starten Sie nun den Webserver – bei Jetty geschieht dies mit einem Skript namens *jetty* oder *jetty.bat* im Wurzelverzeichnis des Webservers – oder starten Sie ihn neu, sollte er bereits laufen. Legen Sie eben noch folgende rudimentäre XML-Datei unter dem Namen *index.html* in das Verzeichnis *groovy*, damit wir ausprobieren können, ob alles funktioniert.

```
<html><body>Es funktioniert. ${new Date()}</body></html>
```

Wenn Sie jetzt Ihren Browser auf die Adresse *http://localhost:8080/groovy* richten (eventuell müssen Sie auch eine andere oder gar keine Portnummer angeben, im Allgemeinen meldet der Webserver beim Start, auf welchem Port er arbeitet), sollte genau dieser Text im Browserfenster erscheinen: »Es funktioniert« sowie die aktuelle Uhrzeit – so wie es Abbildung 6-2 zeigt.

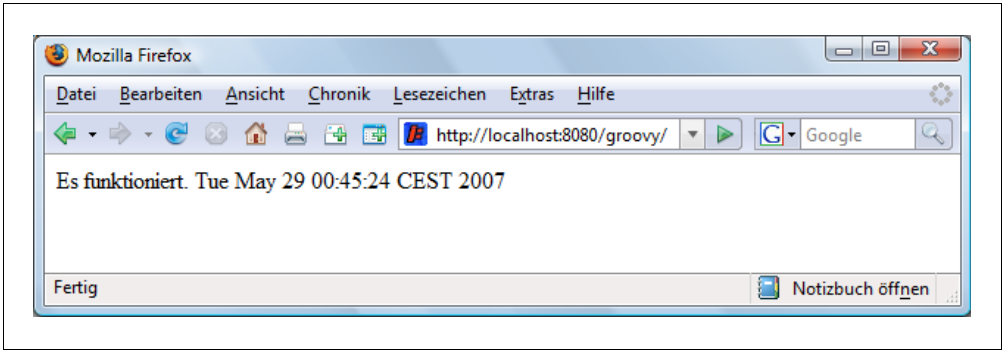


Abbildung 6-2: Webserver mit Groovy erfolgreich gestartet

Groovlets und das GroovyServlet

Der Rest ist fast ein Kinderspiel. Sie können nun ein normales Groovy-Skript bauen, es in das Wurzelverzeichnis Ihrer Webanwendung legen, und es kann unter diesem Namen über das Netzwerk ausgeführt werden; alle Programmausgaben werden automatisch an den Browser geleitet. Achten Sie nur darauf, dass der Dateiname auf *.groovy* endet. Wenn Sie ein neues Skript hinzufügen, eines verändern oder eines löschen, brauchen Sie den Server nicht neu zu starten (oder ein *hot deployment* einzuleiten), das `GroovyServlet` bemerkt es auch so und reagiert sofort.

Solche Skripte, die vom `GroovyServlet` ausgeführt werden, bezeichnet man auch als *Groovlets*. Jedes Groovlet bekommt im Binding alle Informationen mitgeliefert, die auch ein Servlet zur Verfügung hat. Welche es sind, sehen wir an dem Groovlet, das in Beispiel 6-3 dargestellt ist.

Beispiel 6-3: Das Skript GroovletBinding.groovy

```
html.html (xmlns:"http://www.w3.org/1999/xhtml",lang:"de",'xml:lang':"de") {
  head {
    title "Groovlet-Binding"
  }
  body {
    h1 "Groovlet-Binding"
    table (border:1) {
      th "Name"
      th "Typ"
      th "Inhalt"
      for (var in binding.variables) {
        tr {
          td var.key
          td var.value.getClass().simpleName
          td var.value.toString()
        }
      }
    }
```

Beispiel 6-3: Das Skript GroovletBinding.groovy (Fortsetzung)

```
    for (param in params) {
        p (param.toString())
    }
  }
}
```

Das ganze Skript besteht aus einem einzigen Aufruf eines `MarkupBuilder`-Objekts, das in jedem Groovlet unter dem Namen `html` verfügbar ist. Das ist sehr praktisch, muss aber nicht so sein. Das Skript kann aus ganz normalem Groovy-Code bestehen. Sie brauchen auch nicht unbedingt den `MarkupBuilder` zu verwenden, sondern können die darzustellende Webseite auch aus normalen `print()`- und `println()`-Ausgaben zusammensetzen. Dieses Skript macht nichts weiter, als alle Binding-Variablen, die über die Property `binding.variables` in Form einer Map abgefragt werden können, in einer Tabelle darzustellen.

Speichern Sie das Skript im Wurzelverzeichnis Ihrer Webanwendung (also in *groovy*) unter dem Namen *GroovletBinding.groovy*. Rufen Sie dann im Browser die URL *http://localhost:8080/groovy/GroovletBinding.groovy* auf. Das Ergebnis sieht ungefähr so aus, wie in Abbildung 6-3 dargestellt.

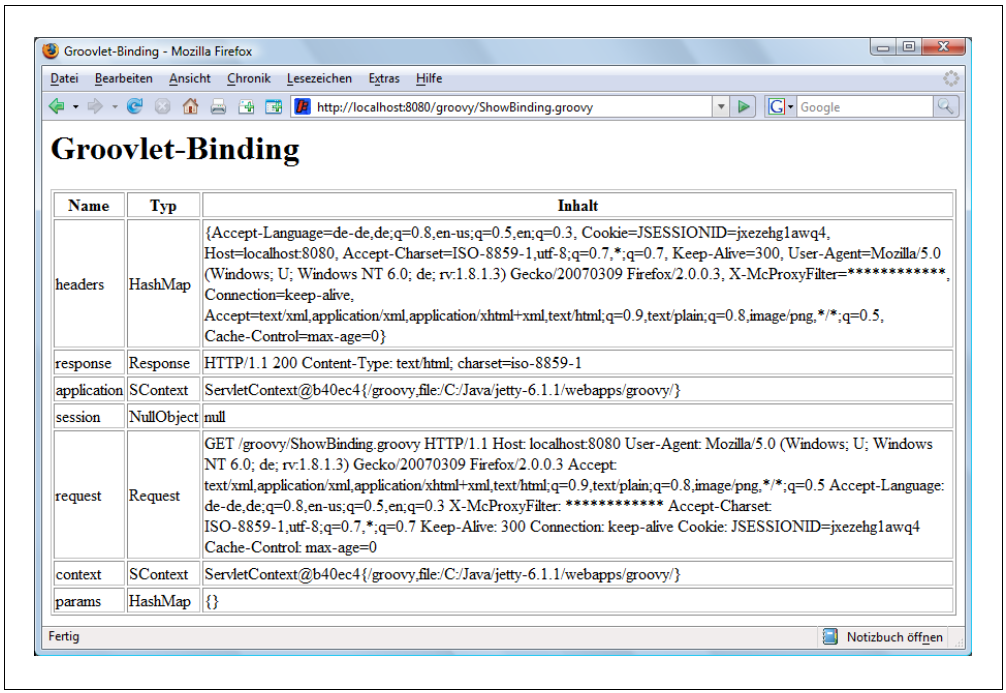


Abbildung 6-3: Das Skript GroovletBinding.groovy im Browser

In der folgenden Liste wollen wir Ihnen die Bedeutung der Binding-Variablen im Einzelnen erläutern.

`javax.servlet.ServletContext context`
: Der Servlet-Kontext, in dem das aktuelle Servlet läuft. Auf die Kontextvariablen kann wie auf Properties zugegriffen werden, z.B. `servername=context.serverInfo`. Eine Map mit allen Kontextvariablen erhält man mit `context.properties`. Der Servlet-Kontext kann auch unter dem Variablennamen `application` angesprochen werden.

`java.util.Map<String,String> headers`
: Map mit den Namen und Werten aller Header aus dem HTTP-Request.

`java.util.Map<String,Object> params`
: Map mit den Namen und Werten der Request-Parameter. Die Werte können Strings oder Arrays von Strings sein.

`javax.servlet.http.HttpServletRequest request`
: Request-Objekt des Servlet-Aufrufs. Auf die Request-Variablen kann wie auf Properties zugegriffen werden, z.B. `aufrufzeit=request.timeStampStr`. Eine Map mit allen Request-Variablen erhält man mit `request.properties`.

`javax.servlet.http.HttpServletResponse response`
: Response-Objekt des Servlet-Aufrufs.

`javax.servlet.http.HttpSession session`
: Aktuelles Session-Objekt. Wenn es keine Session gibt, ist der Wert `null`. Eine neue Session anlegen können Sie mit der Anweisung `session=request.session`.

  Auf die Session-Variablen kann wie auf Properties zugegriffen werden, z.B. `letzterZugriff=session.lastAccessTime`. Eine Map mit allen Session-Variablen kann erhält man mit `session.properties`.

Es stehen im Groovlet noch einige weitere Informationen zur Verfügung, auf die wie auf Binding-Variablen zugegriffen werden kann, obwohl sie dort nicht als reguläre Variablen gespeichert sind. Diese sind:

`groovy.xml.MarkupBuilder html`
: Vordefinierter Builder für die bequeme Erzeugung von HTML-Text. Das Ergebnis wird über `out` ausgegeben.

`java.io.PrintWriter out`
: Writer für die Ausgabe von Text. Die Variable wird erst bei ihrer ersten Verwendung erzeugt.

`javax.Servlet.ServletOutputStream sout`
: Binärer Ausgabe-Stream; die Variable wird erst bei ihrer ersten Vewendung erzeugt.

Der Writer und der Ausgabestrom werden in besonderer Weise behandelt, damit beispielsweise im Response-Objekt Header-Werte gesetzt werden können. Dies ist nicht mehr möglich, wenn schon Ausgaben erfolgt sind.

Wenn Sie etwas mit Groovlets experimentieren, werden Sie feststellen, dass Sie im Fall eines Fehlers in Ihrem Browser unter Umständen nicht viel sehen und kaum eine Chance haben, die Ursache zu ergründen, außer das Fehlerprotokoll des Servlet-Containers zu untersuchen. Laufzeitfehler können Sie allerdings besser sichtbar machen, indem Sie das gesamte Servlet in einen `try`-Befehl einkleiden:

```
try {
    //Hier den gesamten Groovlet-Code einfügen
} catch (Throwable th) {
    th.printStackTrace(out)
}
```

Dann wird der Stacktrace in die Webseite geschrieben. Es kann allerdings sein, dass Sie im Browser SEITENQUELLTEXT ANZEIGEN wählen müssen, um diese Information lesbar zu machen.

Das TemplateServlet

Wenn Sie Ihre Seiten nicht aus einem Skript heraus generieren möchten, sondern lieber Templates einsetzen wollen, um Daten und Präsentation zu trennen, bietet sich das `TemplateServlet` von Groovy an. Es funktioniert im Prinzip genau so wie das `GroovyServlet`, nur stellen Sie hier kein Skript, sondern ein Template ein. Das Template in Beispiel 6-4 soll dasselbe bewirken wie das obige Skript *GroovletBinding.groovy*. Es zeigt, dass im `TemplateServlet` die gleichen Binding-Variablen definiert sind wie im `GroovyServlet`.

Beispiel 6-4: Das Template TemplateBinding.html

```
<html xmlns="http://www.w3.org/1999/xhtml",lang="de",xml:lang="de">
<%
  session = request.session
  //if (session) { session.invalidate(); session = null }
%>
<head>
  <title>Template-Binding"</title>
</head>
<body>
  <h1>Template-Binding</h1>
  <table border="1">
    <tr><th>Name</th><th>Typ</th><th>Inhalt</th></tr>
<% for (var in binding.variables) { %>
    <tr>
      <td>$var.key</td>
      <td>${var.value.getClass().simpleName}</td>
      <td>${var.value.toString()}</td>
    </tr>
<% } %>
  </table>
</body>
</html>
```

Speichern Sie jetzt die Datei unter dem Namen *TemplateBinding.html* im *groovy*-Verzeichnis der Servlet-Engine ab, denn wir haben ja die Dateiendung *.html* mit dem `TemplateServlet` verknüpft. Wenn Sie nun in das Adressfeld des Browsers *http://localhost:8080/groovy/TemplateBinding.html* eintragen, erscheint eine Seite, die genau so aussieht wie oben in Abbildung 6-3 dargestellt, nur lautet der Titel jetzt »Template-Binding«.

Swing

Mit Swing implementierte grafische Benutzeroberflächen (GUIs) gehören häufig zu den komplexesten und unübersichtlichsten Komponenten von Java-Anwendungen. Sie bestehen aus einer Vielfalt von miteinander verknüpften Objekten mit diversen Parametern, deren Struktur kaum einen erkennbaren Zusammenhang mit der sichtbaren Oberfläche hat. Um die Arbeit zu erleichtern, enthalten die integrierten Java-Entwicklungsumgebungen typischerweise GUI-Designtools, die eine grafische Gestaltung der Oberflächen ermöglichen und den zugehörigen Java-Code generieren. Dieser ist aber häufig noch unübersichtlicher als der manuell programmierte; da er von den Werkzeugen anderer Toolhersteller oft nicht richtig interpretiert wird, kann dieses Vorgehen leicht zu einer Abhängigkeit von einem bestimmten Werkzeug führen.

Groovy bietet mit dem `SwingBuilder` ein Hilfsmittel, mit dem sich Swing-Oberflächen in einer Weise darstellen lassen, die die Struktur der Elemente auch im Programmcode erkennbar macht. Es basiert auf dem Builder-Prinzip, das wir schon in verschiedenen anderen Zusammenhängen kennengelernt haben und das ein quasi-deklaratorisches Herangehen erlaubt. Da man bei diesem Werkzeug – zumindest wenn man sich mit der Funktionsweise von Swing gut auskennt – das Ergebnis gut unter Kontrolle hat und verschiedene Programmiertechniken miteinander verknüpfen kann, kann das Arbeiten mit dem `SwingBuilder` durchaus effizienter sein als die Anwendung eines grafischen Gestaltungswerkzeugs.

So einfach die Arbeit mit dem `SwingBuilder` auf den ersten Blick erscheint, so unübersehbar ist doch die Menge der Möglichkeiten, die er zusammen mit der höchst komplexen Swing-API zur Gestaltung von Benutzungsoberflächen bietet. Aus Platzgründen werden wir uns hier darauf beschränken müssen, anhand einiger Beispiele das grundsätzliche Herangehen zu zeigen. Wenn Sie das Prinzip erst einmal verstanden haben, wird es Ihnen nicht schwerfallen, die weiteren Möglichkeiten anhand der Übersichten in Anhang C zu erkunden.

Das Vorgehen

Typischerweise bauen Sie mit dem `SwingBuilder` ein komplettes Bildschirmfenster oder einen kompletten Dialog (also ein Objekt der Klasse `JFrame` oder `JDialog`) auf, das Sie dann mit den üblichen Mitteln sichtbar machen. Das folgende Beispiel erstellt ein `javax.swing.JFrame`-Objekt mit zwei Buttons:

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

sb = new groovy.swing.SwingBuilder()

frame = sb.frame(title:"Klicken", size:[200,100],
                 defaultCloseOperation:WindowConstants.EXIT_ON_CLOSE) {
    panel() {
        button ("Rot",
                foreground:Color.RED,
                actionPerformed: {println "Rot gedrückt"})
        button ("Blau",
                foreground:Color.BLUE,
                actionPerformed: {println "Blau gedrückt"})
    }
}
frame.visible = true
```

Die Methodenaufrufe `frame()`, `panel()` und `button()` generieren jeweils ein Objekt einer Klasse mit dem korrespondierenden Namen `JFrame`, `JPanel` oder `JButton`. Dieses Prinzip wird weitgehend durchgängig für alle GUI-Komponentenklassen so gehandhabt. Allen Elementen werden die zu setzenden Properties in der Form von benannten Parametern mitgegeben. Bei Containerelementen wird eine Closure angegeben, die die Definitionen der enthaltenen Objekte umfasst. Zwei Besonderheiten fallen auf:

- Der benannte Parameter `size:[200,100]` in der `frame()`-Methode wird von Groovy automatisch in den Aufruf `setSize(200,100)` des `JFrame` umgesetzt, dies macht bereits die Sprache von sich aus.
- Die Parameter `actionPerformed` ordnet die übergebene Closure der jeweiligen Komponente als Handler für das `ActionEvent` zu; diese Closure wird also in diesen beiden Fällen beim Drücken der Buttons ausgeführt.

Wenn Sie dieses Skript laufen lassen, sollte ein Fenster auf dem Bildschirm erscheinen, das wie Abbildung 6-4 aussieht.

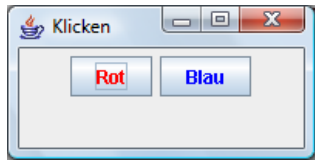


Abbildung 6-4: Beispielfenster mit zwei Buttons

Wenn Sie auf die Buttons klicken, erscheint jeweils der Text »Rot gedrückt« oder »Blau gedrückt« auf der Konsole. Da wir die Property `defaultCloseOperation` entsprechend gesetzt haben, wird die Anwendung beim Schließen des Fensters beendet.

Aktionen

Einfache Aktionen lassen sich gut als Closure per `actionPerformed`-Property den Elementen zuordnen. Wenn Aktionen länger sind oder wiederverwendet werden sollen, bietet sich dieses Vorgehen weniger an. Natürlich können Sie aus der Closure eine Methode aufrufen, die die eigentliche Arbeit macht. Bessere Möglichkeiten bieten aber spezielle Aktionen, die Sie sich vom `SwingBuilder` mithilfe seiner Methode `action()` erzeugen lassen und die bei der auslösenden Komponente nur noch referenziert werden. In unserem Beispiel bietet es sich etwa an, nur eine Ausgabeaktion zu definieren.

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

sb = new groovy.swing.SwingBuilder()

anzeigeAktion = sb.action(closure: { println "$it.source.text gedrückt"})

frame = sb.frame(title:"Klicken", size:[200,100],
                 defaultCloseOperation:WindowConstants.EXIT_ON_CLOSE) {
    panel() {
        button ("Rot",foreground:Color.RED, action:anzeigeAktion)
        button ("Blau",foreground:Color.BLUE,  action:anzeigeAktion)
    }
}
frame.visible = true
```

Das Ergebnis unterscheidet sich nicht vom obigen Beispiel.

Layout

Eine Stärke von Swing besteht in der Möglichkeit, mithilfe von als Layout-Manager bezeichneten Objekten die Darstellung der Oberfläche sehr vielfältig steuern zu können. Der Layout-Manager kann dem Containerelement als gewöhnliche Property zugeordnet werden (z.B. `layout=GridBagLayout`). Einfacher aber ist es, einen Methodenaufruf, der mit dem Namen des Layout-Managers korrespondiert, einzutragen – als wäre es ein enthaltenes Element. Um den Layout-Manager `GridBagLayout` zu verwenden, benutzen wir also den Methodenaufruf `gridBagLayout()`. Die erforderlichen Layout-Constraints können dann bei den enthaltenen Elementen als Properties gesetzt werden.

Im folgenden Beispiel wollen wir nun die Textausgabe nicht mehr auf der Konsole ausführen, sondern in ein zusätzliches Textfeld leiten. Mithilfe eines `GridBagConstraint` legen wir fest, dass das Textfeld unterhalb beider Buttons liegen soll. Für die beiden Buttons brauchen wir hier keine Constraints, da deren Anordnung dem Standard entspricht.

Um den Ausgabetext in das Textfeld eintragen zu können, müssen wir es irgendwie referenzieren. Dazu gibt es mehrere Möglichkeiten: Wir könnten durch die Hierarchie der Elemente navigieren oder die Elemente Variablen zuweisen. Am einfachsten aber ist es,

den Elementen mithilfe der Property id eine eindeutige Kennung zuzuweisen; unter dieser Kennung erscheinen sie dann wieder als Property der SwingBuilder-Instanz. In diesem Beispiel ordnen wir dem Textfeld die Kennung »ausgabe« zu und können es so mit sb.ausgabe referenzieren.

```
import groovy.swing.SwingBuilder
import javax.swing.*
import java.awt.*

sb = new groovy.swing.SwingBuilder()

anzeigeAktion = sb.action(closure: { sb.ausgabe.text="$it.source.text gedrückt"})
println anzeigeAktion.getClass().name

frame = sb.frame(title:"Klicken", size:[200,100]
                 defaultCloseOperation:WindowConstants.EXIT_ON_CLOSE) {
    panel() {
        gridBagLayout()
        button ("Rot",foreground:Color.RED, action:anzeigeAktion)
        button ("Blau",foreground:Color.BLUE,  action:anzeigeAktion
        textField(id:'ausgabe',
          constraints:new GridBagConstraints(gridx:0,gridy:1,gridwidth:2,fill:1))
    }
}
frame.visible = true
```

Das Ergebnis sieht so aus:

Abbildung 6-5: Beispielfenster mit zusätzlichem Textfeld

Sie haben jetzt hoffentlich einen ersten Eindruck davon bekommen, wie in Groovy Swing-Oberflächen gestaltet werden können. Eine Übersicht der verschiedenen Factory-Methoden, mit denen Sie GUI-Komponenten, Layout-Manager und weitere Objekte erzeugen können, finden Sie in Anhang B.

Damit ist auch unser kleiner Rundgang durch die Groovy-API beendet. Sie haben nun einen weitgehend kompletten Überblick darüber, was Ihnen in Groovy über die bloße Programmiersprache hinaus geboten wird. Zwei spezielle Themengebiete fehlen allerdings noch, denen wir uns jetzt zuwenden wollen.

KAPITEL 7

Dynamisches Programmieren

Wenn Sie in Java den Aufruf der Methode `m` eines Objekts `o` der Klasse `K` programmieren, wird dieser Aufruf in ein Stück Bytecode übersetzt, das nichts anderes tut, als genau die von Ihnen genannte Methode `m` der Klasse `K` (oder einer abgeleiteten Klasse) für das Objekt `o` aufzurufen. Nicht mehr und nicht weniger.

Da ist Groovy ganz anders. Der Methodenaufruf wird durch eine Kette von Framework-Klassen hindurchgeleitet, an deren Ende normalerweise auch der Aufruf der Methode `m` der Klasse `K` für das Objekt `o` steht. Abhängig von den Gegebenheiten kann aber auch etwas ganz anderes geschehen, beispielsweise kann eine Closure aufgerufen werden, die einer Property von `o` mit demselben Namen wie `m` zugewiesen worden ist. In dieser Kette von weiterdelegierten Methodenaufrufen gibt es mehrere Möglichkeiten, sich gewissermaßen einzuklinken und den Lauf des Geschehens abzuwandeln. Auf diese Weise können Sie zum Beispiel dynamische Methoden einführen, die in der Klasse nicht definiert sind, sondern deren Funktionalität sich zur Laufzeit aus dem Methodennamen ergibt. Oder Querschnittsmethoden im Stil der aspektorientierten Programmierung, mit denen Aufrufe beliebiger anderer Methoden quer durch das ganze System – etwa zur Protokollierung – abgefangen werden können.

Dies alles gibt Ihnen die Möglichkeit, die Semantik der Sprache stark zu beeinflussen. Die Groovy-Standardbibliothek macht von dieser Möglichkeit verschiedentlich Gebrauch; ein Beispiel dafür sind die Ihnen bereits vertrauten Closures, weitere Beispiele folgen weiter unten in diesem Kapitel. Auch verschiedene »Groovy-Module« (auf Groovy basierende Tools oder Erweiterungen) nutzen die Möglichkeiten, die sich aus den dynamischen Fähigkeiten von Groovy ergeben, allen voran die bekannte Web-Entwicklungsumgebung Grails, die einen großen Teil ihrer Einfachheit und Flexibilität der Nutzung von Groovy als Basis verdankt. Diese Dynamik ist zudem die Grundlage für die leichte Umsetzung von Domain Specific Languages (DSL) mit Groovy.

Um das dynamische Programmieren mit Groovy richtig zu verstehen, muss man sich zwangsläufig in gewissem Umfang mit den Innereien der Sprache und der Laufzeitumgebung auseinandersetzen. Dies werden wir hier aus Platzgründen nur in begrenztem Maße tun können; trotzdem sollten Sie die wichtigsten Mechanismen kennen, die Groovy-Objekte zu dynamischen Objekten machen. Denn dieses Wissen kann für das Verständnis der Sprache und dessen, was in einem Programm vor sich geht – vor allem, wenn einmal *nichts* geht –, von großem Nutzen sein. Zunächst wenden wir uns mit dem Expando und den Kategorienklassen erst einmal zwei Möglichkeiten zu, die wir nutzen können, ohne allzu viel von den internen Mechanismen verstehen zu müssen.

Das Expando

In Kapitel 4 haben wir bereits gezeigt, wie sich mit in einer Map gespeicherten Closures bereits fast ein komplettes Objekt zaubern lässt. Ein Objekt ist ja eigentlich etwas nicht viel anderes als ein Behälter für Code (Methoden) und Daten (Felder). Es gibt da nur ein Problem: Die Closures werden zwangsläufig in einem anderen Kontext als dem dieser als Quasi-Objekt dienenden Map definiert und »sehen« damit nicht den Inhalt der Map, können also nicht direkt auf die eigenen Member zugreifen. Und das würde man sich bei einem richtigen Objekt schon wünschen.

Die Groovy-Standardbibliothek enthält eine Klasse `Expando`, die genau dieses leistet: Sie ist ein Behälter für Closures und andere Daten, und der Inhalt des Expando wird automatisch zum Namensraum der zugeordneten Closures. Wir können das interaktiv ausprobieren:

```
groovy> x = new Expando(
groovy>   feld: "Feld im Expando",
groovy>   methode1: { "Methode 1 - "+feld },
groovy>   methode2: { println "Methode 2 - "+methode1() }
groovy> )
groovy> x.methode2()
Methode 2 - Methode 1 - Feld im Expando
```

Wir weisen dem Expando einen String als Property `feld` zu sowie zwei Closures `methode1` und `methode2`. Die Closure `methode2` ruft `methode1` auf, und diese referenziert `feld`. Das Entscheidende hier ist, dass die Referenz von `methode2` auf `methode1` und die Referenz von `methode1` auf `feld` wie lokale Referenzen in den Namensraum der Closure, also das umgebende Skript aussehen. Trotzdem finden sie die dem Expando zugeordneten Elemente.

Der Trick besteht darin, dass sich das Expando beim Zuordnen der Closures bei diesen selbst als Delegate einträgt. Daher wird die Suche nach einer Property, wenn sie in der Closure selbst oder beim Eigentümerobjekt (der Klasse oder dem Skript, innerhalb derer das Expando-Objekt erzeugt wird) nicht fündig wird, automatisch im Expando fortgesetzt. Und somit werden `feld` und `methode1` auch mit unqualifizierten Referenzen gefunden. Außerdem ist die Methode `invokeMethod()` so überschrieben: Immer, wenn eine aufzurufende Methode nicht gefunden wird, schaut sie nach, ob dem Expando eine passende

Closure mit dem Namen der Methode zugeordnet ist. Und wenn sie eine solche Closure findet, ruft sie diese anstelle der Methode auf.

Expandos können also im Groovy-Kontext fast wie Objekte verwendet werden, denen man Felder und Methoden zur Laufzeit zuordnen kann. Sie sind damit wie geschaffen für generativ erzeugte Ad-hoc-Objekte wie Datencontainer, Transferobjekte usw.

Zu beachten ist, dass die Auflösung der Referenzen im Expando immer nachgelagert ist. Wenn es im Eigentümerobjekt bereits ein Feld `feld` oder eine Methode `methode1` gäbe, würden diese anstelle der gleichnamigen Elemente des Expando gefunden werden. Insofern ist etwas Vorsicht bei der Anwendung von Expando-Objekten angebracht.

Das Expando eignet sich gut als Dummy-Objekt für Testzwecke oder für sonstige Einmal-Objekte, da Sie nicht eigens eine Klasse definieren müssen. Allerdings ist die Verwendbarkeit etwas eingeschränkt, da Expandos keine Vererbung und kein Überladen von Methoden kennen, keine Interfaces implementieren können und auch von Java-Klassen aus nicht sinnvoll nutzbar sind. In vielen Fällen bietet es sich an, anstelle von Expandos einfache Closures oder Maps von Closures einzusetzen, mit denen sich sogar beliebige Interfaces implementieren lassen (siehe Kapitel 4). Demgegenüber haben Expandos den Vorzug, dass sie eine Art lokalen Status kennen, auf den die Closure-Methoden direkt zugreifen können.

Eigene Methoden vordefinieren mit Kategorienklassen

In Kapitel 5 haben wir Ihnen gezeigt, wie Groovy es schafft, vorhandenen Java-Klassen und -Interfaces mittels vordefinierter Methoden gewissermaßen von außen zusätzliche Funktionalität zu verleihen. Diesen Mechanismus können Sie in Ihren Programmen ihn ähnlicher Weise nutzen. Der Schlüssel hierzu ist die vordefinierte Methode `use()`, der bisweilen auch fälschlich als `use`-Anweisung bezeichnet wird. Das geht folgendermaßen:

1. Bauen Sie eine Klasse mit statischen Methoden nach dem Muster, wie wir es in Kapitel 5 für die Klasse `DefaultGroovyMethods` erläutert haben. Sie muss weder ein bestimmtes Interface implementieren noch von einer bestimmten Klasse abgeleitet sein. Der erste Parameter bestimmt den Typ, für den die Methode gelten soll, und die übrigen Parameter nehmen die effektiven Argumente des Methodenaufrufs entgegen. Das ist die Kategorienklasse.
2. Kapseln Sie Ihr Programm – oder den Teil des Programms, in dem die von Ihnen definierten vorgegebenen Methoden gelten sollen – in einer Closure, die Sie zusammen mit Ihrer Kategorienklasse an die `use()`-Methode übergeben.

Einen wesentlichen Unterschied gibt es zwischen den Groovy-eigenen vordefinierten Methoden und denen, die Sie über `use()` einführen: Erstere kommen nur ins Spiel, wenn es keine passenden Methoden bei den Objekten selbst gibt, an denen sie ausgeführt werden sollen. Ihre Methoden haben dagegen Vorrang vor allen anderen Methoden. Um dies zu demonstrieren, bauen wir uns eine Klasse namens `AndereKlasse`, in der es eine

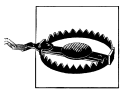
Methode `dump()` gibt, die mit ihrer Signatur mit der einer gleichnamigen vordefinierten Methode übereinstimmt. Wir rufen diese Methode einmal allein auf und einmal im Kontext einer Kategorienklasse `DumpKategorie`, die diese Methode ebenfalls definiert.

```
class DumpKategorie {
    static void dump(Object self) {
        println "DumpKategorie: $self"
    }
}

class AndereKlasse {
    def dump() {
        println "AndereKlasse: ${this}"
    }
}

k = new AndereKlasse()
k.dump()

use (DumpKategorie) {
    k.dump()
}
```

Und tatsächlich erscheinen beim Ausführen dieses Skripts folgende Zeilen auf dem Bildschirm:

```
AndereKlasse: AndereKlasse@181afa3
DumpKategorie: AndereKlasse@181afa3
```

Beim ersten Mal wird die in der Klasse implementierte Methode aufgerufen und nicht die vordefinierte Methode, innerhalb des `use`-Blocks dagegen kommt die entsprechende Methode der Kategorienklasse zum Zuge, da sie Vorrang hat.



Es versteht sich fast von selbst, dass Sie mit den Kategorien in Groovy extrem vorsichtig sein müssen. Wenn Sie versehentlich eine existierende Methode überschreiben, kann dies zu unerwarteten und schwer zu lokalisierenden Nebenwirkungen führen. Eine empfehlenswerte Vorsichtsmaßnahme besteht darin, die Auswirkungen der Kategorien auf spezifische Klassen zu beschränken und nur auf begrenzte Teile des Programms einwirken zu lassen.

Im folgenden Beispiel wollen wir die Funktionalität einer einzelnen vorhandenen Klasse aufbessern. In Kapitel 3 hatten wir eine dünne Kapsel um die `Calendar`-Klasse gebaut, die einen bequemen Zugriff auf einzelne Felder im Stil von Groovy-Properties ermöglicht. Als Alternative können wir auch eine Kategorienklasse mit vordefinierten Methoden für den Typ `Calendar` anlegen, die dasselbe leisten. Der Vorteil ist, dass wir dadurch eine Reihe von Methoden zur Verfügung haben, die wir direkt an einem `Calendar`-Objekt anwenden können, auch wenn dieses beispielsweise von einer anderen Klasse instantiiert wird, die von unserer Erweiterung nichts wissen kann.

```
class KalenderDatumKategorie {
   // Eine Property auslesen
   static int get(Calendar calendar, String propertyname) {
       int wert = calendar.get(calendarConst(propertyname))
       // Null-relativen in 1-relativen Monat umwandeln
       propertyname=='month' ? wert+1 : wert
   }
   // Eine Property setzen
   static void set(Calendar calendar, String propertyname, int wert) {
       // 1-relativen in null-relativen Monat umwandeln.
       if (propertyname=='month') wert--;
       calendar.set(calendarConst(propertyname),wert)
   }
   // Die toString()-Methode von Calendar überschreiben
   static String toString(Calendar calendar) {
       calendar.identity { "${dayOfMonth}.${month}.${year}" }
   }
   // CamelCase in Konstantennamen umwandeln
   private static calendarConst(String propertyname) {
       try {
          def constname = propertyname.replaceAll('[A-Z]','_$0').toUpperCase();
         return Calendar."$constname"
       } catch (MissingPropertyException ex) {
          ex.printStackTrace()
          throw new MissingPropertyException(propertyname,Calendar)
       }
   }
}
```

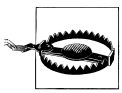
Und so probieren wir die Kategorienklasse aus:

```
use (KalenderDatumKategorie) {
    def cal = new GregorianCalendar()
    println cal.toString()
    cal.dayOfMonth ++  // Um einen Tag erhöhen
    println cal.toString()
}
```

Wenn Sie dieses Skript zufällig am Silvesterabend aufrufen, sieht die Ausgabe korrekt so aus:

```
31.12.2007
1.1.2008
```

Das heißt also, dass wir unsere eleganten Zugriffsmöglichkeiten der `Calendar`-Klasse direkt zuordnen und gar keine Kapselung mehr brauchen.



Man vergisst es zwar leicht, aber außerhalb des Einflussbereichs von Groovy haben die durch Kategorien zugeordneten Methoden keine Gültigkeit mehr. Deswegen müssen wir in dem Beispiel auch `println cal.toString()` schreiben. Ohne `.toString()` würde das `Calendar`-Objekt als Ganzes an `System.out` übergeben werden. Aber dies ist ein ganz normale Java-`PrintStream`, der nichts von Groovys Kategorien weiß und deshalb nur das normale `toString()` des `Calendar` aufrufen kann.

Dynamische Objekte

Zunächst müssen wir begrifflich zwischen Groovy- und Java-Objekten unterscheiden. Groovy-Objekte sind alle Objekte, deren Klassen das Interface `groovy.lang.GroovyObject` implementieren. Das ist bei allen Objekten der Fall, deren Klassen in der Sprache Groovy geschrieben sind, aber natürlich können auch in Java geschriebene Klassen dieses Interface implementieren; genau dies ist bei einer Reihe von Klassen der Groovy-Laufzeitbibliothek der Fall, die damit auch zu Groovy-Objekten werden. Mithilfe der Klasse `groovy.lang.GroovyObjectSupport` ist es auch für Sie kein Problem, eine solche Klasse in Java zu schreiben. Java-Objekte sind alle übrigen Objekte, deren Klassen in Java (oder einer anderen Programmiersprache, die Java-Bytecode erzeugt) geschrieben sind und nicht `GroovyObject` implementieren.

Eine Groovy-Klasse von innen

Als Beispiel schnitzen wir uns eine kleine Groovy-Klasse, die an Primitivität kaum zu überbieten ist, für unsere Untersuchungen aber vollständig ausreicht.

```
class MeineKlasse {
    def inhalt
}
```

Die Klasse hat eine untypisierte Property namens `inhalt`, und wir wissen, dass Groovy von selbst zwei Zugriffsmethoden hinzugeneriert, nämlich `getInhalt()` und `setInhalt()`. Wenn Groovy die Klasse übersetzt, sieht sie in Java formuliert ungefähr folgendermaßen aus:[1]

```
// Java-Äquivalent der Klasse MeineKlasse
import groovy.lang.*;
import org.codehaus.groovy.runtime.InvokerHelper;

public class MeineKlasse implements GroovyObject
{
    // Property "inhalt"
    private Object inhalt;
    // Referenz auf Metaklasse
    transient MetaClass metaClass;
    // Zeitstempel
    public static Long __timeStamp = new Long(1170538070965L);

    /**
     * Getter für die Property "inhalt"
     */
    public Object getInhalt() {
```

1 Wenn Sie den kompilierten Code mit einem Java-Decompiler rückübersetzen, sieht das Ergebnis um einiges komplizierter aus, entspricht aber in der Funktionalität etwa dem hier wiedergegebenen Java-Listing.

```
        return inhalt;
    }

    /**
     * Setter für die Property "inhalt"
     */
    public void setInhalt(Object value) {
        inhalt = value;
    }

    /**
     * Ruft Methoden dieses Objekts auf
     */
    public Object invokeMethod(String method, Object arguments) {
        return getMetaClass().invokeMethod(this, method, arguments);
    }

    /**
     * Liest eine Property dieses Objekts aus
     */
    public Object getProperty(String property) {
     return getMetaClass().getProperty(this, property);
    }

    /**
     * Setzt eine Property dieses Objekts
     */
    public void setProperty(String property, Object value) {
     getMetaClass().setProperty(this, property, value);
    }

    /**
     * Liefert die diesem Objekt zugeordnete Metaklasse
     */
    public MetaClass getMetaClass() {
        MetaClass meta = metaClass;
        if (meta==null) {
            meta = InvokerHelper.getMetaClass(this);
        }
        return meta;
    }

    /**
     * Setzt Metaklasse für dieses Objekt
     */
    public void setMetaClass(MetaClass value) {
     metaClass = value;
    }
}
```

Sie erkennen die konventionellen Getter- und Setter-Methoden für die Property `inhalt`. Diese werden ganz normal aufgerufen, wenn wir beispielsweise ein Objekt dieser Klasse in einem Java-Programm verwenden. Daher lassen sich Groovy-Objekte auch problemlos

im Java-Kontext verwenden. Der ganze Rest ist aber genau das, was ein Objekt zu einem Groovy-Objekt macht. Es implementiert das Interface `GroovyObject` und muss daher auch die darin definierten Methoden anbieten.

Die Groovy-Laufzeitbibliothek bietet auch eine Klasse `groovy.lang.GroovyObjectSupport`, die standardmäßige Implementierungen der `GroovyObject`-Methoden enthält. Die Klasse `MeineKlasse` kann aber nicht einfach von `GroovyObjectSupport` abgeleitet werden, da wir dann nicht mehr die Möglichkeit hätten, eigene Ableitungshierarchien aufzubauen. Somit müssen diese Methoden vom Groovy-Compiler hinzugeneriert werden.

- Die Methode `invokeMethod()` ist dran, wenn eine Instanzmethode des Objekts aus demselben oder einem anderen Groovy-Objekt heraus aufgerufen werden soll – dies normalerweise aber nur, wenn keine Methode mit der passenden Signatur existiert.[2] Sie leitet den Aufruf an eine Metaklasse weiter, die entweder durch eine eigene Referenz oder, wenn diese nicht gesetzt ist, über die Groovy-interne Klasse `InvokerHelper` ermittelt wird.
- Die Methoden `getProperty()` und `setProperty()` kommen immer dann ins Spiel, wenn lesend oder schreibend auf eine nicht statische Property des Objekts zugegriffen werden soll, z.B. über einen Ausdruck in der Property-Notation wie `mk.inhalt` aus einem anderen Namensraum heraus.
- Die beiden Zugriffsmethoden `getMetaClass()` und `setMetaClass()` ermöglichen es, die dem Objekt zugeordnete Metaklasse auszulesen oder zu setzen.

Die Metaklasse

Die generierte Klasse enthält eine Referenz auf eine Groovy-Metaklasse, mit der alle zusätzlich generierten Methoden zu tun haben. Die Metaklasse ist gewissermaßen der Dreh- und Angelpunkt der dynamischen Objektorientierung in Groovy, die Schaltstelle, über die alle Methodenaufrufe und Property-Referenzen geleitet werden.

Die Metaklasse ist eine Instanz einer Klasse, die von der abstrakten Klasse `groovy.lang.MetaClass` abgeleitet ist und über alle Informationen bezüglich einer bestimmten Java- oder Groovy-Klasse verfügt, die Groovy benötigt, um seine dynamischen Fähigkeiten ausspielen zu können. Der Zusammenhang zwischen einer Klasse und der korrespondierenden Metaklasseninstanz wird durch ein Registraturobjekt (`groovy.lang.MetaClassRegistry`) hergestellt. Im Normalfall verwendet Groovy für jede in einem Programm auftretende Klasse genau eine Instanz der Klasse `groovy.lang.MetaClassImpl`; es ist aber möglich, für einzelne Klassen Instanzen anderer Ableitungen von `MetaClass` zu registrieren und damit das Verhalten dieser Klassen wesentlich zu verändern.

2 Als Signatur bezeichnet man die Kombination aus Methodennamen und Parametertypen, die eine Methode eindeutig bezeichnet.

Die auf Klassen bezogene Zuordnung von Metaklassen kann auch auf Instanzebene überschrieben werden. Wie Sie im obigen Listing sehen, hat jedes Groovy-Objekt eine eigene Referenz auf die Metaklasse, die mithilfe von Getter- und Setter-Methoden gelesen und geschrieben werden kann. Sobald einem einzelnen Objekt eine eigene Metaklasseninstanz zugeordnet ist, verwendet es diese und nicht mehr die Metaklasseninstanz, die über die Registratur seiner Klasse zugeordnet ist.

Beispiele für die Möglichkeiten, die sich aus dem Austausch der Metaklasse auf Klassen- oder Instanzebene ergeben, finden Sie weiter unten im Abschnitt »Das Meta-Objekt-Protokoll«.

Aufruf von Methoden einer Groovy-Klasse

Was geschieht, wenn wir aus einem Groovy-Programm heraus eine Methode der Groovy-Klasse aufrufen? Als Beispiel nehmen wir das folgende Codestück, das die `toString()`-Methode aufruft.

```
groovy> mk = new MeineKlasse()
groovy> println mk.toString()
MeineKlasse@123456
```

Dabei interessieren wir uns eigentlich nur für den Ausdruck `mk.toString()`, der den Aufruf enthält. Er wird vom Groovy-Compiler in Bytecode übersetzt, der etwa dem folgenden Java-Ausschnitt entspricht:

```
// Java-Äquivalent für mk.toString()
org.codehaus.groovy.runtime.InvokerHelper.invokeMethod(mk,"toString",null);
```

Es wird also gar nicht direkt die Methode `toString()` der `MeineKlasse`-Instanz aufgerufen, sondern der Aufruf wird an eine statische Methode der internen Framework-Klasse `InvokerHelper` delegiert. Dieser Methode werden eine Referenz auf das Objekt, bei dem die Methode aufgerufen werden soll, der Name der Methode sowie, falls vorhanden, die Aufrufargumente als Array von Objekten mitgegeben (da wir keine Argumente haben, ist es hier einfach `null`).

`InvokerHelper` gehört zu einer Reihe von Klassen im Package `org.codehaus.groovy.runtime`, deren Schnittstellen nicht standardisiert sind und die hauptsächlich dazu dienen, die Menge des vom Compiler zu generierenden Codes zu minimieren. Was da im Einzelnen vor sich geht, soll uns nicht weiter interessieren. Wichtig ist aber, dass im Endeffekt Folgendes geschieht:

1. Ist das Ziel des Aufrufs eine Klasse und keine Instanz, wird der Aufruf als statischer Methodenaufruf an die für diese Klasse registrierte Metaklasse weiterdelegiert.
2. Ist es eine Instanz einer Java-Klasse (die nicht `GroovyObject` implementiert), wird der Aufruf ebenfalls an die für diese Java-Klasse registrierte Metaklasse weiterdelegiert.

3. Ist es ein Groovy-Objekt und implementiert die Klasse *nicht* `GroovyInterceptable`, wird mit `getMetaClass()` die zutreffende Metaklasseninstanz ermittelt und der Aufruf gleich an diese weitergeleitet.
4. Kann die Metaklasseninstanz den Aufruf nicht durchführen, weil bei dem Zielobjekt nichts mit passender Signatur zu finden ist, wird direkt `invokeMethod()` beim Zielobjekt aufgerufen.

Hier haben wir es mit dem dritten Fall zu tun: Da es sich bei `mk` um ein Groovy-Objekt handelt, landet der Aufruf erst einmal bei der Methode `invokeMethod()` dieses Objekts. Diese Methode leitet nun, wie wir oben sehen können, den Aufruf ihrerseits weiter an die Metaklasseninstanz, die dem Objekt zugeordnet ist und die schließlich dafür sorgt, dass die Methode `toString()` per Reflection aufgerufen wird. Abbildung 7-1 zeigt ein vereinfachtes UML-Kollaborationsdiagramm, das die wesentlichsten Schritte der Delegationskette für diesen Fall verdeutlicht.

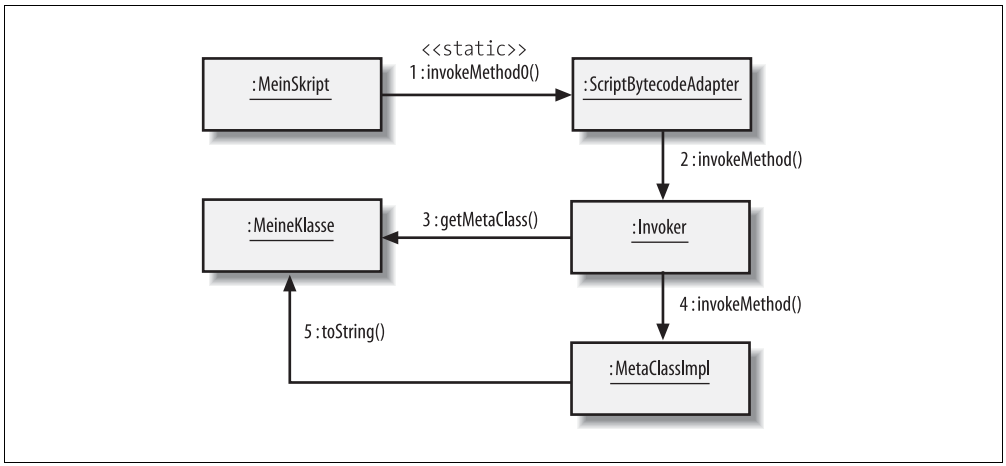


Abbildung 7-1: Aufruf der Methode toString() als Kollaborationsdiagramm

Bevor der Aufruf von `toString()` endlich bei dieser Methode angelangt ist, hat er also schon einen beträchtlichen Weg hinter sich. Es wird klar, dass Groovy-Programme niemals so performant sein können wie äquivalente Java-Programme. Dafür bieten sie verschiedene Möglichkeiten, in das Geschehen einzugreifen; und mit diesen wollen wir uns im Folgenden beschäftigen.

Vorgegaukelte Methoden

Die vom Compiler generierten Methoden sind immer als freundliches Angebot zu verstehen, das anzunehmen Sie nicht gezwungen sind. Wir prüfen dies, indem wir unsere simple Groovy-Klasse mit einer zusätzlichen `invokeMethod()`-Methode versehen. Sie soll nichts weiter tun, als Laut zu geben, sobald sie aufgerufen wird.

```
class MeineKlasse {
    def inhalt
      def invokeMethod (String name, Object args) {
         println "invokeMethod aufgerufen: $name $args"
    }
}
```

Nun versuchen wir verschiedene Methodenaufrufe:

```
groovy> mk = new MeineKlasse()
groovy> println mk.toString()
groovy> println mk.toString("Hallo")
groovy> println mk.tunix()
MeineKlasse@3bc1a1
invokeMethod aufgerufen: toString {"Hallo"}
null
invokeMethod aufgerufen: tunix {}
null
```

Ohne unsere selbst gemachte `invokeMethod()`-Methode würden wir schon bei `mk.toString("Hallo")` einen Laufzeitfehler bekommen, denn Folgendes würde passieren: Die Laufzeitumgebung delegiert den Methodenaufruf gleich am Anfang an die zu `MeineKlasse` gehörende Metaklasse weiter und, da diese keine passende Methode findet, wendet sich dann als letzte Rettung an `invokeMethod()` in der Zielklasse. In der Originalversion delegiert `invokeMethod()` erneut an die Metaklasse, was natürlich immer noch nicht zum Ziel führen kann und schließlich die Runtime-Exception auslöst.

Unsere eigene Implementierung von `invokeMethod()` spart sich die Weiterleitung an die Metaklasse – diese muss ja immer zu einem Fehler führen, denn `invokeMethod()` wird schließlich nur aufgerufen, nachdem die Metaklasse schon einmal den Methodenaufruf erfolglos probiert hat. Stattdessen gibt sie eine kleine Meldung aus. An der Ausgabe des Skripts sehen Sie zudem, dass der erste `toString()`-Aufruf auch korrekt ausgeführt wird. Nur anstelle der beiden anderen Aufrufe, wobei der eine ein überzähliges Argument hat und der zweite eine überhaupt nicht existierende Methode anspricht, tritt unser `invokeMethod()` in Aktion und zeigt an, dass es aufgerufen wurde. Da die Methode kein Ergebnis zurückliefert, gibt `println()` in beiden Fällen `null` aus.

Der typische Anwendungsfall für eigene `invokeMethod()`-Implementierungen sind Klassen, die sich die Syntax von Methodenaufrufen zunutze machen, um etwas ganz anderes zu bewirken. In den Kapiteln 4 und 6 haben Sie am Beispiel der Builder bereits gesehen, welche Möglichkeiten dies eröffnet.

Interzeptoren

Der oben dargestellte Ablauf eines Methodenaufrufs im `Invoker` sieht etwas anders aus, wenn die Klasse des Zielobjekts das Interface `groovy.lang.GroovyInterceptable` implementiert. Dies ist ein Marker-Interface, das selbst keine Methoden definiert, aber der Groovy-Laufzeitumgebung signalisiert, dass alle Methodenaufrufe an die eigene `invokeMethod()`-

Methode geleitet werden sollen und nicht zuerst an die Metaklasse. Abbildung 7-2 stellt auch diesen Ablauf grafisch dar.

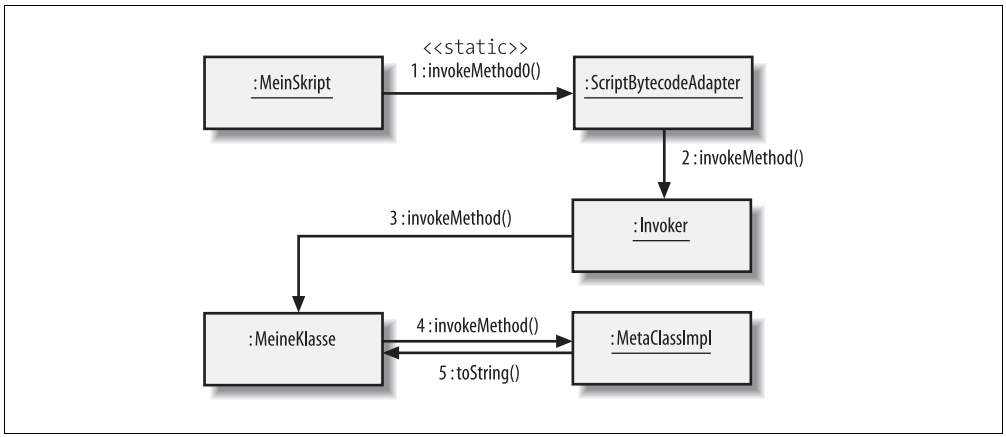


Abbildung 7-2: Methodenaufruf bei einem Interceptable-Objekt

Um das zu zeigen, bauen wir `MeineKlasse` noch etwas um, sodass sie `GroovyInterceptable` implementiert. Dabei ändern wir auch die `println()`-Anweisung in `invokeMethod()` so, dass direkt `System.out` angesprungen wird, um eine Endlosschleife zu vermeiden.

```
class MeineKlasse implements GroovyInterceptable {
    def inhalt
    def invokeMethod (String name, Object args) {
        System.out.println "invokeMethod aufgerufen: $name $args"
    }
}
```

Das Ergebnis der interaktiven Skriptzeilen mit den drei Methodenaufrufen sieht nun anders aus:

```
groovy> mk = new MeineKlasse()
groovy> println mk.toString()
groovy> println mk.toString("Hallo")
groovy> println mk.tunix()
invokeMethod aufgerufen: toString {}
null
invokeMethod aufgerufen: toString {"Hallo"}
null
invokeMethod aufgerufen: tunix {}
null
```

In allen drei Fällen ist unser `invokeMethod()` aufgerufen worden, und das eigentliche `toString()` wurde offenbar überhaupt nicht erreicht.

Das Interface `GroovyInterceptable` ist natürlich nicht dazu gedacht, alle Methodenaufrufe abzublocken. Vielmehr ermöglicht es dem Programmierer, Interzeptoren zu bauen,

das sind Klassen, die jeden Methodenaufruf abfangen, um irgendeine Querschnittsaktion auszuführen. Typische Anwendungsfälle dafür sind Protokollierung und Zugriffskontrollen. An einem simplen Beispiel wollen wir zeigen, wie dies im Prinzip funktionieren kann.

Wir definieren uns eine Klasse namens `ProtokollObjekt`. Sie ist als Basisklasse für beliebige andere Klassen gedacht, deren Methodenaufrufe automatisch protokolliert werden sollen. Wir definieren sie aber nicht ausdrücklich als `abstract`, da sie keine abstrakten Methoden enthält und prinzipiell instantiierbar ist.

```
import java.util.logging.*

class ProtokollObjekt implements GroovyInterceptable {
    def invokeMethod (String name, Object args) {
        Class klasse = metaClass.invokeMethod(this,"getClass")
        String klassenname = klasse.name
        Logger logger = Logger.getLogger (klassenname)
        logger.info "Aufruf $klassenname.$name $args"
        try {
            def ergebnis = metaClass.invokeMethod (this,name,args)
            logger.info "Ergebnis $klassenname.$name: $ergebnis"
            return ergebnis
        } catch (Throwable thr) {
            logger.info "Exception in $klassenname $name: $thr.message"
            throw thr
        }
    }
}
```

Diese Basisimplementierung von `invokeMethod()` lässt sich einen Logger für das aktuelle Objekt geben und protokolliert darin den Aufruf und die Rückkehr aus der gewünschten Methode. Wenn eine Exception auftritt, wird diese abgefangen und auch entsprechend protokolliert, bevor sie erneut ausgelöst wird. Wir müssen in dieser Methode sorgfältig darauf achten, dass keine eigenen Instanzmethoden aufgerufen werden, da es sonst unausweichlich zu einer Endlosschleife kommt. Aus diesem Grund können wir uns auch den aktuellen Klassennamen nicht direkt über `getClass()` holen, sondern müssen damit die Metaklasse beauftragen.

Damit das Protokollobjekt zeigen kann, was es zu bieten hat, entfernen wir `invokeMethod()` aus `MeineKlasse` und leiten die Klasse jetzt von `ProtokollObjekt` ab.

```
class MeineKlasse extends ProtokollObjekt {
    def Inhalt
}
```

Wie das Ganze zusammen läuft, probieren wir wieder interaktiv aus.

```
groovy> mk = new MeineKlasse()
groovy> println mk.toString()
groovy> println mk.toString("Hallo")
06.02.2007 20:30:47 sun.reflect.NativeMethodAccessorImpl invoke0
```

```
INFO: Aufruf MeineKlasse.toString {}
06.02.2007 20:30:47 sun.reflect.NativeMethodAccessorImpl invoke0
INFO: Ergebnis MeineKlasse.toString: MeineKlasse@1c80b01
MeineKlasse@1c80b01
06.02.2007 20:30:47 sun.reflect.NativeMethodAccessorImpl invoke0
INFO: Aufruf MeineKlasse.toString {"Hallo"}
06.02.2007 20:30:47 sun.reflect.NativeMethodAccessorImpl invoke0
INFO: Exception in MeineKlasse toString: No signature of method: MeineKlasse.toString()
is applicable for argument types: (java.lang.String) values: {"Hallo"}
06.02.2007 20:30:47 sun.reflect.NativeMethodAccessorImpl invoke0
INFO: Aufruf MeineKlasse.toString {"Hallo"}
06.02.2007 20:30:47 sun.reflect.NativeMethodAccessorImpl invoke0
INFO: Exception in MeineKlasse toString: No signature of method: MeineKlasse.toString()
is applicable for argument types: (java.lang.String) values: {"Hallo"}
Caught: groovy.lang.MissingMethodException: No signature of method: MeineKlasse.
toString() is applicable for argument types: (java.lang.String) values: {"Hallo"}
        at ProtokollObjekt.invokeMethod(LoggerBeispiel.groovy:10)
        at LoggerBeispiel.run(LoggerBeispiel.groovy:31)
        at LoggerBeispiel.main(LoggerBeispiel.groovy)
```

Zunächst erzeugen wir ein Objekt von `MeineKlasse` und nehmen dann zwei Methodenaufrufe vor. Der erste wird ausgeführt, der zweite jedoch nicht, weil die Methode mit einem Argument nicht definiert ist. Sie sehen, dass die Aufrufe und Ergebnisse bzw. Fehler brav protokolliert werden. Der fehlerhafte Aufruf erfolgt sogar zweimal; das liegt daran, dass der Groovy-`Invoker` mehrere Versuche unternimmt, eine passende Methode zu finden.

Im Ernstfall würden Sie die Meldungen von `ProtokollObjekt` lieber mit `logger.fine()` als mit `logger.info()` ausführen. Es empfiehlt sich aber nicht, mit `logger.finer()` oder den vordefinierten Logger-Methoden `entering()`, `exiting()` und `throwing()` zu arbeiten. Andernfalls bekommen Sie das Problem, dass Sie den Log-Handler auf die Stufe `FINER` setzen müssen und gleich eine Fülle von Meldungen zu sehen bekommen, die Groovy beim Aufruf jeder Methode generiert.

An dem `ProtokollObjekt` können Sie gut erkennen, wie Sie mithilfe der `invokeMethod()`-Methode und dem Interface `GroovyInterceptable` Querschnittsfunktionalitäten implementieren können. Allerdings gibt es hier zwei kleine Probleme, die wir Ihnen nicht vorenthalten wollen: Erstens scheitert der Ansatz, sobald Sie eine Klasse von einer anderen Klasse erben lassen wollen, beispielsweise einer Standard-Java-Klasse, die nicht von `ProtokollObjekt` abgeleitet ist, da Groovy genau wie Java keine Mehrfachvererbung kennt. Zweitens können Sie keine Aufrufe von statischen Methoden und Konstruktoren abfangen, denn da haben Sie keine Instanz in der Hand, deren `invokeMethod()` aufgerufen werden könnte. Weiter unten werden Sie im Abschnitt »Das Meta-Objekt-Protokoll« sehen, wie eine Lösung aussehen kann, die diese Nachteile nicht hat, dafür aber weitaus undurchsichtiger ist.

Dynamische Properties

Beim Übersetzen einer Groovy-Klasse erzeugt der Compiler auch zwei Methoden `getProperty()` und `setProperty()`, mit denen Properties des Objekts anhand ihres Namens abgerufen werden können. Genau wie `invokeMethod()` leiten sie die Aufrufe an die Metaklasse des Objekts weiter, die im Normalfall versucht, die korrespondierenden Getter oder Setter ausfindig zu machen. Im Unterschied zu `invokeMethod()` werden die Property-Methoden aber von der Laufzeitumgebung immer aufgerufen und nicht nur, wenn das Marker-Interface gesetzt ist. Sie können diese Methoden ähnlich einsetzen wie die optionalen Methoden `get()` und `set()`, die wir in Kapitel 3 behandelt haben, um mit dynamischen Properties zu arbeiten. Da `getProperty()` und `setProperty()` aber Vorrang vor der standardmäßigen Behandlung von Properties haben, brauchen Sie hier keine Sorge zu haben, dass es Namenskonflikte mit zufällig vorhandenen Gettern oder Settern gleichen Namens gibt.

Die beiden generischen Property-Zugriffsmethoden sind besonders praktisch, wenn Sie eine Wrapper-Klasse zum Kapseln einer vorhandenen Java-API bauen möchten, die nicht den Regeln für JavaBeans folgt, also keine standardmäßigen Getter und Setter hat, oder die dynamische Elemente hat, aber keine Map implementiert.

Ein Beispiel für eine solche API findet man in JDBC. Das Interface `java.sql.ResultSet` wird von den Herstellern von JDBC-Treibern mit eigenen Klassen implementiert, die Ergebnismengen zu SQL-Abfragen liefern. Dieses Interface definiert eine Vielzahl von Methoden zum Navigieren von Zeile zu Zeile und für den Zugriff auf die Spalten, die recht umständlich zu verwenden sind. Unsere simple Klasse `ResultSetWrapper` hält eine Referenz auf ein `ResultSet`-Objekt und leitet einfach alle lesenden Property-Zugriffe an diese weiter.

```
import java.sql.*

class ResultSetWrapper {
    ResultSet data      // Aktuelles ResultSet

    // Konstruktor übernimmt ein ResultSet
    def ResultSetWrapper(ResultSet resultSet) {
        data = resultSet
    }

    // Properties sind Spalten des ResultSet
    Object getProperty(String property) {
        data.getObject(property);
    }
}
```

Die Klasse definiert einen Konstruktor, dem eine `ResultSet`-Instanz übergeben werden muss, und implementiert die `GroovyObject`-Methode `getProperty()` derart, dass alle Aufrufe an die `getObject()`-Methode des `ResultSet` weitergeleitet werden. Diese Methode ist

intelligent genug, alle Datenbankfelder in äquivalente Java-Objekte umzuwandeln, sodass die Ergebnisse der Property-Abfragen automatisch immer den richtigen Typ haben.

Angenommen, wir hätten ein `ResultSet` namens `resultSet` aus einer Datenbankabfrage mit den Spalten `vorname` und `zuname`, dann würden wir es normalerweise folgendermaßen ausgeben:

```
while (resultSet.next()) {
    println "${resultSet.getObject('vorname')} {resultSet.getObject('zuname')}"
}
```

Mit unserer Wrapper-Klasse ist dies schon mal etwas übersichtlicher, da wir auf die Tabellenspalten wie auf Properties zugreifen können.

```
kapsel = new ResultSetWrapper(resultSet)
while (data.next() {
    println "$data.vorname $data.zuname"
}
```

Dieses Beispiel mag noch nicht allzu beeindruckend sein; im folgenden Abschnitt wollen wir jedoch zeigen, wie Sie mit wenigen Programmzeilen eine komplette Kapsel erstellen können, die einen bequemen Zugriff nach Groovy-Art auf alle Möglichkeiten des `ResultSet` bietet.

Projekt: eine kompletter Wrapper für ResultSet

Wir wollen die JDBC-Ergebnismenge hier als Beispiel für eine ältere Java-API betrachten, die sich nicht an die in Java inzwischen verbreiteten Muster hält. Sie ist nur relativ umständlich zu verwenden, und man muss sich erst mal orientieren, welche Methoden für welchen Zweck vorgesehen sind. Dazu bauen wir eine Groovy-Klasse, die folgende Eigenschaften hat:

- Sie kapselt die Klasse `ResultSet` vollständig, macht also deren Funktionalität komplett verfügbar, ohne dass ein Durchgriff des Anwenders auf das gekapselte Objekt erforderlich ist.
- Sie ermöglicht den lesenden Zugriff auf die Spalten einer Abfrage wie auf Properties (dies kennen Sie schon aus dem vorigen Abschnitt).
- Sie implementiert das Interface `java.util.Iterator`, sodass sie in `for`-Schleifen durchlaufen werden kann.
- Sie verfügt über eine Schleifenmethode `eachRow()`, die für jede Ergebniszeile die übergebene Closure aufruft.

Da die Groovy-Laufzeitbibliothek bereits eine umfangreiche und ausgefeilte Lösung für den Zugriff auf Datenbanken enthält (siehe Kapitel 6), ist es nicht besonders sinnvoll, eine solche Lösung selbst zu programmieren. Das Beispiel soll Ihnen in erster Linie zeigen, welche Möglichkeiten Groovy zum Kapseln solcher vorhandenen APIs zur Verfügung stellt und wie Sie dabei vorgehen können.

Die in Beispiel 7-1 erweiterte Version der Klasse `ResultSetWrapper` enthält bereits alles, was zur Lösung der gestellten Aufgabe erforderlich ist. Die Erläuterung folgt anschließend.

Beispiel 7-1: Die Klasse ResultSetWrapper

```
import java.sql.*

class ResultSetWrapper implements Iterator {

    private ResultSet data    // Aktuelles ResultSet

    // Konstruktor übernimmt ein ResultSet
    def ResultSetWrapper(ResultSet resultSet) {
        data = resultSet
    }

    // Properties sind Spalten des ResultSet
    Object getProperty(String property) {
        data.getObject(property)
    }

    // Setzen von Properties nicht implementiert
    void setProperty(String property, Object value) {
        throw new UnsupportedOperationException()
    }

    // Implementierung von next() aus Iterator
    // Schaltet zur nächsten Zeile
    def next() {
        if (data.next()) return this
        throw new NoSuchElementException()
    }

    // Implementierung von hasNext() aus Iterator
    // Prüft, ob es weitere Zeilen gibt
    boolean hasNext() {
        return ! data.isLast()
    }

    // Keine Implementierung von remove() aus Iterator
    void remove() {
        throw new UnsupportedOperationException()
    }

    // Methode zum Durchlaufen des ResultSet mit einer
    // Closure
    void eachRow (Closure closure) {
        if (! data.isBeforeFirst()) {
            data.beforeFirst()
        }
        while (data.next()) {
            closure(this)
        }
```

Beispiel 7-1: Die Klasse ResultSetWrapper (Fortsetzung)

```
    }

    // Leitet alle Aufrufe von undefinierten Methoden
    // an das ResultSet weiter
    def invokeMethod (String methodName, Object args) {
        return data."$methodName"(*args)
    }
}
```

Die Methode `getProperty()` kennen Sie bereits, sie lässt sich vom `ResultSet` den Wert geben, dessen Spaltenname dem Property-Namen entspricht. Das Pendant zum Schreiben `setProperty()` ist nur der Vollständigkeit halber implementiert und liefert eine Exception. Wir verzichten darauf, die standardmäßige `GroovyObject`-Methode `getProperties()` so zu überschreiben, dass diese alle Properties liefert, die auch die `getProperty()`-Methode findet. Das ist vielleicht nicht ganz konsistent, aber wir befinden uns damit im Einklang mit der Groovy-Standardbibliothek, in der diese Übereinstimmung ebenfalls nicht immer gegeben ist.

Die drei Methoden `next()`, `hasNext()` und `remove()` sind zur Implementierung des `Iterator`-Interface erforderlich. Sie bedienen sich der äquivalenten, aber anders definierten Methoden des `ResultSet`. Allerdings wird `remove()` für unsere Zwecke nicht benötigt und löst daher vorschriftsmäßig die `UnsupportedOperationException` aus. Die Implementierung von `next()` ist etwas unkonventionell, da die Methode einfach das `ResultSet` auf die nächste Zeile schaltet und den `ResultSetWrapper` selbst zurückgibt. Dies reicht aber aus, um `for`-Schleifen über den `ResultSetWrapper` zu ermöglichen.

Eine zweite Möglichkeit zum Durchlaufen der Daten bietet die Methode `eachRow()`, der man eine Closure übergibt, die für jede Ergebniszeile einmal aufgerufen wird. Als Argument erhält die Closure auch hier wieder den `ResultSetWrapper` selbst, dessen Spaltenwerte sich ja wie Properties abfragen lassen.

Schließlich ist noch die Methode `invokeMethod()` zu erwähnen, mit der wir die Anforderung erfüllen, dass das `ResultSet` vollständig gekapselt werden soll. Wie wir weiter oben gezeigt haben, leitet diese Methode alle Methodenaufrufe, zu denen es im `ResultSetWrapper` selbst keine passende Methode gibt, an den gekapselten `ResultSet` weiter. Und schon steht die gesamte Funktionalität des `ResultSet` auch dem Benutzer des `ResultSetWrapper` zur Verfügung, ohne auf das `ResultSet` selbst durchgreifen zu müssen. Daher können wir auch die Variable `data` hier ruhigen Gewissens auf `private` stellen – sie wird von außen nicht benötigt.[3]

3 Ehrlicherweise müssen wir zugeben, dass die Funktionalität des `ResultSet` doch nicht vollständig verfügbar ist: Die Methode `next()` wird vom `ResultSetWrapper` durch eine eigene Methode mit anderer Semantik verdeckt. In diesem Fall lässt es sich verschmerzen, da das verdeckte `next()` nicht mehr benötigt wird; das Beispiel erinnert uns aber daran, dass wir bei solchen Wrapper-Konstruktionen wie hier auf möglicherweise verdeckte Member des gekapselten Objekts achten müssen.

Um das Ganze zu testen, benötigen wir eine Datenbank, denn nur diese kann uns ein `ResultSet` liefern. Das ist nicht schwierig, wenn Sie sich einfach das aktuelle HSQLDB-Datenbanksystem von *http://hsqldb.org/* herunterladen und daraus die Bibliotheksdatei *hsqldb.jar* in das Verzeichnis *~/.groovy/lib* unterhalb Ihres Home-Verzeichnisses platzieren, so dass sie für Groovy im Klassenpfad verfügbar ist. Unter Windows XP lautet der vollständige Pfad also beispielsweise *C:\Dokumente und Einstellungen\krause\.groovy\lib\hsqldb.jar*).

Mit ein paar Zeilen Code öffnen wir eine Datenbankverbindung, legen eine neue In-Memory-Datenbank an, definieren in ihr eine neue Tabelle und füllen diese mit ein paar Daten. Dann lesen wir diese Daten einmal in einer `for`-Schleife und einmal über die `eachRow()`-Methode aus. Beispiel 7-2 zeigt dies anhand eines Testskripts.

Beispiel 7-2: Testskript ResultSetWrapperTest.groovy

```
import java.sql.*

// Datenbanktreiber laden (unter Java 6.0 nicht mehr nötig)
Class.forName 'org.hsqldb.jdbcDriver'
// Verbindung herstellen
conn = DriverManager.getConnection('jdbc:hsqldb:mem:aname', 'sa', '');
stmt = conn.createStatement()

// Zwei Closures zur Vereinfachung von Abfragen
execute = { stmt.execute it }     // Einfacher Befehl
query = { stmt.executeQuery it}   // Abfrage mit Ergebnis

// Tabelle anlegen und mit zwei Zeilen füllen
execute('CREATE TABLE daten (name VARCHAR, geboren DATE, punkte INTEGER)')
execute("INSERT INTO daten values ('Klaus Meier','1972-12-01',100)")
execute("INSERT INTO daten values ('Otto Müller','1966-02-28',99)")

// Erste Testschleife
println 'Test eachRow()'
rsw = new ResultSetWrapper(query('SELECT * FROM daten'))
rsw.eachRow { println "$it.name - $it.geboren - $it.punkte" }
rsw.close()

// Zweite Testschleife
println 'Test for-Schleife'
rsw = new ResultSetWrapper(query('SELECT * FROM daten'))
for (r in rsw) {
    println r.getString('name')+', '+r.getDate('geboren')+
                ', '+r.getInt('punkte')
}
rsw.close()

// Datenbankverbindung schließen
conn.close()
```

Wenn Sie dieses Skript im selben Verzeichnis wie *ResultSetWrapper.groovy* ablegen und es dann aufrufen, sollte das Ergebnis so aussehen:

```
> groovy ResultSetWrapper.groovy
Test eachRow()
Klaus Meier - 1972-12-01 - 100
Otto Müller - 1966-02-28 - 99
Test for-Schleife
Klaus Meier, 1972-12-01, 100
Otto Müller, 1966-02-28, 99
```

Beachten Sie die Aufrufe der Originalmethoden des `ResultSet` an unserer Klasse: `getString()`, `getDate()`, `getInt()` und `rsw.close()`. Allesamt sind sie nicht im `ResultSetWrapper` definiert. Dies ist auch nicht nötig, denn `invokeMethod()` leitet ja alle Methodenaufrufe, die das `ResultSetWrapper`-Objekt nicht bedienen kann, an das gekapselte Objekt weiter. Und so funktionieren die Methoden genau so, als würde man sie am `ResultSet` direkt aufrufen.

Die Klasse Proxy

Um solche Konstellationen noch etwas zu vereinfachen, bietet Groovy in seiner Laufzeitbibliothek eine Klasse namens `groovy.util.Proxy` an, in der die Weiterleitung der Methodenaufrufe an ein anderes Objekt schon eingebaut ist und von dem solche Klassen wie das obige `ResultSetWrapper` bequem abgeleitet werden können. Die folgende Klasse `ResultSetWrapperProxy` macht exakt dasselbe wie der obige `ResultSetWrapper`, ist aber auf Basis der `Proxy`-Klasse implementiert.

Beispiel 7-3: Die Klasse ResultSetWrapperProxy

```
import java.sql.*
import groovy.util.Proxy as GroovyProxy

class ResultSetWrapperProxy extends GroovyProxy {

    // Konstruktor übernimmt ein ResultSet und setzt
    // ihn als adaptee des Proxy
    def ResultSetWrapperProxy(ResultSet resultSet) {
        setAdaptee(resultSet)
    }

    // Properties sind Spalten des ResultSet
    Object getProperty(String property) {
        getAdaptee().getObject(property);
    }

    // Setzen von Properties nicht implementiert
    void setProperty(String property, Object value) {
        throw new UnsupportedOperationException()
    }
```

Beispiel 7-3: Die Klasse ResultSetWrapperProxy (Fortsetzung)

```
    // Implementierung von next() aus Iterator
    // Schaltet zur nächsten Zeile
    def next() {
        if (getAdaptee().next()) return this
        throw new NoSuchElementException()
    }

    // Implementierung von hasNext() aus Iterator
    // Prüft, ob es weitere Zeilen gibt
    boolean hasNext() {
        return ! getAdaptee().isLast()
    }

    // Keine Implementierung von remove() aus Iterator
    void remove() {
        throw new UnsupportedOperationException()
    }

    // Methode zum Durchlaufen des ResultSet mit einer
    // Closure
    void eachRow (Closure closure) {
        if (! getAdaptee().isBeforeFirst()) {
            getAdaptee().beforeFirst()
        }
        while (getAdaptee().next()) {
            closure(this)
        }
    }
}
```

Statt des Felds `data` benutzen wir hier eine in `Proxy` definierte Property namens `adaptee` als Referenz auf das `ResultSet`, an das die Methodenaufrufe automatisch weitergeleitet werden. Und die eigene Implementierung von `invokeMethod()` können wir einfach weglassen, weil deren Aufgabe vom `Proxy` übernommen wird. Wir müssen hier ein wenig aufpassen, dass wir `adaptee` nicht in der Property-Notation ansprechen, sondern immer ordentlich über die Getter und Setter, denn die Property-Zugriffe werden ja alle als Feldanfragen zum `ResultSet` geleitet, was hier natürlich nicht beabsichtigt ist.

Sie können diese Klasse ebenfalls mit dem Skript *ResultSetWrapperTest.groovy* ausprobieren, Sie müssen nur an einer Stelle `ResultSetWrapper` durch `ResultSetWrapperProxy` ersetzen. Das Ergebnis ist genau dasselbe.

Zwei Anmerkungen noch zur Implementierung der Klasse `ResultSetWrapperProxy`. Erstens ist die `import`-Anweisung für `groovy.util.Proxy` mit dem Alias `GroovyProxy` versehen. Eigentlich ist der ausdrückliche Import der Klassen im Package `groovy.util` nicht erforderlich, da es aber einen Namenskonflikt mit der ebenfalls implizit importierten Klasse `java.net.Proxy` geben kann, müssen wir angeben, was wir meinen. Zweitens fällt Ihnen

vielleicht auf, dass die `implements`-Klausel für `Iterator` fehlt, die wir oben beim `ResultSetWrapper` noch angegeben haben. Tatsächlich hätten wir sie oben auch nicht benötigt, denn für die `for`-Schleife in Groovy ist allein wichtig, dass die Methoden `next()` und `hasNext()` vorhanden sind, nicht aber die formale Implementierung des Interface. Ein Beispiel dafür, dass Interfaces für Groovy eine weit weniger wichtige Rolle spielen als für Java.

Das Meta-Objekt-Protokoll

Hinter dem Begriff Meta-Objekt-Protokoll (MOP) verbirgt sich in Groovy der Mechanismus, den die Sprache zur externen Modifikation des Verhaltens von Objekten zur Verfügung stellt. Wir werden hier versuchen, seine prinzipielle Wirkungsweise darzustellen, ohne uns allzu sehr in seinen Tiefen zu verlieren.

Wie oben beschrieben, wird in Groovy die gesamte Kommunikation zwischen Objekten über Metaklassen abgewickelt. Der Aufruf einer anderen Methode – egal ob im eigenen oder in einem fremden Objekt – erfolgt immer vermittelt über die für das Zielobjekt zuständige Metaklasseninstanz. Diese ist in der Regel bei der `MetaClassRegistry` für die Klasse des Zielobjekts eingetragen. Unter bestimmten Voraussetzungen kann aber davon abweichend einem spezifischen Zielobjekt individuell eine eigene Metaklasseninstanz mit einem abweichenden Verhalten zugeordnet werden.

Jede Metaklasseninstanz »kennt« die Klasse, für die sie zuständig ist. Die wichtigsten Methoden, die sie zur Verfügung stellt, sind folgende:

- `void initialize()` schließt den Initialisierungsprozess der Instanz ab und muss aufgerufen werden, bevor eine der folgenden Methoden verwendet wird. Auf die Initialisierung selbst wollen wir hier nicht näher eingehen.
- `Object invokeConstructor(Object[] args)` erzeugt eine neue Instanz der betreffenden Klasse.
- `Object invokeMethod (Object object, String method, Object[] args)` ruft bei dem übergebenen Objekt, das eine Instanz der betreffenden Klasse sein muss, die Methode mit dem angegebenen Namen und den Argumenten auf und liefert das Ergebnis des Aufrufs zurück.
- `Object getProperty (Object object, String property)` liest bei dem übergebenen Objekt, das eine Instanz der betreffenden Klasse sein muss, die Property mit dem angegebenen Namen aus und liefert deren Wert zurück.
- `void setProperty (Object object, String property, Object wert)` schreibt bei dem übergebenen Objekt, das eine Instanz der betreffenden Klasse sein muss, den übergebenen Wert in die Property mit dem angegebenen Namen.

Wenn Sie eine eigene Metaklassenimplementierung haben, können Sie diese folgendermaßen ins Spiel bringen.

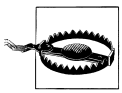
Soll die Metaklasse einer gesamten Klasse zugeordnet werden, können Sie sie direkt bei der `MetaClassRegistry` registrieren. Dabei handelt es sich um eine Singleton-Instanz, die Sie sich am besten über einen statischen Aufruf von `groovy.lang.GroovySystem` holen. Die folgende Anweisung registriert eine Metaklasse für die Klasse `java.lang.Integer`.

```
GroovySystem.metaClassRegistry.setMetaClass (Integer.class, new MeineMetaClass())
```

Eine andere Möglichkeit besteht darin, eine Klasse einfach auf eine bestimmte Art zu benennen. Bilden Sie den Namen der Metaklasse so, dass Sie vor dem Package-Pfad der Zielklasse `groovy.runtime.metaclass.` setzen und nach dem Namen der Zielklasse `MetaClass` anfügen. Wenn wir also die Metaklasse für `Integer` folgendermaßen benennen, brauchen wir sie nicht explizit anzumelden, da sie von der `MetaClassRegistry` bei einem Zugriff auf ein `Integer`-Objekt automatisch gefunden wird.

```
groovy.runtime.metaclass.java.lang.IntegerMetaClass
```

Beachten Sie, dass Metaklassen immer genau für die Instanzen der einen Klasse gelten, für die sie registriert sind, und nicht etwa für Instanzen davon abgeleiteter Klassen. Wenn Sie also beispielsweise eine Metaklasse für die Klasse `java.util.HashMap` registrieren, gilt sie nicht für Objekte des davon abgeleiteten Typs `java.util.LinkedHashMap`. Demzufolge ist es auch völlig sinnlos, Metaklassen für abstrakte Klassen (`java.util.AbstractMap`) oder Interfaces (`java.util.Map`) zu registrieren.



Indem Sie einer Klasse oder einem einzelnen Objekt eine spezielle Metaklasse zuordnen, können Sie massiv in das Verhalten der betreffenden Klasse oder des Objekts eingreifen. Die von Groovy standardmäßig verwendete `MetaClassImpl` ist ein recht komplexes Gebilde, das durch eigene Kreationen zu ersetzen zumindest bei den Standardklassen wegen möglicherweise auftretender äußerst undurchsichtiger Folgewirkungen generell nicht zu empfehlen ist.

Die Groovy-Standardbibliothek bietet einige spezielle Metaklassenimplementierungen, die anstelle von `MetaClassImpl` eingesetzt werden können und zusätzliche Möglichkeiten bieten. An zwei Beispielen werden wir zeigen, wie Sie mit ausgetauschten Metaklassen arbeiten können. Auch hier ist aber eine gewisse Vorsicht wegen möglicherweise unerwarteter Nebenwirkungen durchaus am Platze.

Die DelegatingMetaClass

Die Klasse `java.lang.DelegatingMetaClass` tut nichts weiter, als alle Methodenaufrufe an eine andere Metaklasse weiterzuleiten. Man kann ihr die Metaklasse, an die delegiert werden soll, direkt übergeben, man kann aber auch die Klasse übergeben, für die sie zuständig sein soll; in diesem Fall erzeugt sie sich selbst eine `MetaClassImpl`-Instanz und leitet an diese weiter.

Für sich gesehen, bewirkt der Einsatz der `DelegatingMetaClass` also wenig; man kann aber eine eigene Metaklasse von ihr ableiten und hat dann die Möglichkeit, in das

Geschehen einzugreifen, indem man eine oder mehrere der oben genannten MetaClass-Methoden überschreibt.

Beim Behandeln des Interface GroovyInterceptable haben wir eine abstrakte Basisklasse ProtokollObjekt als Beispiel entwickelt, das die automatische Protokollierung bei davon abgeleiteten Klassen ermöglicht. Dabei haben wir auch bemerkt, dass die Notwendigkeit, von dieser Klasse abzuleiten, wegen der resultierenden Beschränkungen bezüglich der Klassenhierarchie nur begrenzt anwendbar ist. Solche Querschnittsaufgaben wie Protokollierung können Sie mit Metaklassen völlig transparent implementieren. Die folgende Klasse ProtokollDelegatingMetaClass funktioniert analog zur erwähnten Klasse ProtokollObjekt. Die Klasse, für die sie zuständig sein soll, wird ihr als Konstruktorargument übergeben. Auch hier geschieht nichts weiter, als dass der Aufruf von invokeMethod() abgefangen wird, die hier eine andere Signatur als die gleichnamige Methode des GroovyObject hat, da das Objekt, an dem die Methode ausgeführt werden soll, als Argument übergeben werden muss. Zwischen zwei Protokolleinträgen wird der Methodenaufruf an die DelegatingMetaClass weitergegeben, von der wir abgeleitet haben, und diese reicht ihn an die Metaklasse weiter, die ursprünglich zuständig war.

```
import java.util.logging.*

class ProtokollDelegatingMetaClass extends DelegatingMetaClass {

    Logger logger
    String klassenname

    def ProtokollDelegatingMetaClass(Class zurKlasse) {
        super(zurKlasse)
        klassenname = zurKlasse.name
        logger = Logger.getLogger (klassenname)
        initialize()
    }

    def invokeMethod (Object object, String name, Object[] args) {
        logger.info "Aufruf $klassenname.$name $args"
        try {
            def ergebnis = super.invokeMethod(object,name,args)
            logger.info "Ergebnis $klassenname.$name: $ergebnis"
            return ergebnis
        } catch (Throwable thr) {
            logger.info "Exception in $klassenname $name: $thr.message"
            throw thr
        }
    }
}
```

Beachten Sie, dass der Konstruktor mit einem Aufruf von initialize() endet; bevor diese Methode ausgeführt worden ist, kann eine Metaklasse grundsätzlich nicht angewendet werden.

Zum Ausprobieren schreiben wir ein kurzes Skript namens *TestProtokollDelegatingMetaClass.groovy*, das noch mal eine Mini-Klasse namens `MeineKlasse` zum Ausprobieren definiert, unsere Metaklasse bei der `MetaClassRegistry` für `MeineKlasse` anmeldet und schließlich einige Methodenaufrufe an einer Instanz von `MeineKlasse` durchführt.

```
// Eine Miniaturklasse als Versuchstier
class MeineKlasse {
    def inhalt
}

// Instanz der Metaklasse erzeugen und mit der Zielklasse initialisieren
metaKlasse = new ProtokollMetaClass(MeineKlasse)
// Metaklasse für die Zielklasse registrieren
org.codehaus.groovy.runtime.InvokerHelper.instance.
          metaRegistry.setMetaClass(MeineKlasse, metaKlasse )

// Instanz der Versuchsklasse anlegen
// und einige Methodenaufrufe durchführen
mk = new MeineKlasse()
println mk.toString()
mk.setInhalt("Wichtige Daten")
println mk.getInhalt()
```

Schließlich führen wir das Skript in der Konsole aus, und neben den `println()`-Ausgaben erscheinen die erwarteten Protokollausgaben.

```
> groovy TestProtokollDelegatingMetaClass
MeineKlasse@641e9a
Wichtige Daten
21.02.2007 21:06:57 gjdk.java.util.logging.Logger_GroovyReflector invoke
INFO: Aufruf MeineKlasse.toString {}
21.02.2007 21:06:57 gjdk.java.util.logging.Logger_GroovyReflector invoke
INFO: Ergebnis MeineKlasse.toString: MeineKlasse@641e9a
21.02.2007 21:06:57 gjdk.java.util.logging.Logger_GroovyReflector invoke
INFO: Aufruf MeineKlasse.setInhalt {"Wichtige Daten"}
21.02.2007 21:06:57 gjdk.java.util.logging.Logger_GroovyReflector invoke
INFO: Ergebnis MeineKlasse.setInhalt: null
21.02.2007 21:06:57 gjdk.java.util.logging.Logger_GroovyReflector invoke
INFO: Aufruf MeineKlasse.getInhalt {}
21.02.2007 21:06:57 gjdk.java.util.logging.Logger_GroovyReflector invoke
INFO: Ergebnis MeineKlasse.getInhalt: Wichtige Daten
```

Wir haben unsere Metaklasse hier explizit mit `setMetaClass()` bei der `MetaClassRegistry` für die ganze Klasse `MeineKlasse` angemeldet, damit war sie automatisch auch für die eine konkrete Instanz zuständig. Alternativ hätten wir sie einfach folgendermaßen nennen können:

```
groovy.runtime.metaclass.MeineKlasseMetaClass
```

Die bloße Existenz dieser Klasse im Klassenpfad hätte genügt, und eine ausdrückliche Anmeldung wäre nicht mehr nötig gewesen. In diesem Fall wäre es natürlich nicht beson-

ders sinnvoll, denn die ProtokollDelegatingMetaClass ist ja prinzipiell für viele Zielklassen anwendbar, und man würde wohl nicht für jeden Anwendungsfall eine Kopie mit eigenem Namen anlegen wollen.

Schließlich hätten wir sie auch der einzigen Instanz von MeineKlasse individuell zuordnen können:

```
mk = new MeineKlasse()
mk.metaClass = new ProtokollDelegatingMetaClass(MeineKlasse)
```

Das Ergebnis des Skriptaufrufs wäre in jedem Fall dasselbe.

Zusammenfassend kann man sagen, dass sich mit dem Meta-Objekt-Protokoll ähnliche Ziele umsetzen lassen, wie sie auch von der Aspektorientierten Programmierung angestrebt werden.

Die ProxyMetaClass

Während die eben vorgestellte DelegatingMetaClass eine Art Schweizer Messer für diverse Modifikationen der Kommunikation zwischen Objekten darstellt, ist die ebenfalls in der Groovy-Laufzeitbibliothek enthaltene ProxyMetaClass ganz auf das Abfangen von Methodenaufrufen abgestimmt. Für einen konkreten Verwendungszweck passen wir sie nicht durch Überschreiben von Methoden an, sondern mithilfe eines speziellen Objekts, das das Interface groovy.lang.Interceptor implementiert und der DelegatingMetaClass übergeben wird.

Das Interceptor-Interface sieht folgendermaßen aus:

```
// Java
package java.lang;
public interface Interceptor {
     Object beforeInvoke(Object object, String methodName, Object[] arguments);
    Object afterInvoke(Object object, String methodName, Object[] arguments, Object
result);
    boolean doInvoke();
}
```

Die drei Methoden, die das Interface definiert, sind folgendermaßen zu verstehen:

- Die Methode beforeInvoke() wird jedes Mal vor jedem Aufruf einer Methode im Zielobjekt aufgerufen und erhält eine Referenz auf das Zielobjekt, den Methodennamen und die Aufrufargumente. Der Rückgabewert dieser Methode spielt nur dann eine Rolle, wenn der nachfolgende Aufruf von doInvoke() das Ergebnis false liefert.
- Das Ergebnis von doInvoke() entscheidet darüber, ob der Methodenaufruf überhaupt an die eigentlich zuständige Metaklasse und damit an das Zielobjekt weitergeleitet werden soll. Ist dies nicht der Fall, wird der Rückgabewert von beforeInvoke() anstelle des Methodenaufrufergebnisses weiterverwendet.

- In jedem Fall wird abschließend die Methode `afterInvoke()` aufgerufen. Sie hat die gleichen Parameter wie `beforeInvoke()` und zusätzlich noch das Ergebnis des Methodenaufrufs (das auch von `beforeInvoke()` stammen kann). Der Rückgabewert von `afterInvoke()` wird letztendlich an den ursprünglichen Aufrufer zurückgegeben.

Wenn wir diese Variante ausprobieren wollen, müssen wir also eine entsprechende Klasse erstellen, die `Interceptor` implementiert. Wir nennen sie `ProtokollInterceptor`:

```
import java.util.logging.*

class ProtokollInterceptor implements Interceptor {

    Logger logger
    String klassenname

    // Konstruktor merkt sich nur den Klassennamen
    // und legt einen Logger an
    def ProtokollInterceptor (Class klasse) {
        klassenname = klasse.name
        logger = Logger.getLogger (klassenname)
    }

    // Methodenaufruf protokollieren
    def beforeInvoke(objekt, String name, Object[] args) {
        logger.info "Aufruf $klassenname.$name $args"
    }

    // Methodenausgang protokollieren
    def afterInvoke(objekt, String name, Object[] args, ergebnis) {
        logger.info "Ergebnis $klassenname.$name: $ergebnis"
        ergebnis
    }

    // Aufruf soll immer ausgeführt werden
    boolean doInvoke() {
        true
    }
}
```

Zum Ausprobieren benutzen wir das folgende Skript *TestProtokollInterceptor.groovy*. Es verwendet zum Instantiieren der Metaklasse die von `ProxyMetaClass` zur Verfügung gestellte statische Factory-Methode `getInstance()`. Die Registrierung wird übrigens von dieser Metaklasse selbst übernommen; dafür gibt es eine Methode `use()`, der Sie eine Closure übergeben können. Sie registriert sich selbst bei der `MetaClassRegistry` für ihre Zielklasse, führt die Closure aus und deregistriert sich anschließend. Auf diese Weise kann sie sehr gezielt eingesetzt werden.

```
class MeineKlasse {
    def inhalt
}
```

```
metaKlasse = ProxyMetaClass.getInstance(MeineKlasse)
metaKlasse.interceptor = new ProtokollInterceptor(MeineKlasse)

metaKlasse.use {
    mk = new MeineKlasse()
    println mk.toString()
    mk.setInhalt("Wichtige Daten")
    println mk.getInhalt()
}
```

Wenn Sie dieses Skript starten, sieht das Ergebnis genau so aus wie oben bei dem Skript *TestProtokollDelegatingMetaClass.groovy*. Man kann die `use()`-Methode übrigens auch ganz gezielt auf eine einzige Instanz der Zielklasse anwenden, indem man diese zusätzlich als erstes Argument angibt:

```
mk = new MeineKlasse()
metaKlasse.use (mk) {
    ...
}
```

Die Groovy-Laufzeitbibliothek liefert übrigens schon zwei Standardimplementierungen des `Interceptor`-Interface mit, mit denen die häufigsten Anwendungsfälle der `ProxyMetaClass` abgedeckt sind. Sie sind sehr einfach anzuwenden, deshalb genügt hier eine kurze Beschreibung dieser Klassen.

- Die Klasse `groovy.lang.TracingInterceptor` definiert einen Interzeptor, der ähnlich wie unser obiges Beispiel jeden Methodenaufruf protokolliert, allerdings werden die Protokollzeilen in einen `Writer` geschrieben, den Sie über die Property `writer` setzen können. Standardmäßig protokolliert der `TracingInterceptor` auf der Konsole.
- Der `groovy.lang.BenchmarkInterceptor` dient der Erstellung von Metriken. Er zählt einfach jeden Methodenaufruf mit. Mithilfe einer von ihm zur Verfügung gestellten Methode `statistic()` können Sie dann am Ende eine Auflistung der Ergebnisse abholen.

Die ExpandoMetaClass

Eine neue Errungenschaft der Groovy-Version 1.1 ist die aus dem Grails-Projekt übernommene `ExpandoMetaClass`. Sie ermöglicht es auf geradezu faszinierende Art und Weise, vorhandene Klassen mit zusätzlichen Methoden zu versehen, ohne auf vergleichsweise umständliche Konstruktionen wie die Kategorienklassen zurückgreifen zu müssen. Sie weisen der `ExpandoMetaClass` einfach eine Closure als virtuelle Property zu, und schon steht die Closure der zugeordneten Klasse als neue Methode zur Verfügung.

Haben Sie nicht auch schon die zwei String-Methoden `ltrim()` und `rtrim()` vermisst, mit denen man an einem String die Leerzeichen links bzw. rechts entfernen kann? Selbst als vordefinierte Groovy-Methoden gibt es sie noch nicht – also machen wir sie uns einfach

selbst. Beispiel 7-4 zeigt ein Skript namens *ExpandoMetaClassSkript.groovy*, in dem diese beiden Methoden verfügbar gemacht werden.

Beispiel 7-4: Das Programm ExpandoMetaClassSkript.groovy.

```
emc = new ExpandoMetaClass (String)
emc.rtrim = {
    def str = delegate
    while (str && str[-1]==' ') str = str[0..-2]
    str
}
emc.ltrim = {
    def str = delegate
    while (str && str[0]==' ') str = str[1..-1]
    str
}
emc.initialize()

println '|'+  abc  '.ltrim()+'|'
println '|'+  abc  '.rtrim()+'|'
```

Gleich am Anfang sehen Sie, dass Sie die `ExpandoMetaClass` nicht explizit registrieren müssen, vielmehr genügt es, die Zielklasse als Konstruktorargument anzugeben. Sie dürfen nicht vergessen, die Methode `initialize()` aufzurufen, nachdem alle Methoden-Closures zugewiesen sind und bevor die zugeordnete Klasse verwendet wird.

Innerhalb der Closure kann mithilfe der Property `delegate` auf das Objekt zugegriffen werden, an dem die Methode auszuführen ist. Genau dies tun die beiden Closures, dann schneiden sie in einer Schleife links bzw. rechts die Leerzeichen ab und übergeben den Rest als Ergebnis. Wenn Sie das Skript ausführen, finden wir tatsächlich eine links und eine rechts gekürzte Version des übergebenen Strings:

```
> groovy ExpandoMetaClassSkript
|abc  |
|  abc|
```

Groovy erhielt auch eine neue vordefinierte Methode namens `getMetaClass()`, die auf Klassen anwendbar ist. Sie ordnet der jeweiligen Klasse eine eigene `ExpandoMetaClass` zu, sofern dies nicht schon geschehen ist, und gibt diese zurück. Die Definition der Methoden-Closures im *ExpandoMetaClassSkript.groovy* hätte also auch in dieser Art formuliert werden können:

```
String.metaClass.ltrim = { ... }
```

Die Initialisierung der Metaklasse ist in diesem Fall nicht erforderlich.

Anzumerken ist, dass die hinzugefügten Methoden nur effektiv sind, wenn es keine entsprechende direkt implementierte Methode gibt. Sie können also die Originalmethoden des jeweiligen Typs nicht überschreiben. Insofern ist die `ExpandoMetaClass` etwas weniger

gefährlich als die Kategorienklassen, deren Methoden Vorrang vor den in der Klasse implementierten Methoden haben.

Die `ExpandoMetaClass` bietet noch einige weitere Möglichkeiten, die wir hier kurz zusammenfassen wollen:

- Wenn Sie eine Methode hinzufügen und dabei sichergehen wollen, dass keine vorhandene Methode überschrieben wird, benutzen Sie den <<-Operator anstelle des Gleichheitszeichens bei der Zuweisung der Closure:

  ```
  emc.ltrim << { ... }
  ```

- Statische Methoden können Sie zuordnen, indem Sie diese nicht der Metaklasse selbst, sondern der Property `static` zuordnen. Da `static` ein reserviertes Wort ist, muss es in Anführungszeichen gesetzt werden.

  ```
  emc.'static'.statischeMethode = { ... }
  ```

- Analog können Sie auch einen virtuellen Konstruktor definieren, indem Sie die Closure der Property `constructor` zuweisen:

  ```
  emc.constructor = {arg1, arg2 -> ... }
  ```

  Eine Konstruktor-Closure muss eine neue Instanz der jeweiligen Klasse liefern. Diese Möglichkeit eignet sich besonders gut dazu, statische Factory-Methoden als Konstruktoren der zu erzeugenden Klasse zu tarnen. Achten Sie aber darauf, dass Sie keine Endlosschleife programmieren, wenn Sie versuchen, den echten Konstruktor der Klasse aufzurufen.

- Sie können auch einer Klasse eine zusätzliche Property hinzufügen. Setzen Sie dazu einfach deren Vorgabewert in der `ExpandoMetaClass`:

  ```
  String.metaClass.meineProperty = 42
  ```

  Dann hat beispielsweise der Ausdruck `"abc".meineProperty` den Wert 42. Der Property-Wert kann auf Instanzebene geändert werden, er ist aber Thread-lokal und nur von begrenzter Haltbarkeit.

- Überladen von Methoden ist möglich. Sie können der `ExpandoMetaClass` durchaus mehrere Closures unter dem gleichen Namen zuweisen, die sich in Anzahl oder Typ der Parameter unterscheiden.

- Standardmäßig wirkt die `ExpandoMetaClass` nur auf Instanzen der Klasse ein, der sie direkt zugeordnet ist, und nicht auf Instanzen abgeleiteter Klassen. Wenn Sie dies ändern möchten, müssen Sie zu Beginn Ihres Programms folgenden Aufruf tätigen:

  ```
  ExpandoMetaClass.enableGlobally()
  ```

  Nun können Sie die `ExpandoMetaClass` auch abstrakten Klassen und Interfaces zuordnen, und sie wirkt sich auf alle Klassen auf, die von diesen abgeleitet sind bzw. sie implementieren – sofern die betreffenden Methoden dort nicht überschrieben sind.

Sie haben nun die wichtigsten dynamischen Möglichkeiten der Sprache Groovy kennengelernt. Damit bekommen Sie einige sehr mächtige Werkzeuge in die Hand, die Ihnen helfen können, Programme sehr effektiv zu schreiben – deren Gefahren wegen massiver Nebeneffekte aber unübersehbar sind. Zurückhaltung, sauberes Design, extensive Dokumentation und systematische Tests sind noch wichtiger als sonst, wenn Ihre Programme auch dann noch wartbar bleiben sollen, wenn Sie die Vorteile der dynamischen Programmierung nutzen.

KAPITEL 8

Groovy und Java integrieren

Wie Sie Java-Klassen in Groovy integrieren, haben Sie in den vorangehenden Kapiteln schon in vielfältiger Weise gesehen. Es kommt in jedem Groovy-Programm sehr häufig vor, dass auf Klassen des Java Development Kit zugegriffen werden muss, und in vielen Fällen kann es sinnvoll sein, andere konventionell mit Java programmierte Programmteile einzubinden. So könnten Sie etwa bereits vorhandene Klassen weiterverwenden wollen, oder Ihre Softwarearchitektur sieht eine systematische Trennung zwischen Java- und Groovy-Programmen vor. Schließlich kann es sich auch aus Performancegründen anbieten, einzelne kritische Routinen in Java zu implementieren, um die sehr aufwendige Auflösung von Feldern und Methoden zur Laufzeit einzusparen.

Das Vorgehen dabei unterscheidet sich nicht wesentlich von der Benutzung von Java-Klassen aus Java selbst. Anders sieht es aus, wenn Sie den umgekehrten Weg beschreiten wollen: Groovy-Klassen oder -Skripte in Java-Programme integrieren. Auch dafür kann es verschiedene Gründe geben: Zum einen möchten Sie vielleicht in einigen Programmen einfach die mächtigeren Möglichkeiten und die arbeitssparende Syntax der Sprache Groovy nutzen, um einen Teil Ihrer Anwendung mit geringerem Aufwand, aber ansonsten ganz normal in Form von kompilierten Klassen zu implementieren.

Sie könnten aber auch die dynamischen Möglichkeiten von Groovy-Programmen nutzen wollen, die erst zur Laufzeit kompiliert und eingebunden werden. Dies ist insbesondere dann sinnvoll, wenn Teile des Programms anpassbar sein sollen, ohne dass die ganze Anwendung neu gebaut und ausgeliefert werden muss, oder wenn sogar die Anwender in die Lage versetzt werden sollen, Formeln oder Skripte selbst zu schreiben und im Rahmen der Anwendung ausführen zu lassen.

In diesem Abschnitt zeigen wir zunächst, wie Groovy-Klassen statisch, also in kompilierter Form, in Java-Programme eingebunden werden können. Dann stellen wir einige der vielfältigen Möglichkeiten vor, Groovy-Skripte dynamisch, also erst zur Laufzeit übersetzt, in Java zu integrieren.

In welcher Weise Sie auch immer Groovy mit Java kombinieren – beachten Sie, dass wir auch hier immer die Groovy-Bibliotheken im Klassenpfad haben müssen, sobald wir eine mit Groovy erstellte Klasse verwenden. Sie benötigen neben den eigentlichen Groovy-Bibliotheken (*groovy-x.x.jar*) auch die ANTLR- und ASM-Bibliotheken (*antlr-x.x.x.jar* und *asm-x.x.jar*), die Sie im *lib*-Ordner unterhalb des Groovy-Installationsverzeichnisses finden. Einfacher ist es, wenn Sie die umfassendere Groovy-Bibliothek (*embeddable/groovy-all.x.x.jar*) benutzen, in der alles Benötigte enthalten ist – wie wir es auch in den obigen Beispielen getan haben.[1]

Und denken Sie daran, dass Groovy eine Java-Version nicht unter 1.4 benötigt. Dies gilt natürlich auch, wenn Sie Groovy-Klassen oder Groovy-Skripte in eigene Java-Programme integrieren.

Statische Integration

Als statische Integration bezeichnen wir hier die Kombination von kompilierten Java- und Groovy-Programmteilen; dabei werden keine Groovy-Skripte oder -Klassen zur Laufzeit übersetzt. Statische Integration bedeutet aber nicht, dass die dynamischen Fähigkeiten von Groovy außer Kraft gesetzt werden. Dies werden Sie weiter unten bestätigt finden, wenn wir auch mit den Mitteln von Java versuchen, die dynamischen Möglichkeiten der Sprache Groovy zu nutzen.

Eine Groovy-Klasse konventionell einbinden

Wir üben die statische Integration einer Groovy-Klasse in ein Java-Programm wieder einmal an einem sehr einfachen Beispiel, an dem sich das Prinzip besonders gut darstellen lässt. Die darin dargestellte Groovy-Quelldatei definiert eine Klasse namens `EineGroovyKlasse`.

```
// Groovy
class EineGroovyKlasse {
    String eineProperty
    def eineMethode() { println 42 }
}
```

Wir schreiben die Datei mit einem normalen Texteditor und speichern sie unter *EineGroovyKlasse.groovy* ab. Die Klasse hat eine Property namens `eineProperty`, und wir wissen, dass Groovy dazu zwei Accessor-Methoden `getEineProperty()` und `setEineProperty()` generiert. Wir ziehen es vor, die Property zu typisieren und nicht einfach nur mit `def` zu deklarieren, denn andernfalls müsste bei jedem lesenden Java-Zugriff ein

1 Dies gilt besonders, wenn Sie selbst die von Groovy verwendeten Libraries einsetzen, aber eine andere Version benötigen. Die *groovy-all*-Bibliothek bindet die ANTLR- und ASM-Klassen in einer Weise ein, die keine Versionskonflikte entstehen lässt.

Typecast durchgeführt werden. Die Methode `eineMethode()` hat keine Parameter und kann wie jede andere Methode eines Java-Objekts verwendet werden.

Ein kurzes Java-Programm instantiiert die Klasse, setzt die Property, liest sie aus und ruft einmal die Methode auf. Wir speichern es als *EinJavaProgramm.java* ab.

```
// Java
public class EinJavaProgramm {
    public static void main(String[] args) {
        EineGroovyKlasse eg = new EineGroovyKlasse();
        eg.setEineProperty("Beispiel");
        System.out.println(eg.getEineProperty());
        eg.eineMethode();
    }
}
```

Übersetzen Sie beide Klassen mit dem Groovy- bzw. dem Java-Compiler und rufen Sie anschließend die Java-Klasse ganz normal auf. Wie gesagt: Dabei dürfen Sie nicht vergessen, die Groovy-Bibliotheken in den Klassenpfad aufzunehmen.

```
> groovyc EineGroovyKlasse.groovy
> javac EinJavaProgramm.java
> java -cp %GROOVY_HOME%\embeddable\groovy-all-1.1.jar;. EinJavaProgramm
Beispiel
42
```

Insofern also eigentlich nichts Besonderes. Allerdings ist zu beachten, dass auf diese Weise nur die tatsächlich in die Klasse hineinkompilierten Felder und Methoden zur Verfügung stehen. Dies sind die explizit definierten Methoden und Felder, die hinzugenerierten Getter- und Setter-Methoden sowie die im Interface `GroovyObject` definierten Methoden.

Auf dynamische Methoden und Properties zugreifen

Die auf das jeweilige Objekt anwendbaren vordefinierten sowie die dynamisch hinzugefügten Methoden können Sie allerdings nutzen, indem Sie die bei allen Groovy-Objekten vorhandenen Methoden `invokeMethod()`, `getProperty()` und `setProperty()` anwenden.

So lässt sich der Aufruf einer Methode `getProperties()` bei einer Instanz von `EineGroovyKlasse`, die in einem Groovy-Programm immer verfügbar ist, in Java nicht kompilieren:

```
// Java
Map properties = eg.getProperties();   // nicht kompilierbar
```

Sie können diese Methode aber dynamisch aufrufen, müssen das Ergebnis allerdings immer mit `cast` auf den erwarteten Typ anpassen:

```
// Java
Map properties = (Map)invokeMethod("getProperties",null);
```

Ähnlich können Sie mit den Properties des Objekts verfahren. Die beiden folgenden Java-Zeilen setzen die Property `eineProperty` und lesen sie aus.

```
// Java
eg.setProperty("eineProperty","ein Text");
String propertyInhalt = (String)eg.getProperty("eineProperty");
```

Allerdings stehen die Methoden `invokeMethod()`, `getProperty()` und `setProperty()` für den dynamischen Zugriff auf Methoden und Properties nur bei Objekten zur Verfügung, die das Interface `GroovyObject` implementieren. Nun kann es aber durchaus einmal vorkommen, dass Sie von Java aus auf Methoden oder Properties eines Nicht-Groovy-Objekts zugreifen möchten. In diesem Fall müssen Sie auf die in Groovy definierte Metaklasse zugreifen. Beispielsweise gibt es eine vordefinierte Methode namens `reverse()` für Strings, die einen String mit den Zeichen des Strings in umgekehrter Reihenfolge liefert. Und auch für Java-Klassen ist die virtuelle Property `properties` verfügbar. Wenn Sie diese Dinge aus Java nutzen möchten, können Sie auf die entsprechende Metaklasse ausweichen:

```
// Java
import groovy.lang.*;
...
String text = "abcdefg";
MetaClass meta = GroovySystem.getMetaClassRegistry()
                 .getMetaClass(text.getClass());
String textVerkehrt = meta.invokeMethod(text,"reverse",null);
Map properties = (Map)meta.getProperty(text,"properties");
```

Ein Wrapper für dynamische Klassen

Alle diese dynamischen Zugriffe auf Groovy-Methoden und -Properties sind etwas umständlich und vor allem architektonisch unschön, weil Sie in Ihrem Java-Code erhebliche Abhängigkeiten von der Groovy-Umgebung bekommen. Völlig verhindern können Sie dies natürlich nicht, wenn Sie die speziellen Möglichkeiten von Groovy in Java-Programmen nutzen möchten. Sinnvoll ist es aber, die Abhängigkeit von Groovy in einer Klasse zu kapseln, die den Zugriff etwas vereinfacht und ohne intime Kenntnisse der Groovy-eigenen Mechanismen verwendet werden kann. Beispiel 8-1 zeigt die Klasse `DynamicObject`, die genau dies leistet. Sie setzt eine Java-Version von mindestens 5.0 voraus, da wir variable Argumentlisten und generische Methoden nutzen.

Beispiel 8-1: Die Java-Klasse DynamicObject ermöglicht dynamische Zugriffe

```
import groovy.lang.*;
import org.codehaus.groovy.runtime.*;
import org.codehaus.groovy.runtime.typehandling.DefaultTypeTransformation;

public class DynamicObject {

    static MetaClassRegistry registry = InvokerHelper.getInstance()
            .getMetaRegistry();

    Object theObject;       // Das Groovy- oder Java-Objekt
    boolean isGroovy;       // Flag für Groovy-Objekt
```

Beispiel 8-1: Die Java-Klasse DynamicObject ermöglicht dynamische Zugriffe (Fortsetzung)

```
/**
 * Konstruktor übernimmt das gekapselte Objekt und prüft,
 * ob es sich um ein Groovy-Objekt handelt
 */
public DynamicObject(Object obj) {
    theObject = obj;
    isGroovy = obj instanceof GroovyObject;
}

/**
 * Untypisierter dynamischer Aufruf einer Methode
 * @param method   Name der Methode
 * @param args     Argumente für den Methodenaufruf
 * @return         Ergebnis als Object
 */
public Object invoke(String method, Object... args) {
    if (isGroovy) {
        return ((GroovyObject)theObject).invokeMethod(method, args);
    } else {
        return getMeta().invokeMethod(theObject, method, args);
    }
}

/**
 * Typisierter dynamischer Aufruf einer Methode
 * @param type     Erwarteter Typ des Ergebnisses
 * @param method   Name der Methode
 * @param args     Argumente für den Methodenaufruf
 * @return         Ergebnis als Objekt des angegebenen Typs
 */
public <T> T invoke(Class<T> type, String method, Object... args) {
    return cast(invoke(method,args),type);
}

/**
 * Untypisierte Abfrage einer Property
 * @param property  Name der Property
 * @return          Wert der Property als Object
 */
public Object get(String property) {
    if (isGroovy) {
        return ((GroovyObject)theObject).getProperty(property);
    } else {
        return getMeta().getProperty(theObject, property);
    }
}

/**
 * Typisierte Abfrage einer Property
 * @param type      Erwarteter Typ des Ergebnisses
```

Beispiel 8-1: Die Java-Klasse DynamicObject ermöglicht dynamische Zugriffe (Fortsetzung)

```
     * @param property  Name der Property
     * @return          Wert der Property als Object
     */
    public <T> T get(Class<T> type, String property) {
        return cast(get(property), type);
    }

    /**
     * Setzt eine Property
     * @param property  Name der Property
     * @param value     Zu setzender Wert
     */
    public void set(String property, Object value) {
        if (isGroovy) {
            ((GroovyObject)theObject).setProperty(property, value);
        } else {
            getMeta().setProperty(theObject, property, value);
        }
    }

    // Ermittelt die Metaklasse zum aktuellen Objekt
    // Nur benötigt, wenn es kein Groovy-Objekt ist
    private MetaClass getMeta() {
        return registry.getMetaClass(theObject.getClass());
    }

    // Typabbildung auf den angegebenen Zieltyp
    // Nutzt die Groovy-Mechanismen zur Typanpassung
    private <T> T cast(Object obj, Class<T> type) {
        Object res = DefaultTypeTransformation.castToType(obj,type);
        return type.cast(res);
    }
}
```

Mit `DynamicObject` können die dynamischen Zugriffe auf `EineGroovyKlasse` nun folgendermaßen aussehen:

```
// Java
EineGroovyKlasse eg = new eineGroovyKlasse();
// DynamicObject-Instanz erzeugen
DynamicObject dyn = new DynamicObject(eg);
// Dynamischer Aufruf einer Methode
Map properties = dyn.invoke("getProperties");
// Dynamisches Setzen und Abfragen einer Property
dyn.set("eineProperty","ein Text");
// Dynamisches Auslesen einer Property Typanpassung
String inhalt = dyn.get(String.class,"eineProperty");
```

In derselben Weise können Sie auch dynamische Methoden und Properties eines Java-Objekts verwenden.

```
// Java
...
String text = "abcdefg";
DynamicObject dynText = new DynamicObject(text);
String textVerkehrt = dynText.invoke("reverse");
Map properties = dynText.get(Map.class,"properties");
```

Wir könnten sogar noch einen Schritt weiter gehen und in `DynamicObject` diejenigen vordefinierten Methoden, die für jedes Objekt verfügbar sind, schon gewissermaßen fest verdrahten. Dadurch würden wir etwas mehr Typsicherheit zur Compile-Zeit gewinnen, die es im dynamischen Groovy nicht geben kann, die wir aber in Java-Programmen nicht ohne Grund schätzen. Als Beispiel könnte es eine zusätzliche Methode `iterator()` geben, die, wie in Groovy üblich, für jedes beliebige Objekt einen Iterator liefert. Damit implementieren wir auch gleich noch das Interface `java.lang.Iterable`.

```
// Java
public class DynamicObject implements Iterable {
...
    public Iterator<T> iterator(Class<T> elementType) {
        return invoke(elementType,"iterator");
}
```

Das Elegante dabei ist, dass wir den Iterator auch gleich in `for`-Schleifen im Java 5.0-Stil einsetzen könnten:

```
// Java
String s = "abcde";
for (Object c : new DynamicObject(s).iterator()) { . . . }
```

Natürlich sollte der Iterator auch noch typisiert werden; wie Sie dies machen, ist aber eher ein Java- als ein Groovy-Problem und soll daher an dieser Stelle nicht weiter behandelt werden.

Architekturfragen

Wenn Sie Anwendungen schreiben, die teils aus Java- und teils aus Groovy-Klassen bestehen, können Sie sehr leicht in ein schwer auflösbares Gestrüpp wechselseitiger Abhängigkeiten geraten, das zu nicht mehr kompilierbaren Programmen führt. Ein einfaches Beispiel zeigt das Problem: Angenommen, Sie haben drei Klassen K1, K2 und K3. K1 benötigt K2 und K2 benötigt K3 (siehe UML-Diagramm in Abbildung 8-1). Wenn nun die Klassen K1 und K3 in Java programmiert sind, K2 aber in Groovy, haben Sie ein Problem, denn Sie müssen die Java- und die Groovy-Klassen getrennt kompilieren. Die Java-Klassen K1 und K3 können Sie nicht kompilieren, da K2 noch nicht vorhanden ist, und die Groovy-Klasse K2 können Sie nicht kompilieren, weil es K3 noch nicht gibt.

In solchen Situationen empfiehlt es sich, die Anwendung in verschiedene Konfigurationseinheiten aufzuteilen, die getrennt kompiliert werden können, und die Klassen durch Interfaces zu entkoppeln.

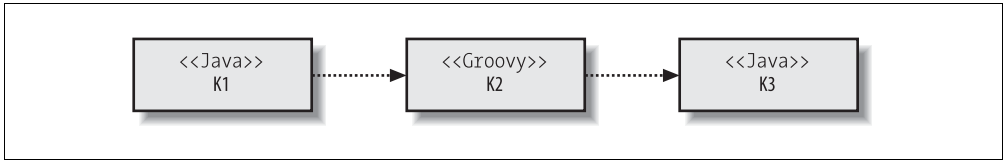


Abbildung 8-1: Wechselseitige Abhängigkeit zwischen Groovy- und Java-Klassen

- Legen Sie eine Konfigurationseinheit A mit zwei in Java geschriebenen Interfaces I2 und I3 an. Die Interfaces müssen von den Klassen K1 bzw. K2 entsprechend implementiert werden.
- Eine zweite Konfigurationseinheit B enthält die Java-Klassen. K1 wird so geändert, dass sie nicht mehr direkt auf K2, sondern stattdessen auf das von K2 implementierte Interface I2 zugreift, und K3 implementiert jetzt das Interface I3.
- Die dritte Konfigurationseinheit C beinhaltet die Groovy-Klassen. Die Klasse K2 wird entsprechend so angepasst, dass sie das Interface I2 implementiert und selbst auf das Interface I3 anstelle von K3 zugreift.

Damit dies funktioniert, können Sie die Klassen K2 und K3 nicht mehr über Konstruktoren instantiieren, sondern müssen dafür eine »neutrale« Factory verwenden. Abbildung 8-2 zeigt diese Struktur der Anwendung.

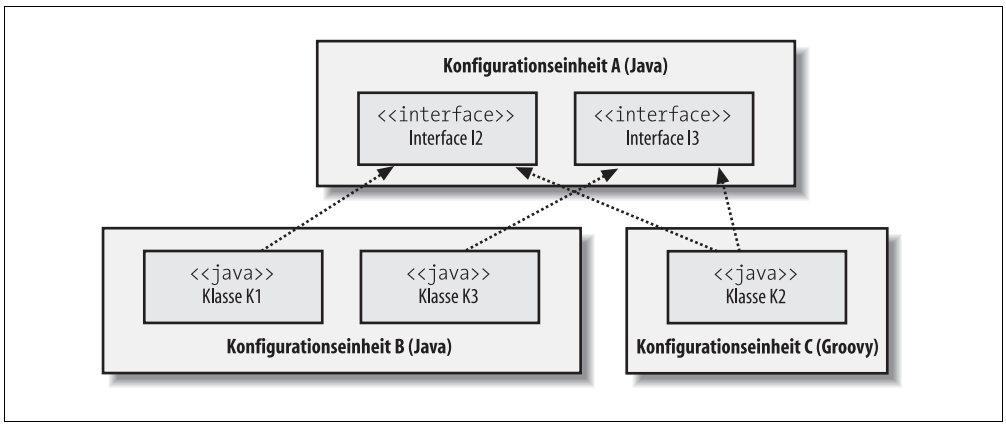


Abbildung 8-2: Auflösung der Abhängigkeit durch getrenntes Kompilieren

Die Java- und Groovy-Klassen sind nun nicht mehr direkt voneinander abhängig, sondern über die Interfaces in Konfigurationseinheit A verknüpft, und die gesamte Anwendung lässt sich problemlos in der Reihenfolge A – B – C kompilieren.

Wir wollen auf die Detailfragen, die mit einer solchen Architektur verbunden sind, nicht näher eingehen, da sie nicht spezifisch für die Groovy-Einbindung sind. Denken Sie aber daran, sich rechtzeitig mit den Problemen wechselseitiger Abhängigkeiten auseinanderzusetzen, wenn Sie gemischte Java-Groovy-Anwendungen schreiben wollen.

Um Anwendungen zu schreiben, die aus sorgfältig entflochtenen Komponenten bestehen, bietet es sich an, einen der zahlreichen verfügbaren Container zu verwenden, der nach dem Inversion-of-Control-Prinzip arbeitet. Bekannte Beispiele hierfür sind das Spring-Framework und der Pico-Container.

Dynamische Integration

Eine der wesentlichen Triebfedern für die Integration von Skriptsprachen in Java-Anwendungen ist die Möglichkeit, Teile der Anwendung auf diese Weise flexibel gestalten zu können, so dass sie modifiziert werden können, ohne dass ein neues Build oder Deployment der gesamten Anwendung erforderlich ist. Dies kann reizvoll bei lokal gültigen Geschäftsregeln, Umrechnungsformeln, Anwendungsoberflächen usw. sein, sofern sie nicht mehr ohne Weiteres durch Konfigurationsdaten darstellbar sind. Sie können einfach mithilfe von Skripten implementiert und dann jeweils den örtlichen und aktuellen Verhältnissen angepasst werden.

Groovy unterstützt die dynamische Integration in Java-Programme auf unterschiedliche Weise. Wir werden uns im Folgenden jedoch auf die wichtigsten Möglichkeiten beschränken, die uns die `GroovyShell` zur dynamischen Integration von Groovy-Skripten in Java-Programmen bietet. Daran lassen sich die wesentlichen Aspekte der Integration so weit erläutern, dass dies für die normalen Anwendungsfälle ausreichen sollte. Die weitergehenden Integrationsmöglichkeiten ausführlich zu behandeln würde den Rahmen dieses Buchs sprengen. Sie sollen aber kurz erwähnt werden.

- Mit der `GroovyShell` können Sie einfache Skripte aus Strings, Dateien und anderen Quellen in unkomplizierter Weise kompilieren und ausführen.
- Die `GroovyScriptEngine` ist besonders zur Ausführung von Skripten geeignet, die wiederum andere Skripte aufrufen.
- Die Klasse `GroovyClassLoader` stellt den Groovy-eigenen Classloader, der letztendlich für die Ausführung aller Groovy-Skripte verantwortlich ist und der auch direkt benutzt werden kann.
- Daneben bestehen noch weitere Integrationsmöglichkeiten, die durch andere Werkzeuge geboten werden, zum Beispiel die Integration in den bekannten Container Spring, der nach dem Inversion-of-Control-Pattern arbeitet.
- Seit Java 6.0 gibt es im JDK eine abstrakte API für die Einbindung über einen `ScriptEngineManager`. Um diese Möglichkeit zu nutzen, benötigen Sie eine Factory-Klasse, die das Interface `javax.script.ScriptEngineFactory` implementiert. Eine solche Klasse kann derzeit noch nicht Bestandteil der Groovy-Standardbibliothek sein, da diese noch mit Java 1.4 lauffähig sein soll. Es ist aber kein großes Problem, eine solche Factory-Klasse selbst zu erstellen.

Beachten Sie auch den Anhang B, in dem die wichtigsten Klassen zur Integration ausführlicher dokumentiert sind.

Einfache Skripte übersetzen und ausführen

Der wesentliche Unterschied zwischen der statischen und der dynamischen Integration besteht darin, dass das Groovy-Programm einmal als Bytecode und einmal als Quellcode vorliegt. Während im ersten Fall der Bytecode einer Klasse direkt eingebunden wird, muss im zweiten Fall erst einmal der Quellcode kompiliert werden, bevor der daraus resultierende Bytecode vom Programm aufgerufen werden kann. Mithilfe der `GroovyShell` geht das folgendermaßen:

```
// Java
import groovy.lang.*;
. . .
String text = "1+1";                    // Quelltext des Skripts
GroovyShell shell = new GroovyShell();  // Eine Instanz der Groovy-Shell
Script script = shell.parse(text);      // Das Skript übersetzen
Object ergebnis = script.run();         // Das Skript ausführen
```

Das Skript besteht hier aus einem einfachen arithmetischen Ausdruck. Durch den Aufruf der `parse()`-Methode der `GroovyShell` mit dem Quellcode als Argument erhalten wir ein Objekt, das das Interface `Script` implementiert. Sie kennen es schon, da wir uns in Kapitel 3 damit beschäftigt haben, das die Besonderheiten von Skripten gegenüber normalen Groovy-Klassen behandelt. Dieses Skript führen Sie aus, indem Sie seine Methode `run()` aufrufen, die das Ergebnis des Skripts als Rückgabewert hat. In diesem Fall ist das Ergebnis ein `Integer`-Objekt mit dem Wert 2.

Die `parse()`-Methode der `GroovyShell` kann auch mit einem `File`-Objekt oder einem `InputStream` als Eingabequelle für den Quelltext aufgerufen werden; am Ergebnis ändert dies jedoch nichts.

Sie können das obige `Script`-Objekt durchaus mehrmals ausführen. In diesem Fall ist dies nicht besonders sinnvoll, denn das Ergebnis wird immer dasselbe sein. Für die Situation, dass ein Groovy-Skript nur einmal ausgeführt werden soll, gibt es eine Methode `evaluate()`, die das Parsen und das Ausführen gleich zusammenfasst. Sie ist besonders handlich für die Auswertung kurzer Formeln.

```
// Java
Object ergebnis = sh.evaluate("1+1");
```

Unabhängig davon, ob Sie das Skript mit der `evaluate()`-Methode oder mit der Kombination aus `parse()` und `run()` ausführen, sind in jedem Fall zwei Dinge zu beachten:

1. Kleiden Sie die Methoden zum Parsen und Ausführen des Skripts in einen try-catch-Block ein und fangen Sie jedes `Throwable` ab, denn in einem dynamischen Skript können sich ja immer unvorhergesehene Fehler befinden.
2. Das Ergebnis ist vom Typ `Object`. Bevor Sie es in einen Typ umwandeln, mit dem Sie weiterarbeiten können, müssen Sie immer eine Typprüfung ausführen, denn Sie wissen nicht im Vorhinein, ob das Skript ein Ergebnis vom erwarteten Typ liefert.

Unter der Annahme, dass das Skript im letzten Beispiel immer einen Integer-Wert liefern soll, könnte der korrekt abgesicherte Aufruf also etwa so aussehen:

```
// Java
String text = "1+1";
int ergebnis = 0;
try {
    Object obj = sh.evaluate(text);
    if (obj instanceof Number) {
        ergebnis = ((Number)obj).intValue();
    } else {
        throw new RuntimeException("Falscher Ergebnistyp");
    }
} catch (Throwable thr) {
    System.err.println("Fehler aufgetreten: "+thr);
}
```

Die Absicherung ist noch nicht komplett, weil wir auf diese Weise nicht verhindern, dass bösartiger Code die Anwendung stört. Dies zu verhindern ist etwas komplizierter; im Abschnitt »Sicherheitsfragen« weiter unten gehen wir auf dieses Problem näher ein.

Das Binding

Sie wissen bereits, dass Sie in Groovy-Skripten undeklarierte Variablen verwenden können. Sie werden in einem als Binding oder Kontext bezeichneten, Map-ähnlichen Objekt gespeichert, das auch für den Austausch von Informationen zwischen dem Skript und seiner Umgebung verwendet werden kann. Auch die `GroovyShell` verfügt über ein solches Binding-Objekt; es wird an die durch `parse()` oder `evaluate()` erzeugten Skripte weitergereicht. Im Normalfall bleiben dadurch die Binding-Variablen von einem Skriptaufruf zum nächsten erhalten, allerdings kann das Binding auch im Script-Objekt ausgetauscht werden.

Das folgende Java-Programm `MiniKonsole` (Beispiel 8-2) ist gewissermaßen eine Sparversion der interaktiven, zeichenorientierten Skriptkonsole `groovysh`, die Sie schon aus Kapitel 1 kennen. Es ermöglicht uns, interaktiv jeweils eine Skriptzeile einzugeben, und zeigt das jeweilige Ergebnis und dessen Typ an, sofern es nicht `null` ist. Eine leere Eingabezeile beendet die Anwendung.

Beispiel 8-2: Die Klasse MiniKonsole

```
// Java
import java.io.*;
import java.util.*;
import groovy.lang.*;

public class MiniKonsole {

    public static void main(String[] args) throws Exception {
```

Beispiel 8-2: Die Klasse MiniKonsole (Fortsetzung)

```
        GroovyShell shell = new GroovyShell();
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        shell.setVariable("start",new Date());
        while (true) {
            System.out.print("groovy> ");
            String zeile = in.readLine();
            if (zeile.trim().length()==0) break;
            try {
                Script script = shell.parse(zeile);
                Object ergebnis = script.run();
                if (ergebnis!=null) {
                    System.out.println("==> "+ergebnis+
                            " ("+ergebnis.getClass().getName()+")");
                }
            } catch (MissingMethodException ex) {
                if (ex.getMethod().equals("main")) {
                    // Klassendefinition ohne main()-Methode
                    // stillschweigend ignorieren
                } else {
                    ex.printStackTrace();
                }
            } catch (Throwable th) {
                th.printStackTrace();
            }
        }
    }
}
```

Wie Sie sehen, werden hier die Skriptaufrufe korrekt gegen Fehler gesichert. Das Auftreten einer `MissingMethodException` wird gesondert behandelt, wenn die fehlende Methode den Namen `main` hat. In diesem Fall enthält das Skript typischerweise eine Klassendefinition ohne `main()`-Methode und lässt sich daher nicht ausführen. Wir überspringen diesen Fehler, denn der Sinn der Zeile besteht dann sicherlich nur darin, die entsprechende Klasse zu definieren, damit sie später verwendet werden kann.

Eine kurze Session mit der `MiniKonsole` zeigt, dass Klassendefinitionen und Binding-Variablen von einem Skriptaufruf zum nächsten erhalten bleiben und dass die in der `MiniKonsole` zu Beginn gesetzte Binding-Variable `start` im Skript verfügbar ist.

```
> java -cp %GROOVY_HOME%\embeddable\groovy-all-1.1.jar;. MiniKonsole
groovy> println start
Sun Jun 10 13:53:48 CEST 2007
groovy> class K1 { def prop; String toString() { "K1=$prop" } }
groovy> k1 = new K1(prop:42)
==> K1=42 (K1)
groovy> k1.properties.each { println it }
prop=42
metaClass=groovy.lang.MetaClassImpl@126e85f[class K1]
class=class K1
groovy>
```

Zusätzliche Funktionalität im Skript

Sie können dem dynamischen Skript nicht nur zusätzliche Informationen in Form von Binding-Variablen mitgeben, sondern auch zusätzliche Funktionalität. Eine Möglichkeit besteht darin, eine eigene Basisklasse zu definieren, die vom Skript anstelle der Klasse `java.lang.Script` implementiert werden soll. Die Alternative besteht in der Zuweisung einer Closure als Binding-Variable.

Angenommen, Sie möchten Ihren Skripten gern eine zusätzliche Methode namens `dump()` zur Verfügung stellen, die den Namen und den Typ aller aktuell im Binding enthaltenen Variablen anzeigt. Dazu können Sie beispielsweise eine neue Skriptklasse `ExtScript` anlegen, wie sie in Beispiel 8-3 zu sehen ist.

Beispiel 8-3: Erweiterte Basisklasse für Skripte

```
public abstract class ExtScript extends Script {
    public void dump() {
        for (Object var : this.getBinding().getVariables().keySet()) {
            String name = (String)var;
            println(name+" : "+
                this.getBinding().getVariable(name).getClass().getName());
        }
    }
}
```

Die Klasse muss abstrakt sein, denn die von `Script` geerbte abstrakte Methode `run()` wird erst im konkreten Skript implementiert. Jetzt müssen wir der `GroovyShell` noch die neue Basisklasse für Skripte bekannt machen; dazu müssen wir eine Instanz der Klasse `CompilerConfiguration` anlegen, der wir den vollständigen Klassennamen der neuen Basisklasse übergeben. Diese `CompilerConfiguration` wiederum übergeben wir im Konstruktor der `GroovyShell`.

```
// Java
CompilerConfiguration config = new CompilerConfiguration();
config.setScriptBaseClass("ExtScript");
GroovyShell shell = new GroovyShell(config);
```

Nun können Sie Skriptzeilen mit der Anweisung `dump()` eingeben.

```
groovy> dump()
start : java.util.Date
k1 : K1
```

Alternativ können Sie auch Closures als Binding-Variablen zuweisen. Am einfachsten ist dies natürlich mit einem kleinen Skript:

```
// Java
shell.evaluate("methode={ arg1,arg2 -> . . . }")
```

Im Skript können sie die Closure unter dem Variablennamen aufrufen, als wäre es eine Methode:

```
groovy> methode(x,y)
```

Dies geht natürlich nur, wenn die Informationen, auf die die Closure zugreift, innerhalb des Skripts – zu Beispiel als Binding-Variablen – verfügbar sind. Wenn dies nicht der Fall ist, können Sie aber recht einfach im Java-Progamm eine Closure erzeugen, die auf eine vorhandene Methode verweist:

```
// Java
shell.setVariable("methode",new MethodClosure(this,"dieMethode"));
```

Sicherheitsfragen

Die dynamische Integration von Skripten in Ihre Java-Programme wirft spezielle Fragen nach der Sicherheit auf. Solchen Skripten sollte nicht erlaubt werden, Aktionen zu unternehmen, die die Funktionalität der Anwendung stören. Die Ursache dafür muss ja nicht unbedingt Böswilligkeit sein – auch wenn ein Benutzer sich irrt oder sich über die Konsequenzen seines Handelns nicht bewusst ist, können die Folgen fatal sein.

Rufen Sie doch noch einmal die `MiniKonsole` von oben auf und geben Sie ein:

```
> java –cp %GROOVY_HOME%\embeddable\groovy-all-1.1.jar;. MiniKonsole
groovy> System.exit(1)
```

Wie Sie sehen, verabschiedet sich das Programm sang- und klanglos. Das ist in diesem Fall natürlich nicht so schlimm, aber wenn das Skript beispielsweise im Rahmen einer unternehmenskritischen Serveranwendung mit Hunderten von Benutzern läuft, sollte so etwas besser vermieden werden.

Java bietet eine Möglichkeit, die Rechte von Teilen eines Programms bezüglich bestimmter Aktionen zu begrenzen. Solche Aktionen sind Zugriffe auf externe Ressourcen, zum Beispiel Dateien, Netzwerkverbindungen oder Datenbanken, oder kritische Manipulationen an der virtuellen Maschine, wie etwa das eben ausprobierte `System.exit()`. Sie können auch Methoden Ihrer eigenen Klassen schützen.

Als Voraussetzung dafür, dass diese Schutzmechanismen überhaupt wirksam werden können, müssen Sie Ihre Anwendung der Kontrolle eines Security-Managers unterstellen; dies ist eine Klasse mit bestimmten Eigenschaften. Im Allgemeinen genügt es, durch Setzen des Schalters `-Djava.security.manager` den seit Java 1.2 standardmäßig mit dem JDK gelieferten Security-Manager zu aktivieren. Dieser Standard-Security-Manager erlaubt erst einmal allen zum JDK gehörenden Klassen alles. Darüber hinaus sind alle kritischen Aktionen allen Klassen verwehrt, sofern sie nicht in einer sogenannten Policy-Datei eigens eine Erlaubnis erhalten haben. Dabei werden die Klassen, die eine solche Erlaubnis bekommen sollen, durch ihre *Codebase* gekennzeichnet, das ist der Dateiname bzw. die URL des Verzeichnisses oder der JAR-Datei, in dem sich die entsprechenden *.class*-Dateien befinden.

Das Ziel besteht nun darin, allen Java-Programmen alle Rechte zu geben (wie es ohne Security-Manager auch der Fall ist) und nur die dynamisch ausgeführten Groovy-Skripte so zu beschränken, dass diese keinen Schaden anrichten können. So einen abgeschotteten Programmteil bezeichnet man auch als *Sandbox* (Sandkasten). Gehen Sie dazu folgendermaßen vor.

Schreiben Sie zunächst eine Policy-Datei, in der Sie Ihrem Programm sowie allen von diesem Programm benötigten Bibliotheken – darunter befindet sich auch die JAR-Datei von Groovy – alle Rechte verleihen. Der Name der Datei ist egal; wir nennen sie einer Konvention folgend *MiniKonsole.policy* (vgl. Beispiel 8-4).

Beispiel 8-4: Policy-Datei MiniKonsole.policy

```
grant codeBase "file:./bin" {
    permission java.security.AllPermission;
};
grant codeBase "file:./lib/*" {
    permission java.security.AllPermission;
};
```

In dieser Policy-Datei gehen wir davon aus, dass sich die Klassendateien Ihres Programms in einem Unterverzeichnis namens *bin* und alle benötigten JAR-Dateien in einem Unterverzeichnis namens *lib* befinden. Diese Pfade müssen Sie natürlich Ihrer Anwendung entsprechend anpassen.

Rufen Sie nun die `MiniKonsole` so auf, dass erstens der Security-Manager aktiviert und zweitens die Policy-Datei benannt wird. Dies geschieht mit den Java-Schaltern `-Djava.security.manager -Djava.security.policy=policy-datei`, die Sie beim Aufruf des Programms mit angeben.

```
 > java –cp %GROOVY_HOME%\embeddable\groovy-all-1.1.jar;. MiniKonsole -Djava.security.
manager -Djava.security.policy=policy-datei
groovy> System.exit(1)
java.security.AccessControlException: access denied (java.lang.RuntimePermission exitVM.1)
```

Wenn Sie jetzt versuchen, die virtuelle Maschine gewaltsam abzubrechen, bekommen Sie durch eine `AccessControlException` den deutlichen Hinweis, dass Sie versucht haben, etwas Verbotenes zu tun. Aber das Programm wird dadurch nicht mehr beendet, denn unsere `MiniKonsole` fängt diese Exception ab wie alle anderen auch.

Die Möglichkeiten von dynamischen Groovy-Skripten innerhalb Ihrer Programme sind auf diese Weise ziemlich begrenzt. In vielen Fällen, zum Beispiel wenn die Skripte zum Berechnen von Formeln dienen oder für einfache Steuerungsaufgaben vorgesehen sind, kann dies durchaus reichen. Wenn Sie Ihre Skripte mit weiter reichenden Befugnissen ausstatten möchten, haben Sie zwei Möglichkeiten: entweder über eine Erweiterung der Policy-Datei oder mithilfe privilegierter Aktionen.

Erweiterte Rechte für Groovy-Skripte

Wenn Sie den dynamischen Skripten zusätzliche Rechte einräumen möchten, können Sie diese im Prinzip – wie bei allen anderen Programmteilen auch – in die Policy-Datei eintragen. Das Problem ist nur, dass es hier keine Codebase gibt, der Sie die Rechte zuordnen können, denn das Programm kann aus irgendwelchen Quellen kommen, und es gibt überhaupt keine Klassendateien. Um dieses Problem lösen zu können, bietet die Groovy-Bibliothek eine virtuelle Codebase an, der Sie irgendeinen Namen geben können – der nicht unbedingt auf eine existierende Datei oder eine sonstige Datenquelle verweisen muss –, und dieser Codebase können Sie Rechte geben wie jeder anderen.

Wir müssen zu diesem Zweck die `MiniKonsole` derart umprogrammieren, dass wir interaktiv aus den eingegebenen Skriptzeilen ein `GroovyCodeSource`-Objekt erzeugen und dieses anstelle des eigentlichen Skriptcodes an die `GroovyShell` übergeben. Der entsprechende Ausschnitt sieht so aus:

```
// Java
GroovyCodeSource gcs = new GroovyCodeSource(zeile,"tempScript","/scripts");
Script script = shell.parse(gcs);
```

Die `GroovyCodeSource` bekommt in diesem Konstruktor drei Strings übergeben: den Text des Skripts, einen Namen für das Skript und einen Namen für die virtuelle Codebase, der nicht auf ein wirklich existierendes Verzeichnis verweisen muss. Diesen Namen können Sie nun dazu verwenden, die Policy-Datei um spezielle Berechtigungen für das Skript zu erweitern:

```
grant codeBase "file:/scripts" {
    permission java.io.FilePermission "-", "read";
    permission java.util.PropertyPermission "file.encoding", "read";
};
```

Wir haben die Codebase *file:/scripts* hier um die Berechtigung erweitert, beliebige Dateien im aktuellen und in allen darunter liegenden Verzeichnissen zu lesen. Zusätzlich benötigen wir noch die Berechtigung, die System-Property `file.encoding` auszulesen, denn diese wird benötigt, um den Zeichensatz für die Dekodierung des Texts zu bestimmen. Wir starten die veränderte `MiniKonsole` mit der veränderten Policy-Datei und versuchen Folgendes (unter der Annahme, dass eine Datei *beispiel.txt* im aktuellen Verzeichnis existiert):

```
groovy> println new File("beispiel.txt").text
Das sind die Daten
groovy> new File("beispiel.txt").write("Das sind die Daten")
java.security.AccessControlException: access denied (java.io.FilePermission beispiel.txt
write)
```

Tatsächlich können wir auf eine Datei im aktuellen Verzeichnis lesend zugreifen, während der Versuch, in dieselbe Datei zu schreiben, an der fehlenden Erlaubnis scheitert.

Privilegierte Aktionen

Wenn Sie die Rechte eines Skripts feiner steuern möchten, bietet sich die Verwendung privilegierter Methoden an, die Sie den Skripten zur Verfügung stellen. Weiter oben haben wir im Abschnitt »Zusätzliche Funktionalität im Skript« gezeigt, wie Sie mithilfe einer eigenen Basisklasse dem Skript zusätzliche Methoden zur Verfügung stellen können. Wir erweitern diese Klasse `getUserName()`, die den aktuellen Benutzernamen aus den System-Properties liest:

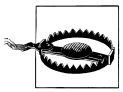
```
public String getFileEncoding() {
    return AccessController.doPrivileged(new PrivilegedAction<String>() {
        public String run() {
            return System.getProperty("user.name");
        }
    });
}
```

Die Methode ruft die System-Property nicht direkt auf, sondern tut dies über eine `PrivilegedAction`; dies hat zur Folge, dass die Methode auch dann ausgeführt wird, wenn sie aus unsicherem Code heraus aufgerufen worden ist. Mit dieser Erweiterung ausgeführt, können Sie nun legal den Benutzernamen abfragen, während das Auslesen des Home-Verzeichnisses nach wie vor nicht möglich ist:

```
groovy> println userName
testuser
groovy> printnln System.getProperty("user.home")
java.security.AccessControlException: access denied (java.util.PropertyPermission user.
home read)
```

Ein Vorteil von privilegierten Methoden besteht darin, dass Sie damit eine genau definierte Aktivität zulassen. Das ist mit einer Security-Policy nicht immer in der gewünschten Granularität möglich.

Dieser Hinweis auf die Möglichkeiten, dynamische, aus Java-Programmen heraus aufgerufene Groovy-Skripte abzusichern, sollte an dieser Stelle genügen. In jedem Fall ist es zu empfehlen, dass Sie sich intensiver mit der Java-Sicherheitsarchitektur auseinandersetzen, wenn Sie in kritischen Anwendungen mit dynamischen Skripten arbeiten wollen.



Denken Sie ebenfalls daran, dass Sie auch dann, wenn Sie benutzerdefinierte Skripte in der total abgeschotteten Sandbox laufen lassen, eine Sicherheitslücke öffnen. Beispielsweise wird es etwa einem Kundigen nicht schwerfallen, sich ein Skript auszudenken, das Ihren Rechner einfach lahmlegt, ohne dabei auf irgendeine geschützte Information zugreifen zu müssen. Wenn Sie also nicht genau wissen, wer die möglichen Autoren der Skripte sind und was diese im Schilde führen könnten, sollten Sie auf jeden Fall weitere Maßnahmen ergreifen, um die Verfügbarkeit Ihres Systems abzusichern.

Am Ende dieses Kapitels über die Integration von Groovy- und Java-Programmen sollte noch angemerkt werden, dass Groovy in dieser Hinsicht Möglichkeiten zur Verfügung stellt, die kaum bei einer anderen Java-basierten Sprache mit skriptartigen Features zu finden sein wird. Das macht Groovy zu *der* Sprache für dynamische Erweiterungen von in Java geschriebenen Programmen.

Mit diesem Kapitel beenden wir auch unsere Betrachtung der Möglichkeiten, die Ihnen Groovy zum Programmieren zur Verfügung stellt, und wenden uns der Frage zu, wie Sie Groovy als Tool einsetzen können, um den Software-Produktionsprozess – auch für Anwendungen, die nicht in Groovy selbst programmiert werden – effektiver zu gestalten.

KAPITEL 9

Groovy im Entwicklungsprozess

In den vorangehenden Kapiteln haben Sie eine Vielfalt von Möglichkeiten kennengelernt, mit deren Hilfe Sie Programme durch den Einsatz von Groovy effizienter erstellen können: Weniger Code, eine mächtige Programmiersprache und erweiterte Bibliotheken erlauben Ihnen, mit weniger und übersichtlicherem Programmcode mehr zu erreichen. Nun wollen wir uns der Frage zuwenden, wie Sie mit Groovy den Entwicklungsprozess selbst optimieren können. Wegen seiner Plattformunabhängigkeit, Skripting-Fähigkeit und der leichten Integrierbarkeit mit Java-Programmen bietet sich Groovy hervorragend als Tool für die Steuerung von Build- und sonstigen automatisierten Prozessen an. Darüber hinaus beinhalten die Groovy-Bibliotheken bereits einige häufig benötigte Werkzeuge wie Ant und JUnit und stellen Hilfsmittel bereit, mit denen diese Tools unkompliziert benutzt und besser integriert werden können.

Wir wenden uns in diesem Kapitel daher zwei Themen zu, die im Entwicklungsprozess größerer Projekte eine besonders wichtige Rolle spielen und bei denen Groovy aufgrund seiner spezifischen Möglichkeiten von großem Nutzen sein kann: der Steuerung von Build-Prozessen mit einer Kombination aus Groovy und Apache Ant sowie der speziellen Unterstützung von Unit-Tests mit Groovy und JUnit.

Die Themen dieses Kapitels sind größtenteils auch für Projekte interessant, die ausschließlich durch konventionelle Java-Programmierung implementiert werden. Groovy kann durch seine Agilität und Mächtigkeit einerseits und durch seine Java-Nähe andererseits die Java-Entwicklung effektiv unterstützen, wobei sich gleichzeitig der Aufwand für das Erlernen einer weiteren Sprache in Grenzen hält und dauerhafte Abhängigkeiten von Groovy vermieden werden können.

Groovy und Ant

In fast jedem Java-Projekt wird *Apache Ant* als Build-Tool eingesetzt, mit dem diverse komplexe Aktivitäten im Zusammenhang mit der Softwareentwicklung automatisiert werden können. Ant umfasst eine Vielzahl vordefinierter, als *Task* bezeichneter Aktionen, die mithilfe von XML-Dateien konfiguriert und zu *Targets* gruppiert werden, zwischen denen wiederum Abhängigkeiten bestehen können. Weitere Tasks kann man mit recht übersichtlichem Aufwand selbst erstellen; Hersteller von Tools, deren Integration in Build-Prozesse sinnvoll ist, liefern sie häufig gleich fix und fertig mit.

Da sich Groovy als Skriptsprache wie gesagt besonders gut dazu eignet, automatisierte Prozesse im Java-Umfeld zu steuern, ist Ant bereits in der Groovy-Laufzeitbibliothek enthalten. Dazu gibt es einige fertige Tasks zur Erweiterung von Ant und eine Klasse namens `AntBuilder`, mit der sich Build-Prozesse ausgesprochen elegant steuern lassen.

Um diesen Abschnitt verstehen zu können, sollten Sie sich etwas mit Ant auskennen. Ausführliche (englischsprachige) Informationen erhalten Sie auf den Projektseiten von Ant unter *http://ant.apache.org/manual*. Oder lesen Sie beispielsweise das Buch *Ant – kurz & gut* von Stefan Edlich und Jörg Staudemeyer (O'Reilly Verlag).

Ant-Tasks für Groovy

Klar, dass Sie Ant auch in größeren Groovy-Projekten einsetzen möchten (für einfache Skripte, die Sie schreiben und gleich ausführen, ist dies natürlich weniger sinnvoll). Um Ihnen dies zu ermöglichen, stellt Groovy einige dafür erforderliche Tasks zur Verfügung.

Um diese Tasks in einem Build-Skript einsetzen zu können, müssen Sie sie zunächst deklarieren. Dies können Sie beispielsweise für den `groovy`-Task mit dem folgenden Stück XML-Code tun, den Sie vor der erstmaligen Verwendung dieses Tasks einfügen sollten.

```
<!-- Klassenpfad für die Groovy-Tasks -->
<path id="GROOVYPATH"
        location="${ENV.GROOVY_HOME}/embeddable/groovy-all-1.1.jar" />
<!-- Taskdefinition für den Groovy-Skriptstarter -->
<taskdef name="groovy"
        classname="org.codehaus.groovy.ant.Groovy"
        classpathref="GROOVYPATH"/>
```

Den neu definierten Task können Sie nun innerhalb beliebiger Targets dazu verwenden, ein Groovy-Skript direkt auszuführen, beispielsweise so:

```
<target name="rungroovy">
    <groovy src="BeispielSkript.groovy"/>
</target>
```

Im Folgenden wollen wir Ihnen die von Groovy mitgelieferten Ant-Tasks mit ihren Attributen vorstellen, soweit sie in üblichen Build-Prozessen üblicherweise Anwendung fin-

den. (Es gibt noch einige weitere Tasks und Attribute, die aber eher für Spezialprobleme im Rahmen der Groovy-Entwicklung verwendet werden.)

Der Task <groovy>

Führt das angegebene Skript direkt aus. Zu dem Task gibt es die folgenden Attribute:

***append* –** Boolescher Wert; legt fest, ob die Standardausgabe an die mit dem Attribut `output` benannte Datei angefügt wird oder ob sie die Datei ersetzen soll. Nur sinnvoll, wenn das Attribut `output` gesetzt ist. Der Vorgabewert ist `false`, d.h., wenn nichts angegeben ist, wird die Datei überschrieben.

***classpath* –** Klassenpfad, der beim Laden des Skripts sowie der von ihm verwendeten Skripts und Klassen verwendet werden soll.

***classpathref* –** Name eines Klassenpfads, der an anderer Stelle des Build-Skripts definiert worden ist.

***output* –** Name einer Datei, in die alle Standardausgaben geschrieben werden sollen. Wenn dieses Attribut nicht gesetzt ist, erscheinen die Ausgaben im Ant-Protokoll.

***src* –** Name der auszuführenden Groovy-Quelldatei. Muss angegeben sein, wenn sich das Skript nicht als Text im Rumpf des Tags befindet.

***stacktrace* –** Boolescher Wert, der angibt, ob bei Übersetzungsfehlern ein vollständiger Stacktrace ausgegeben werden soll. Der Vorgabewert ist `false`.

Das Skript kann entweder durch einen Dateinamen im Attribut `src` angegeben sein oder sich als Klartext zwischen dem Anfangs- und dem End-Tag befinden.

Anstelle des `src`-Attributs können auch ein oder mehrere Pfade für die Quelldateien als innere `<src>`-Tags angegeben werden. Dasselbe gilt für den Klassenpfad, der in Form von inneren `<classpath>`-Tags spezifiziert werden kann.

In einem durch diesen Task ausgeführten Skript stehen einige spezielle Skriptvariablen zur Verfügung:

- `ant` verweist auf einen frisch instantiierten `AntBuilder`, so dass Sie hier Groovy-gesteuerte Ant-Tasks direkt aufrufen können (siehe weiter unten).
- `project` beinhaltet das aktuelle Ant-Projekt als Objekt.
- `properties` ist eine Map mit den Ant-Properties des aktuellen Projekts.
- `target` ist das aktuelle Ant-Target als Objekt.
- `task` verweist auf die Instanz des `Groovy`-Tasks.

Der Task <groovyc>

Übersetzt Groovy-Quelldateien in Java-Klassen analog zum Ant-Task `javac`. Dabei wird eine Quelldatei nur dann übersetzt, wenn es nicht schon eine entsprechende Klassendatei neueren Datums gibt. Der Task `groovyc` kennt diese Attribute:

classpath – Klassenpfad, der beim Kompilieren verwendet werden soll.

classpathref – Name eines Klassenpfads, der an anderer Stelle des Build-Skripts definiert worden ist.

destdir – Zielverzeichnis für die zu erzeugenden *.class*-Dateien. Wenn nicht angegeben, wird das aktuelle Verzeichnis verwendet.

encoding – Kodierung der Quelldateien. Wenn nicht angegeben, wird die Standardkodierung der virtuellen Maschine verwendet.

failonerror – Boolescher Wert, der angibt, ob der Build-Prozess abbrechen soll, wenn beim Kompilieren eines Quellprogramms ein Fehler auftritt. Vorgabewert ist `true`.

listfiles – Boolescher Wert, der angibt, ob die Namen der zu übersetzenden Dateien im Protokoll ausgegeben werden sollen. Vorgabewert ist `false`.

proceed – Das Gegenteil von `failonerror`. Es sollte nur eines der Attribute `proceed` und `failonerror` angegeben sein, andernfalls ist das Ergebnis undefiniert.

srcdir – Wurzelverzeichnis der zu kompilierenden Quelldateien. Muss angegeben sein, wenn es keine inneren `<src>`-Elemente gibt.

stacktrace – Boolescher Wert, der angibt, ob bei Compiler-Fehlern ein vollständiger Stacktrace ausgegeben werden soll. Vorgabewert ist `false`.

verbose – Boolescher Wert, der angibt, ob der Compiler Einzelheiten zum Fortschritt seiner Tätigkeit melden soll. Vorgabewert ist `false`.

Anstelle des `srcdir`-Attributs können auch ein oder mehrere Pfade für die Quelldateien als innere `<src>`-Tags angegeben werden. Dasselbe gilt für den Klassenpfad, der in Form von inneren `<classpath>`-Tags spezifiziert werden kann.

Der Task <groovydoc>

Dieser Task erzeugt eine Quellcode-Dokumentation im Stil von JavaDoc aus den in den Programmen enthaltenen Dokumentationskommentaren. Zu diesem Task gibt es die folgenden Attribute:

destdir – Legt das Zielverzeichnis an, in dem die HTML-Seiten für die Dokumentation abgelegt werden sollen.

packagenames – Kommaseparierte Liste mit den Namen der zu verarbeitenden Packages. Wenn an der Stelle des letzte Pfadelements eines Package-Namens ein Stern angegeben ist (z.B. `groovy.util.*`), werden alle unterliegenden Packages verarbeitet. Ist das Attribut nicht gesetzt, werden alle Packages durchlaufen.

private – Boolescher Wert; legt fest, ob auch private Klassen und Klassen-Member berücksichtigt werden sollen. Vorgabe ist `false`.

sourcepath – Wurzelverzeichnis für die Quelldateien. Muss angegeben sein.

windowtitle – Legt den Titel der zu generierenden HTML-Dateien fest.

Ant mit Groovy statt XML

So perfekt Ant mit seinem enormen Funktionsumfang und seiner Flexibilität als Mittel für Builds und sonstige automatisierte Prozesse auch ist, so gibt es doch ein gravierendes Problem: Die Build-Skripte werden in XML formuliert, einer Sprache, die sich zwar hervorragend für statische Konfigurationsdaten eignet, automatisierte Prozesse sind aber nun einmal dynamisch, und da wird es mit XML schnell kompliziert und unübersichtlich, wenn komplexere Abläufe gesteuert werden müssen. Die Build-Skripte verfügen zwar über Sprachmittel, mit denen sich Abhängigkeiten und Verzweigungen formulieren lassen, sobald es aber über sehr einfache Ablaufstrukturen hinausgeht, gerät man rasch an die Grenzen des Möglichen oder zumindest des Nachvollziehbaren. In der Praxis gibt es daher häufig eine Mischung aus Ant- und Shell-Skripten zur Steuerung von Build-Vorgängen, um die Vielfalt der Möglichkeiten von Ant mit einer verständlichen Ablauflogik zu verbinden. Dies hat aber wieder den Nachteil, dass zwei ganz unterschiedliche Sprachmittel beherrscht werden müssen und dass die resultierenden Skripte in der Regel nicht mehr plattformneutral sind.

Für dieses Problem bietet Groovy eine äußerst elegante Lösung an, bei der prozedurale und deklarative Elemente harmonisch miteinander verknüpft werden können. Sie kennen bereits die verschiedenen Builder in Groovy, mit deren Hilfe sich in quasi-deklaratorischer Weise hierarchische Strukturen aufbauen lassen, seien es XML-Dokumente, Objektbäume oder die Elemente einer Bedienoberfläche. Einen solchen Builder gibt es auch für Ant; er ermöglicht Ihnen, die Aufrufe von Ant-Tasks in normale Groovy-Skripte einzubetten. Das funktioniert nach folgendem Schema:

```
ant = new AntBuilder()
ant.echo "Dies ist Ant"
```

In der ersten Zeile erzeugen wir eine Instanz des `AntBuilder`, und in der zweiten führen wir einen Ant-Task namens `<echo>` aus, der den angegebenen String in das Ant-Protokoll schreibt. Geben Sie die beiden Zeilen ohne jede weitere Vorbereitung beispielsweise in eine `groovysh` ein, und Sie sehen, dass tatsächlich die Ausgabe von Ant erscheint:

```
    [echo] Dies ist Ant.
```

Nun ist die Wirklichkeit natürlich komplizierter. Wie wir in Groovy einen etwas komplexeren Build-Prozess steuern, wollen wir an einem realitätsnahen Lehrbuchbeispiel eines Ant-Skripts zeigen. Es dient dazu, in einem gemischten Java- und Groovy-Projekt folgende Aufgaben auszuführen, für die es jeweils ein Target gibt:

- Target `clean`: Alle Zielverzeichnisse mit ihren Inhalten löschen.
- Target `buildjava`: Alle Java-Quelldateien in ein Verzeichnis *classes* kompilieren.
- Target `buildall`: Alle Groovy-Quelldateien in das *classes*-Verzeichnis kompilieren. Setzt voraus, dass zuvor `buildjava` gelaufen ist, da die Groovy-Klassen von den Java-Klassen abhängig sind.

- Target `runjava`: Führt das Programm aus, indem es die aus einer Groovy-Quelldatei kompilierte Main-Klasse aufruft. Setzt `buildall` voraus.
- Target `rungroovy`: Führt das Programm aus, indem es die Main-Klasse als Groovy-Skript startet. Setzt nur `buildjava` voraus, denn die Groovy-Klassen müssen nicht übersetzt sein.
- Target `dist`: Erstellt aus allen kompilierten Klassen eine JAR-Datei und kopiert sie zusammen mit allen übrigen benötigten Java-Bibliotheken in ein Verzeichnis *dist*. Setzt `clean` und `buildall` voraus.

Beispiel 9-1 zeigt das entsprechende Build-Skript.

Beispiel 9-1: Build-Skript in XML

```
<?xml version="1.0"?>
<project name="BeispielProjekt" basedir="." default="buildall">

    <!-- Einige Variablen definieren -->
    <property environment="ENV"/>
    <property name="JAVA_SOURCE" value="src"/>
    <property name="GROOVY_SOURCE" value="src-groovy"/>
    <property name="LIB" value="lib"/>
    <property name="BUILD" value="classes"/>
    <property name="DIST" value="dist"/>
    <property name="APPNAME" value="beispiel"/>
     <property name="MAINCLASS" value="GroovyKlasse"/>
     <property name="VERSION" value="0.1-beta-6"/>

    <!-- Klassenpfad aus Build-Verzeichnis und Bibliotheken -->
    <path id="MAINPATH">
        <fileset dir="${LIB}">
            <include name="*.jar"/>
        </fileset>
        <pathelement path="${BUILD}"/>
    </path>

    <!-- Klassenpfad für die Groovy-Tasks -->
    <path id="GROOVYPATH"
                location="${ENV.GROOVY_HOME}/embeddable/groovy-all-1.1.jar" />

    <!-- Task-Definitionen für Groovy -->
    <taskdef name="groovyc"
            classname="org.codehaus.groovy.ant.Groovyc"
            classpathref="GROOVYPATH" />
    <taskdef name="groovy"
            classname="org.codehaus.groovy.ant.Groovy"
            classpathref="GROOVYPATH"/>

    <!-- Clean löscht alle Zielverzeichnisse -->
    <target name="clean">
        <delete dir="${BUILD}"/>
```

Beispiel 9-1: Build-Skript in XML (Fortsetzung)

```
        <delete dir="${DIST}"/>
    </target>

    <!-- Buildjava kompiliert nur die Java-Quellen -->
    <target name="buildjava">
        <mkdir dir="${BUILD}"/>
        <javac destdir="${BUILD}" debug="true" failonerror="true">
            <src path="${JAVA_SOURCE}"/>
            <classpath refid="MAINPATH"/>
        </javac>
    </target>

    <!-- Buildall kompiliert alle Quellen -->
    <target name="buildall" depends="buildjava">
        <groovyc destdir="${BUILD}" srcdir="${GROOVY_SOURCE}" classpathref="MAINPATH"/>
    </target>

    <!-- Runjava führt die Anwendung als Java-Executable aus -->
    <target name="runjava" depends="buildjava">
        <java classname="${MAINCLASS}" classpathref="MAINPATH"/>
    </target>

    <!-- Rungroovy führt die Anwendung als Skript aus -->
    <target name="rungroovy" depends="buildall">
        <groovy src="${GROOVY_SOURCE}/${MAINCLASS}.groovy" classpathref="MAINPATH"/>
    </target>

    <!-- Dist erzeugt JAR und kopiert alle Bibliotheken in Zielverzeichnis -->
    <target name="dist" depends="clean, buildall">
        <mkdir dir="${DIST}"/>
        <jar basedir="${BUILD}" destfile="${DIST}/${APPNAME}-${VERSION}.jar" />
        <copy todir="${DIST}">
            <fileset dir="${LIB}" includes="**/*.jar" />
        </copy>
    </target>

</project>
```

Am Anfang definieren wir eine Reihe von Ant-Properties, die ungefähr die gleiche Bedeutung haben wie Variablen in einem Programm. Sie definieren die benötigten Klassenpfade und die beiden Tasks zum Kompilieren und Ausführen von Groovy-Programmen. Dann folgen die einzelnen Targets wie oben beschrieben.

Sie können das Build-Skript als *build.xml* speichern, und sofern Ant richtig installiert ist, genügt ein Aufruf von `ant` oder `ant targetname`, um entweder das angegebene Target oder das Default-Target `buildall` auszuführen.

Wenn wir nun dasselbe mit Groovy machen wollen, müssen wir folgendermaßen vorgehen:

- Alle Ant-Properties in Groovy-Variablen verwandeln.
- Die Targets zu Methoden des Groovy-Skripts umformen. Wo Abhängigkeiten bestehen, müssen die vorausgesetzten Targets als Methoden aufgerufen werden.
- Aus allen Task-Aufrufen Methodenaufrufe einer `AntBuilder`-Instanz machen. Dabei werden die XML-Attribute zu benannten Methodenparametern. Wenn es eingebettete Elemente gibt, müssen diese mit geschweiften Klammern zu einer Closure verbunden werden.
- Überall dort, wo in dem Ant-Skript Attributwerte als Strings angegeben sind, können Sie stattdessen Groovy-Ausdrücke passenden Typs einfügen (z.B. Groovy: `debug:true` anstelle von XML: `debug="true"`).
- Dann benötigen Sie noch etwas Logik, die dafür sorgt, dass das richtige Target aufgerufen wird.

Das Ergebnis sieht aus, wie in Beispiel 9-2 dargestellt.

Beispiel 9-2: Build-Skript in Groovy

```
ant = new AntBuilder()

// Einige Variablen definieren
BASEDIR = '.' // entspricht basedir-Attribut
JAVA_SOURCE = "$BASEDIR/src"
GROOVY_SOURCE = "$BASEDIR/src-groovy"
LIB = "$BASEDIR/lib"
BUILD = "$BASEDIR/classes"
DIST = "$BASEDIR/dist"
APPNAME = "beispiel"
MAINCLASS = "GroovyKlasse"
VERSION = "0.1-beta-6"
DEFAULT_TARGET = "buildall"

// Methode zum Ermitteln von Umgebungsvariablen
def env(name) { System.getenv(name) }

// Klassenpfad aus Build-Verzeichnis und Bibliotheken
MAINPATH = ant.path {
    fileset (dir:LIB) {
        include (name:'*.jar')
    }
    pathelement(path:BUILD)
}

// Task-Definitionen für Groovy
ant.taskdef (name:'groovyc',classname:'org.codehaus.groovy.ant.Groovyc')
ant.taskdef (name:'groovy',classname:'org.codehaus.groovy.ant.Groovy')

// Clean löscht alle Zielverzeichnisse
def clean() {
    println "clean:"
```

Beispiel 9-2: Build-Skript in Groovy (Fortsetzung)

```
    ant.delete(dir:BUILD)
    ant.delete(dir:DIST)
}

// Buildjava kompiliert alle Java-Quellen
def buildjava() {
    println "buildjava:"
    ant.mkdir(dir:BUILD)
    ant.javac(destdir:BUILD,debug:true,failonerror:true) {
        src(path:JAVA_SOURCE)
        classpath(path:MAINPATH)
    }
}

// Buildall kompiliert alle Quellen
def buildall() {
    buildjava()
    println "buildall:"
    ant.groovyc (destdir:BUILD,/*srcdir:GROOVY_SOURCE,*/classpath:MAINPATH)
}

// Runjava führt die Anwendung als Java-Executable aus
def runjava() {
    buildall()
    println "runjava:"
    ant.java (classname:"GroovyKlasse",classpath:MAINPATH)
}

// Rungroovy führt die Anwendung als Skript aus
def rungroovy() {
    buildjava()
    println "rungroovy:"
    ant.groovy (src:"$GROOVY_SOURCE/${MAINCLASS}.groovy",classpath:MAINPATH)
}

// Dist erzeugt JAR und kopiert alle Bibliotheken in Zielverzeichnis
def dist() {
    clean()
    build()
    println "dist:"
    ant.mkdir(dir:DIST)
    ant.jar(destfile:"$DIST/${APPNAME}.${VERSION}.jar",basedir:BUILD)
}

// Skript-Routine
// Wenn kein Target angegeben ist, nehmen wir das DEFAULT_TARGET
if (!args) args = [DEFAULT_TARGET]
// Alle als Argumente angegebenen Targets der Reihe nach aufrufen
args.each { target -> "$target"() }
```

Das Groovy-Programm ist fast eine Eins-zu-eins-Übertragung des Ant-Skripts, was Sie anhand der Kommentare leicht nachvollziehen können sollten. Auch die Task-Definitionen für die beiden Groovy-Tasks `<groovy>` und `<groovyc>` finden Sie wieder. Sie sind etwas kompakter, da wir, weil wir uns bereits in einem Groovy-Skript befinden, nicht noch einmal den Klassenpfad von Groovy deklarieren müssen. Am Anfang einiger Target-Methoden finden Sie die Aufrufe der Target-Methoden, die jeweils vorausgesetzt werden. Außerdem steht dort immer eine `println()`-Anweisung, die den Namen des Targets genau in der gleichen Weise ausgibt, wie Ant es tut.

Am Ende des Skripts gibt es noch genau zwei Zeilen Ausführungslogik. Sie sorgt dafür, dass das Default-Target gesetzt wird, wenn kein Target explizit als Aufrufparameter genannt ist, und ruft dann alle Aufrufparameter als parameterlose Methoden auf. Denken Sie daran, dass sich bei diesem Beispiel die *tools.jar*-Datei aus dem JDK im Klassenpfad befinden muss, da sonst der Java-Compiler nicht gefunden wird.

Sie sehen schon an diesem Beispiel, dass das Groovy-Skript deutlich übersichtlicher ist als das äquivalente Ant-Skript, vor allem wenn Ihnen als Groovy- (oder zumindest Java-)Programmierer die Handhabung von Ant-Skripten weniger vertraut ist. Dieser Unterschied wird aber noch wesentlich drastischer, wenn Sie komplexe Abläufe oder sogar Schleifen abbilden müssen.

Die Tatsache, dass Sie in Groovy das ganze Ant-Paket mit seinen Standard-Tasks immer automatisch dabei haben, hat noch einen netten Nebeneffekt: Sie können es auch an Stellen nutzen, die mit Build-Vorgängen überhaupt nicht zu tun haben. Denken Sie daran, was Sie zu tun haben, um den Inhalt eines Verzeichnisses mit allen Unterverzeichnissen zu löschen. Mit Groovy und seinen vordefinierten Methoden für die Klasse `File` ist die Aufgabe zwar ohnehin übersichtlich, aber zusammen mit Ant wird es geradezu grotesk simpel:

```
new AntBuilder().delete(dir:'C:/build/temp')
```

Fertig.

Der Vollständigkeit halber seien auch zwei Nachteile des `AntBuilder` genannt. Einerseits gibt es hier – wie auch im Vergleich mit Java – die Schwierigkeit, dass es noch keine gleichwertige Toolunterstützung gibt. Wenn Ihre Entwicklungsumgebung ein Werkzeug zum Bearbeiten von Ant-Skripten zur Verfügung stellt, das Sie mit sofortiger Syntaxprüfung, automatischen Ergänzungen und Auswahllisten für mögliche Eingaben verwöhnt, müssen Sie mit Groovy an dieser Stelle etwas Verzicht üben. Am Ende werden Sie aber belohnt mit kompakten, verständlicheren und dadurch leichter zu wartenden Build-Skripten.

Eine spezielle Fähigkeit von Ant-Skripten lässt sich mit dem `AntBuilder` von Groovy nicht ohne Weiteres abbilden: die Behandlung von Abhängigkeiten. Ant ordnet alle auszuführenden Targets mit ihren Abhängigkeiten so an, dass jedes Target nur einmal aufgerufen werden muss, und zwar bevor irgendein anderes Target an die Reihe kommt, das von

ihm abhängig ist. Wir haben in unserem obigen Beispiel die Abhängigkeiten einfach durch verschachtelte Methodenaufrufe abgebildet. Dies kann zu Mehrfachaufrufen desselben Targets und unter bestimmten Bedingungen zu inkonsistenten Ergebnissen führen. In solchen Fällen müssen Sie sich noch etwas einfallen lassen, um die korrekte Reihenfolge der Abarbeitung von Targets zu garantieren.

Hilfreich könnte in diesem Zusammenhang ein derzeit noch im Aufbau befindliches Groovy-Modul namens *Gant* sein, das getrennt erhältlich ist. Auf dem `AntBuilder` aufbauend, unterstützt es auch Build-Skripte mit darüber hinausgehenden Anforderungen. Weitere Informationen dazu finden Sie unter *http://groovy.codehaus.org/Gant*.

Unit-Test

Die Einführung automatisierter Tests ist zumindest in größeren Projekten, die mit Groovy arbeiten, ein absolutes Muss. Anders als bei Java nimmt es der Groovy-Compiler klag- und kommentarlos hin, wenn Sie beispielsweise den Aufruf einer Methode programmieren, die gar nicht existiert. Das muss er auch, denn er kann ja nicht wissen, was es zur Laufzeit aufgrund geänderter Metaklassen, umgelenkter `invokeMethod()`-Methoden in der Klasse selbst, nachträglich angewendeter Kategorienklassen usw. überhaupt für Methoden an einem bestimmen Typ geben kann. Dazu kommt, dass die Auswirkungen mancher der dynamischen Möglichkeiten, die Groovy bietet, schwer zu begrenzen sind. Wenn Sie etwa eine neue Metaklasse für eine vielfach genutzte Klasse registrieren, sind die Auswirkungen möglicherweise nur schwer überschaubar.

Folgerichtig unterstützt daher die Groovy-Bibliothek automatisierte Tests auf unterschiedliche Weise und macht sie dadurch angenehm handhabbar. Genau wie die Automatisierung von Build-Prozessen ist auch dies eine Fähigkeit, die sich sehr gut in Projekten einsetzen lässt, bei denen die eigentliche Implementierung nicht in Groovy erfolgt.

Dieser Abschnitt setzt grundlegende Kenntnisse der Methoden des Unit-Testings und des Test-Frameworks JUnit voraus. Mehr Informationen dazu finden Sie unter *http://junit.sourceforge.net* oder in dem *JUnit Pocket Guide* von Kent Beck (O'Reilly Verlag, englisch).

JUnit, die im Java-Umfeld sehr populäre Open Source-Umgebung für automatisierte Unit-Tests, ist als API bereits in der Groovy-Laufzeitbibliothek enthalten, und es gibt zwei zusätzliche Basisklassen für Tests namens `groovy.util.GroovyTestCase` und `groovy.util.GroovyTestSuite`. Sie sind von den korrespondierenden JUnit-Klassen `TestCase` und `TestSuite` abgeleitet, daher können mit ihnen implementierte Testklassen ohne Weiteres im Testrunner Ihrer Entwicklungsumgebung ausgeführt werden. Ihre wichtigste Eigenschaft besteht darin, dass sie beide eine `main()`-Methode enthalten und dadurch auch ohne Testrunner ausführbar sind. Außerdem stellt `GroovyTestCase` eine Reihe zusätzlicher Prüfmethoden zur Verfügung (z.B. `assertLength()` für alle möglichen unterschiedlichen Typen). In Anhang B finden Sie eine vollständige Liste der hinzugefügten Testmethoden.

Ebenfalls wichtig für automatisierte Tests sind Stellvertreterobjekte (häufig als *Dummies* oder *Mocks* bezeichnet). Sie dienen dazu, andere Objekte zu simulieren, die von den zu testenden Klassen benutzt werden, sich aber nicht für den Testzweck eignen. Grund dafür kann sein, dass die zu ersetzenden Klassen Nebenwirkungen produzieren, die nur schwer wieder rückgängig gemacht werden können, oder von bestimmten externen Bedingungen abhängig sind. Ganz typische Vertreter diese Art sind datenbankbasierte Domain-Objekte, deren Verwendung in Tests einerseits voraussetzt, dass die Datenbank mit ganz bestimmten Testdaten gefüllt ist, andererseits aber die Datenbank in einem Zustand hinterlassen, der sie für weitere Tests nicht mehr zulässt. Andere Gründe für den Einsatz von Stellvertreterobjekten können darin bestehen, dass die zu testenden Objekte einfach nur isoliert werden sollen, um Klarheit zu haben, was genau getestet wird, oder weil die Objekte, auf die die zu testenden Objekte zugreifen sollen, noch nicht fertig implementiert sind.

Unit-Tests mit Groovy durchzuführen ist für ein Java-Entwicklerteam eigentlich der ideale Einstieg in die Arbeit mit Groovy. Wenn man in einem Projekt erst einmal nur die JUnit-Tests mit Groovy erledigt, der eigentliche Anwendungscode weiterhin in Java bleibt, also performanter ist und in der gewohnten Sprache, hält sich das Risiko in Grenzen und man erfährt dennoch schon viel von den Möglichkeiten, die Groovy bietet.

An einem einfachen, aber realitätsnahen Beispiel wollen wir zeigen, wie so ein automatisierter Unit-Test mit Groovy aussehen kann. In unserem Szenario gibt es eine zu prüfende Klasse namens `PersonValidator`, deren Aufgabe darin besteht, Plausibilitätsprüfungen an Objekten des Typs `Person` vorzunehmen. Sie ist in Beispiel 9-3 dargestellt.

Beispiel 9-3: Die Klasse PersonValidator

```
// Java
public class PersonValidator {
    private Person person;
    List<String> fehler = new ArrayList<String>();

    private static Pattern EMAIL_PATTERN = Pattern.compile(
        "^[a-zA-Z][\\w\\.-]*[a-zA-Z0-9]@[a-zA-Z0-9][\\w\\.-]*[a-zA-Z0-9]"+
        "\\.[a-zA-Z][a-zA-Z\\.]*[a-zA-Z]$");

    public PersonValidator(Person person) {
        this.person = person;
    }

    public boolean check() {
        if (isEmpty(person.getName()) {
            fehler.add("Name ist leer.");
        }
        if (!isEmpty(person.getEmail()) &&
                !EMAIL_PATTERN.matcher(person.getEmail()).matches()) {
```

Beispiel 9-3: Die Klasse PersonValidator (Fortsetzung)

```
            fehler.add("Ungültige E-Mail-Adresse");
        }
        // Weitere Prüfungen ...
        return fehler.size()==0;
    }

    public List<String> getFehler() {
        return fehler;
    }

    private static boolean isEmpty(String s) {
        return s==null || s.trim().length()==0;
    }
}
```

`Person` (siehe Beispiel 9-4) ist ein Interface mit zahllosen Gettern und Settern, dessen komplexe Implementierungen datenbankbasiert sind und nur von einer Datenbankzugriffsschicht erzeugt werden:

Beispiel 9-4: Das Interface Person

```
// Java
public interface Person {
    String getName();
    void setName(String name);
    String getEmail();
    void setEmail(String email);
    // zahlreiche weitere Zugriffsmethoden
}
```

Der `PersonValidator` prüft nur, ob der Name nicht leer ist und die E-Mail-Adresse, sofern gesetzt, eine gültige Form hat; wenn eine der beiden Bedingungen nicht erfüllt ist, schreibt er eine entsprechende Meldung in eine Fehlerliste und gibt das Ergebnis `false` zurück. Die zahlreichen weiteren Felder des Typs `Person` interessieren nicht.

Wenn Sie in dieser Situation einen isolierten JUnit-Testfall in Java schreiben wollen, haben Sie ein Problem: Die Originalimplementierung von `Person` steht nicht zur Verfügung, wenn Sie nicht auf die Datenbankzugriffsschicht zurückgreifen wollen. Eine eigene Dummy-Implementierung von `Person` ist sehr aufwendig, da Sie die vielen weiteren Methoden implementieren müssen, die Sie im Test gar nicht interessieren. Und jedes Mal, wenn dem `Person`-Interface eine weitere Methode hinzugefügt wird, müssen Sie Ihren Test anpassen, auch wenn diese neue Methode völlig irrelevant für den Test ist.

Wenn wir den Test in Groovy schreiben, haben wir es da einfacher. Wir implementieren das Interface `Person` einfach mithilfe einer Map, wie wir es in Kapitel 4 gesehen haben, zum Beispiel so:

```
// Groovy
dummy = [:]
dummy.getName = { 'Tim' }
dummy.getEmail = { 'tim@oreilly.com' }
p = dummy as Person
```

Die Variable `p` verweist jetzt auf eine Implementierung von `Person`, deren Methoden – soweit vorhanden – die Closures in der Map sind. Der Aufruf von `p.getName()` und `p.getE-mail()` liefert nun erwartungsgemäß den String `"Tim"` bzw. `"tim@oreilly.com"`. Der Aufruf einer nicht in `Person` deklarierten Methode, zum Beispiel `p.setName("O'Reilly")`, führt natürlich zu einer Exception, wenn sie nicht implementiert ist.

Ein einfacher Testfall auf Basis des `GroovyTestCase` könnte dann so aussehen:

```
// Groovy
import groovy.util.GroovyTestCase

class PersonValidatorTest extends GroovyTestCase {

    void testCheckPositiv() {
        def dummy = [getVorname:{'Tim'},getEmail:{'tim@oreilly.com'}]
        def pv = new PersonValidator(dummy as Person)
        assertTrue pv.check()
        assertEquals (0, pv.getFehler().size())
    }
    void testCheckNegativ() {
        def dummy = [getVorname:{''},getEmail:{'tim.oreilly.com'}]
        def pv = new PersonValidator(dummy as Person)
        assertFalse pv.check()
        assertEquals (2, pv.getFehler().size())
    }
}
```

Sie können das Programm wie jeden anderen auf der Basis der JUnit-Klasse `TestCase` geschriebenen Testfall im Testrunner Ihrer Entwicklungsumgebung laufen lassen. Alternativ lässt es sich aber auch als normales Skript ohne Testrunner starten; in beiden Fällen werden die Testmethoden alle der Reihe nach abgearbeitet, und auch wenn ein Test scheitert, werden die übrigen Tests fortgeführt.

Groovy bietet noch weitere Möglichkeiten, solche Stellvertreterobjekte für Tests einfach zu erstellen. So bilden die Klassen im Package `groovy.mock` ein kleines Framework für Mock-Objekte, die erweiterte Prüfmöglichkeiten für Tests bieten.

Schreiben Sie konsequent Tests auf der Basis von `GroovyTestsCase` und `GroovyTestSuite`, isolieren Sie dabei die zu testenden Klassen mithilfe dynamischer Groovy-Objekte und automatisieren Sie Build und Test mit Skripten, die den `AntBuilder` nutzen, sodass die Prüfungen nach jeder Programmänderung ablaufen und sofort eine Rückmeldung liefern können. So erhalten Sie mit geringstem Aufwand die zuverlässigsten Anwendungen – sogar wenn diese in einer dynamischen Sprache wie Groovy implementiert sind.

Und mit diesem guten Ratschlag sind wir am Ende des Textteils unseres Groovy-Buchs angekommen. Es sollte Sie nun dafür fit gemacht haben, mit Groovy produktiv zu arbeiten. Aber wir konnten längst nicht alles behandeln, was man mit Groovy machen kann. Schon beim Schreiben des Buchs sind neue Features hinzugekommen, und während Sie dies lesen, wird die Entwicklung weitergehen. Es lohnt sich also, hin und wieder auf den Webseiten des Projekts nach Neuem Ausschau zu halten, und auch auf der Website zu diesem Buch finden Sie Hinweise auf Weiterentwicklungen.

ANHANG A

Vordefinierte Methoden

Vordefinierte Methoden (gelegentlich auch als Groovy-JDK oder GDK-Methoden bezeichnet) sind ein wichtiger Bestandteil der Groovy-Laufzeitbibliothek. Mit ihrer Hilfe werden Objekte mit zusätzlicher Funktionalität versehen, wenn sie Instanzen bestimmter Klassen oder Interfaces sind. Diese zusätzlichen Methoden dienen teilweise nur der Vereinfachung des Programmierens, teilweise sind sie aber auch erforderlich, um die speziellen Eigenschaften der Sprache Groovy richtig nutzen zu können. Als vordefinierte Methoden können sie in Groovy den vorhandenen Java-Klassen zugeordnet werden, ohne dass in irgendeiner Weise in Vererbungshierarchien eingegriffen werden muss.

Dieser Anhang soll helfen, einen Überblick über die zahlreichen vordefinierten Methoden zu erlangen, und als Nachschlagewerk dienen. Um die Übersichtlichkeit zu erhöhen, haben wir einige Vereinfachungen vorgenommen; es ist aber immer erkennbar, welche Methoden zu welchem Java-Typ anwendbar sind.

Zur Beschreibung der Methoden benutzen wir eine abgekürzte Form, die nicht der Groovy-Sprachsyntax entspricht, aber eine schnelle Orientierung erlaubt und sich an Java 5.0 anlehnt.

- Die Methoden werden generell dargestellt in der Form

  ```
  Ergebnistyp methodenname (Parameter)
  ```

- Parametrisierte Typennamen im Stil von Java 5.0, wie `List<String>`, deuten an, dass Containertypen aus Objekten bestimmter Typen bestehen.
- Drei Punkte (`...`) weisen darauf hin, dass das Argument mit dem betreffenden Typ mehrfach angegeben oder auch weggelassen werden kann.
- Closure-Parameter werden in geschweiften Klammern dargestellt, dabei sind vor dem -> die Parameter und dahinter der erwartete Ergebnistyp der Closure angegeben. Beispiel:

```
void each { Object element -> void }
```

Hier ist der Parameter der Methode `each()` eine Closure, die als Parameter ein Objekt aus der Collection erhält und von der kein Ergebnis erwartet wird. Die Methode selbst liefert ebenfalls kein Ergebnis.

Allgemeine Hilfsmethoden

Die hier aufgeführten vordefinierten Methoden dienen dazu, an jeder beliebigen Stelle im Programm bestimmte Funktionen auszulösen. Diese Methoden sind zwar prinzipiell der Klasse `Object` zugeordnet, haben aber keinerlei spezifischen Bezug zu dem Objekt, an oder in dem sie aufgerufen werden. Daher werden sie wie Funktionen einer nicht objektorientierten Sprache verwendet.

addShutdownHook()

```
void addShutdownHook { null -> void }
```

Die angegebene argumentlose Closure wird als eigener Thread automatisch beim Beenden der aktuellen virtuellen Maschine ausgeführt.

print(), println(), printf()

```
void print (obj)
void println (obj)
void println ()
void printf (String format, Object ... args)
```

Ruft die analogen Methoden des Standardausgabe-Streams `System.out` auf. Die Methode `printf()` steht nur zur Verfügung, wenn Groovy unter Java 5.0 betrieben wird.

sleep()

```
static void sleep (long millis)
static void sleep (long millis) { java.lang.InterruptedException ex
-> void }
```

Diese statische Methode unterbricht die Ausführung für die angegebene Anzahl von Millisekunden. Unterbrechungen werden während dieser Zeit intern abgefangen; man kann also davon ausgehen, dass die angegebene Zeit tatsächlich ungefähr eingehalten wird. Wenn eine Closure angegeben ist, wird diese im Fall einer Unterbrechung mit einer `InterruptedException` als Argument aufgerufen.

sprintf()

```
void sprintf (String format, Object ... args)
```

Wie `printf()`, allerdings wird das Ergebnis als String zurückgegeben und nicht auf der Konsole ausgegeben.

use()

```
Object use (Class ... kategorien) { null -> Object }
Object use (List<Class> kategorien) { null -> Object }
```

Die beiden Methoden bewirken die Ausführung der übergebenen Closure, dabei werden bei allen innerhalb der Closure (auch indirekt) ausgelösten Methodenaufrufen die in den Kategorienklassen definierten Methoden wie vordefinierte Methoden behandelt. Im Gegensatz zu den vordefinierten Methoden von Groovy haben diese Methoden jedoch Vorrang vor den Methoden, die durch die jeweiligen Objekte selbst deklariert sind. Die beiden `use`-Methoden unterscheiden sich darin, dass bei der einen eine variable Anzahl einzelner Kategorienklassen und bei der anderen eine Liste von Kategorienklassen angegeben werden kann.

Beide Methoden liefern den Rückgabewert der Closure ihrerseits als Ergebnis zurück.

Für alle Objekte geltende Methoden

Die folgenden Methoden sind für `java.lang.Object` definiert und gelten damit für alle Objekte, sofern sie nicht in der betreffenden Klasse überschrieben sind und sofern es keine spezifischeren vordefinierten Methoden gibt. Aus Gründen der Übersichtlichkeit sind sie hier vorangestellt.

Für manche diese Methoden gibt es für bestimmte Typen spezialisierte Varianten; diese sind in dieser Übersicht nur dann noch einmal gesondert aufgeführt, wenn sie für den jeweiligen Typ eine spezifische Bedeutung haben.

Diese Methoden sind in einem Groovy-Programm für alle Objekte definiert. Neben den hier aufgelisteten gibt es einige weitere vordefinierte Methoden, die zwar der Klasse `Object` zugeordnet sind, aber keinen Bezug zu dem Objekt haben, in dem sie aufgerufen werden. Diese sind weiter oben unter »Allgemeine Hilfsmethoden« gesondert aufgeführt.

asType()

```
<T> T asType(java.lang.Class<T> type)
```

Wandelt das aktuelle Objekt in eine Instanz des angegebenen Typs um. Dient der Implementierung des `as`-Operators. Die Methode `asType()` wird aber in vielen Klassen überschrieben.

dump()

```
String dump()
```

Generiert einen zusammenfassenden String aus Klassennamen, Hashcode und Feldern des Objekts, der besonders für Debugging-Zwecke geeignet ist.

getAt() – Operator: [ ] (Index)

```
Object getAt (String property)
```

Ermöglicht den lesenden Zugriff auf die Properties eines Objects über den Property-Namen wie auf den Index einer Map.

Beispiel: `objekt[index]` wird übersetzt in `objekt.getAt(index)`, und dies ist wiederum gleichbedeutend mit `objekt.getProperty(index)`.

getMetaPropertyValues()

```
List<MetaProperty> getMetaPropertyValues()
```

Liefert eine Liste aus Objekten des Typs `groovy.lang.MetaProperty`. Diese Objekte beschreiben die Properties des aktuellen Objekts.

getProperties()

```
Map<String,Object> getProperties()
```

Liefert eine Map mit Name-Wert-Paaren, die alle Properties des Objekts enthält. Die Map kann nicht zum Setzen von Properties im Objekt verwendet werden.

identity()

```
identity { Object thisObject -> Object } : Object
```

Ruft die Closure mit dem aktuellen Objekt selbst als Argument auf. Objektreferenzen in der Closure werden ebenfalls über das aktuelle Objekt aufgelöst, da es gleichzeitig als Delegate angemeldet ist. Die Zeile

```
objekt.identity { closure }
```

ist also ungefähr gleichbedeutend mit

```
closure.delegate=objekt
closure.call (objekt)
```

Allerdings ist nach dem `call()`-Aufruf der Delegate nicht mehr gesetzt.

inspect()

```
String inspect ()
```

Generiert einen String, dessen Inhalt die Form hat, mit der in einem Groovy-Programm ein Objekt mit demselben Inhalt wie das aktuelle Objekt gebildet werden könnte. Wenn dies nicht möglich ist, liefert die Methode dasselbe Ergebnis wie `toString()`.

Beispiel: Das Ergebnis von `[1,2,3].inspect()` ist ein String mit dem Inhalt `"[1,2,3]"`.

invokeMethod()

```
Object invokeMethod (String name, Object args)
```

Ruft die Methode mit dem angegebenen Namen und den in `args` enthaltenen Argumenten auf. Dabei kann `args` ein einzelnes Argument oder ein Array mit Argumenten sein.

is()

```
is (Object obj) : Boolean
```

Prüft, ob das aktuelle Objekt identisch mit dem als Argument übergebenen Objekt ist. Diese Methode wird benötigt, da der Gleichheitsoperator (`==`) in Groovy die `equals()`-Methode aufruft und nicht, wie in Java, auf Objektidentität prüft.

isCase()

```
isCase (Object switchObject) : Boolean
```

Prüft, ob das als Argument übergebene `switch`-Objekt in einer `switch-case`-Verzweigung ein »Fall« des aktuellen Objekts ist. Wenn diese Methode nicht überladen ist, gibt sie das Ergebnis eines Aufrufs von `equals(switchObject)` zurück.

print(), println()

```
print (PrintWriter writer)
println (PrintWriter writer)
```

Gibt eine String-Repräsentation des aktuellen Objekts über die `print()`- bzw. `println()`-Methode des angegebenen `PrintWriter` aus.

putAt()

```
putAt (String name, Object wert)
```

Ermöglicht den lesenden Zugriff auf die Properties eines Objekts über den Property-Namen wie auf den Index einer Map.

Iterative Methoden

Die Groovy-Laufzeitbibliothek definiert eine Reihe von Methoden, mit denen Objektmengen iterativ abgearbeitet werden können und die für verschiedene Containerklassen in der gleichen Form definiert sind. Sie gelten auch für einzelne Objekte, die keine Container sind; in diesem Fall wird das Objekt wie ein Container behandelt, der nur dieses eine Objekt enthält.

any()

```
boolean any()
boolean any { Object element -> boolean }
```

Prüft, ob mindestens eines der enthaltenen Elemente den Wert `true` ergibt. Wenn eine Closure angegeben ist, wird diese für die Prüfung jedes einzelnen Elements aufgerufen; andernfalls werden die Groovy-Regeln für die Interpretation von Wahrheitswerten angewendet. Verwendet die `iterate()`-Methode des aktuellen Objekts. Die Schleife wird abgebrochen, sobald ein Closure-Aufruf `true` liefert.

collect()

```
Collection collect (Collection collection = new List()) { Object element -> Object }
```

Liefert eine Collection aus den Ergebnissen, die von der angegebenen Closure bei dem Aufruf für jedes Element geliefert werden. Wenn eine Collection als Argument übergeben worden ist, werden die Closure-Ergebnisse zu dieser hinzugefügt, andernfalls legt die Methode ein neues `List`-Objekt an.

each()

```
void each { Object element -> void }
```

Führt die angegebene Closure für jedes Element aus.

eachWithIndex()

```
void eachWithIndex { Object element, int index -> void }
```

Führt die angegebene Closure für jedes Element aus. Der Closure wird neben dem jeweiligen Element auch ein laufender Zähler, beginnend bei 0, übergeben.

every()

```
boolean every { Object element -> boolean }
boolean every()
```

Prüft, ob alle enthaltenen Elemente den Wert `true` ergeben. Wenn eine Closure angegeben ist, wird diese für die Prüfung jedes einzelnen Elements aufgerufen; andernfalls werden die Groovy-Regeln für die Interpretation von Wahrheitswerten angewendet. Verwendet die `iterate()`-Methode des aktuellen Objekts.

find()

```
Object find { Object element -> boolean }
```

Ermittelt das erste Element, für das die angegebene Closure `true` ergibt.

findAll()

```
List findAll { Object element -> boolean }
```

Stellt eine Liste aus allen Elementen zusammen, für die die angegebene Closure `true` ergibt.

findIndexOf()

```
int findIndexOf { Object element -> boolean }
```

Ermittelt den Index, beginnend bei 0, des ersten Elements, für das die angegebene Closure `true` ergibt.

grep()

```
List grep (Object filter)
```

Durchläuft alle Elemente des aktuellen Objekts und liefert alle Elemente daraus, die mit dem als Argument übergebenen Objekt übereinstimmen, als Liste zurück. Ob es eine Übereinstimmung gibt, wird jeweils durch einen Aufruf von `obj.isCase()` ermittelt.

iterator()

```
Iterator iterator()
```

Liefert einen Iterator über alle enthaltenen Elemente. Diese Methode existiert bei vielen Java-Klassen, die mit dem Collection-Framework konform gehen. Standardmäßig liefert diese Methode einen Iterator, der nur gelesen werden kann und als einziges Element das aktuelle Objekt liefert.

Array- und Listenmethoden

Eine weitere Gruppe von vordefinierten Methoden ist auf alle Objekte anwendbar, die Elemente enthalten, auf die über einen numerischen Index zugegriffen werden kann. Standardmäßig sind dies alle Arrays sowie Objekte, die das Interface `java.util.List` implementieren.

Es versteht sich von selbst, dass bei typisierten Arrays ausgelesene oder zugewiesene Elemente immer vom entsprechenden Typ sein müssen, auch wenn sie im Folgenden als `Object` bezeichnet sind.

Neben den hier aufgeführten vordefinierten Methoden sind auf diese Objekte natürlich auch die obigen iterativen Methoden anwendbar.

equals()

```
boolean equals(java.lang.Object[] right)
boolean equals(java.util.List right)
```

Diese Varianten der standardmäßigen `equals()`-Methode ermöglichen es, Listen und Arrays untereinander und gegenseitig zu vergleichen. Die Methoden liefern dann und nur dann `true`, wenn entweder beide Objekte `null` sind oder beide Objekte gleich viele Elemente haben und der Vergleich aller Elemente mit an der gleichen Position `equals()` `true` ergibt.

getAt() – lesender Indexoperator [ ]

```
Object getAt (int index)
```

Liefert das Element mit dem angegebenen Index.

```
List getAt (Collection indizes)
List getAt (Range indexIntervall)
```

Liefert eine Liste der Elemente mit den angegebenen Indizes. Die gewünschten Indizes können durch eine Collection oder durch ein Intervall benannt sein.

inject()

```
Object inject (Object arg) { Object arg, Object element -> Object }
```

Durchläuft die Elemente in der Weise, dass im ersten Durchgang der Closure das übergebene Argument und der erste Listenwert übergeben werden. In den folgenden Durchgängen erhält die Closure dann jeweils das Closure-Ergebnis des vorherigen Durchgangs und den entsprechenden Listenwert. Das Ergebnis des letzten Closure-Aufrufs stellt dann den Rückgabewert der Methode `inject()` dar.

join()

```
String join (String separator)
```

Wandelt alle Elemente in Strings um und verknüpft sie zu einem langen String, dabei wird der angegebene Separator jeweils zwischen zwei Elementen eingefügt.

putAt() – schreibender Indexoperator [ ]

```
void putAt(int index, Object value)
```

Setzt den Wert mit dem angegebenen Index. Listen werden, wenn der Wert außerhalb der aktuellen Listenlänge liegt, entsprechend vergrößert und mit `null`-Werten aufgefüllt.

size() `int size()`

Liefert die aktuelle Größe des Containerobjekts. Diese Methode ist bei Collections ohnehin vorhanden; die vordefinierte Methode für Arrays ermöglicht es, Arrays und Collections in der gleichen Weise zu behandeln.

toList() `List toList()`

Wenn das aktuelle Objekt ein Array ist, bildet die Methode eine modifizierbare Liste aus dessen Elementen. Bei Arrays aus primitiven Typen findet ein entsprechendes Boxing statt. Wenn das aktuelle Objekt bereits eine Liste ist, wird es selbst zurückgegeben.

Methoden zu einzelnen Java-Typen

Primitive Typen

Zu byte[]

encodeBase64()

`Writable encodeBase64()`

Erzeugt ein Objekt, das das Interface `groovy.lang.Writable` implementiert und den in Base64 kodierten Inhalt des Arrays enthält. Dieser kann über seine Methode `write()` effizient in einen `Writer` geschrieben werden. Ein Aufruf von `toString()` an dem Ergebnis liefert den kodierten Wert als String.

Das Gegenstück zu dieser Methode ist die String-Methode `decodeBase64()`.

Package java.io

Zu java.io.BufferedReader

getText() `String getText()`

Liest den gesamten Inhalt des Readers in einen String und schließt den Reader anschließend.

Zu java.io.BufferedWriter

writeLine() `void writeLine (String line)`

Schreibt einen String in den Writer und fügt einen plattformspezifischen Zeilenwechsel hinzu.

Zu java.io.DataInputStream

iterator() `Iterator<Byte> iterator ()`

Liefert einen Iterator, der den Inhalt des Datenstroms als Bytes liefert.

Zu java.io.File

append()
```
void append (Object text)
void append (Object text, String zeichensatz)
```

Wandelt das Argument in einen String um und schreibt ihn an das Ende der Datei. Wenn der Zeichensatz angegeben ist, wird dieser für die Kodierung verwendet, andernfalls wird die Standardkodierung angewandt.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine schreibbare Textdatei bezeichnet.

asWritable()
```
Writable asWritable ()
java.io.File asWritable (String encoding)
```

Wandelt die Datei in ein Objekt, das das Interface `groovy.lang.Writable` implementiert. Dies ermöglicht es, den Inhalt der Datei direkt über einen Writer auszugeben. Wenn der Zeichensatz angegeben ist, wird dieser für die Kodierung verwendet, andernfalls wird die Standardkodierung angewandt.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine lesbare Textdatei bezeichnet.

eachByte() `void eachByte { Byte byte -> void }`

Ruft die angegebene Closure für jedes Byte der Datei auf.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine lesbare Datei bezeichnet.

eachDir() `void eachDir { File dir -> void }`

Ruft die angegebene Closure für jedes im Verzeichnis enthaltene Unterverzeichnis auf.

Diese Methode setzt voraus, dass das aktuelle File-Objekt ein Dateiverzeichnis bezeichnet.

eachDirMatch() `void eachDirMatch (Object filter) { File dir -> void }`

Ruft die angegebene Closure für jedes im Verzeichnis enthaltene Unterverzeichnis auf, sofern der Aufruf der Methode `isCase()` beim Filter mit dem Unterverzeichnis `true` ergibt.

Diese Methode setzt voraus, dass das aktuelle File-Objekt ein Dateiverzeichnis bezeichnet.

eachDirRecurse()

```
void eachDirRecurse { File dir -> void }
```

Ruft die angegebene Closure rekursiv für jedes im Verzeichnis enthaltene Unterverzeichnis auf.

Diese Methode setzt voraus, dass das aktuelle File-Objekt ein Dateiverzeichnis bezeichnet.

eachFile()

```
void eachFile { File datei -> void }
```

Ruft die angegebene Closure für jede im Verzeichnis enthaltene Datei auf.

Diese Methode setzt voraus, dass das aktuelle File-Objekt ein Dateiverzeichnis bezeichnet.

eachFileMatch()

```
void eachFileMatch (Object filter) { File datei -> void }
```

Ruft die angegebene Closure für jede im Verzeichnis enthaltene Datei auf, sofern der Aufruf der Methode `isCase()` beim Filter mit dieser Datei `true` ergibt.

Diese Methode setzt voraus, dass das aktuelle File-Objekt ein Dateiverzeichnis bezeichnet.

eachFileRecurse()

```
void eachFileRecurse { File datei -> void }
```

Ruft die angegebene Closure rekursiv für jede im Verzeichnis enthaltene Datei auf.

Diese Methode setzt voraus, dass das aktuelle File-Objekt ein Dateiverzeichnis bezeichnet.

eachLine()

```
void eachLine { String zeile -> void }
```

Liest die Datei zeilenweise und ruft mit jeder Zeile die angegebene Closure auf. Die Kodierung wird aus dem Inhalt der Datei ermittelt.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine lesbare Textdatei bezeichnet.

eachObject()

```
void eachObject { Object obj -> void }
```

Liest die Datei Objekt für Objekt und ruft jeweils die angegebene Closure auf.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine Datei mit serialisierten Objekten bezeichnet.

filterLine()

```
Writable filterLine { String zeile -> boolean }
```

Erzeugt ein Objekt, das das Interface `groovy.lang.Writable` implementiert und alle Zeilen enthält, für die die angegebene Closure `true` ergibt.

```
void filterLine (Writer writer) { String zeile -> boolean }
```

Schreibt alle Zeilen der Datei in den angegebenen Writer, bei denen der Aufruf der Closure `true` ergibt.

Beide Methoden setzen voraus, dass das aktuelle File-Objekt eine lesbare Textdatei bezeichnet.

getText()

```
String getText ()
String getText (String zeichensatz)
```

Liest den gesamten Inhalt der Datei in einen String. Wenn der Zeichensatz angegeben ist, wird dieser für die Kodierung verwendet, andernfalls wird die Kodierung aus dem Inhalt ermittelt.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine lesbare Textdatei bezeichnet.

iterator()

```
Iterator<String> iterator()
```

Liefert einen Iterator, mit dem die Datei zeilenweise gelesen werden kann.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine lesbare Textdatei bezeichnet.

leftShift() – Operator <<

```
File leftShift (Object text)
```

Wandelt das Argument in einen String um und schreibt diesen an das Ende der Datei. Wenn der Zeichensatz angegeben ist, wird dieser für die Kodierung verwendet, andernfalls wird die Standardkodierung angewandt.

Als Ergebnis wird das aktuelle File-Objekt geliefert, so dass diese Operation auch verkettet werden kann.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine schreibbare Textdatei bezeichnet.

new…Stream(), new…Writer(), new…Reader()

```
DataInputStream newDataInputStream ()
DataOutputStream newDataOutputStream ()
BufferedInputStream newInputStream ()
ObjectInputStream newObjectInputStream ()
ObjectOutputStream newObjectOutputStream ()
BufferedOutputStream newOutputStream ()
PrintWriter newPrintWriter (String zeichensatz=null)
```

```
BufferedReader newReader ()
BufferedReader newReader (String zeichensatz=null)
BufferedWriter newWriter (java.lang.String zeichensatz=null, boolean
append=false)
```

Einfache Hilfsmethoden zum Erzeugen eine Streams, eines Readers oder eines Writers für das aktuelle File-Objekt. Wenn der Zeichensatz angegeben ist, wird dieser für die Kodierung verwendet, andernfalls wird die Kodierung aus dem Inhalt ermittelt. Wenn der Parameter `append` auf `true` gesetzt ist, wird ein Writer erzeugt, der an den bestehenden Inhalt der Datei anfügt; andernfalls werden bei schreibenden Streams und Writern gegebenenfalls die Dateien überschrieben.

Diese Methoden setzen voraus, dass das aktuelle File-Objekt eine les- bzw. schreibbare Textdatei bezeichnet.

readBytes()

```
byte[] readBytes()
```

Liest den gesamten Inhalt der Datei in ein `byte`-Array.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine lesbare Datei bezeichnet.

readLines()

```
List<String> readLines()
```

Liest den gesamten Inhalt der Datei zeilenweise in eine Liste von Strings.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine lesbare Textdatei bezeichnet.

size()

```
long size()
```

Ermittelt die Länge der Datei. Diese Methode ist ein Synonym für die `length()`-Methode der Klasse `File`, soll aber sicherstellen, dass es eine einheitliche Methode für die Abfrage der Größe eines Objekts gibt.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine Datei bezeichnet.

splitEachLine()

```
void splitEachLine (String regex) { String ... args -> void }
```

Zerteilt jede Zeile der Datei anhand eines regulären Ausdrucks in Teilstrings und übergibt diese jeweils als getrennte Argumente an die Closure.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine lesbare Textdatei bezeichnet.

with…Stream(), with…Reader(), with…Writer()

```
void withDataInputStream { DataInputStream in -> void }
void withDataOutputStream { DataOutputStream out -> void }
void withInputStream { InputStream in -> void }
void withObjectInputStream { ObjectInputStream in -> void }
```

```
void withObjectOutputStream { ObjectOutputStream out -> void }
void withOutputStream { OutputStream out -> void }
void withPrintWriter { PrintWriter out -> void }
void withReader { BufferedReader in -> void }
void withWriter { BufferedWriter out -> void }
void withWriter (String zeichensatz=null) { BufferedWriter out -> void }
void withWriterAppend (String zeichensatz=null) { BufferendWriter out
-> void }
```

Jede dieser Methoden erzeugt einen Stream, Reader oder Writer auf der aktuellen Datei und übergibt diesen an die angegebene Closure. Dabei wird sichergestellt, dass die Ein- oder Ausgabe in jedem Fall auch wieder geschlossen wird. Wenn ein Zeichensatz angegeben ist, wird dieser für die Kodierung verwendet, andernfalls wird die Kodierung aus dem Inhalt ermittelt. Die Methode `withWriterAppend()` öffnet die Datei zum Hinzufügen von Daten, alle anderen schreibenden Methoden überschreiben die Datei, wenn sie bereits vorhanden ist.

Diese Methoden setzen voraus, dass das aktuelle File-Objekt eine les- bzw. schreibbare Textdatei bezeichnet.

write() `void write (String text, String zeichensatz=null)`

Schreibt den Inhalt des Strings in die Datei. Wenn der Zeichensatz angegeben ist, wird dieser für die Kodierung verwendet, andernfalls wird die Standardkodierung benutzt.

Diese Methode setzt voraus, dass das aktuelle File-Objekt eine schreibbare Datei bezeichnet.

Zu java.io.InputStream

eachByte() `void eachByte { Byte byte -> void }`

Ruft die angegebene Closure für jedes Byte des Streams auf. Der Stream wird am Ende geschlossen.

eachLine() `void eachLine { String zeile -> void }`

Liest den Stream zeilenweise und ruft mit jeder Zeile die angegebene Closure auf. Verwendet die Standardkodierung. Der Stream wird am Ende geschlossen.

filterLine() `Writable filterLine { String zeile -> boolean }`

Erzeugt ein Objekt, das das Interface `groovy.lang.Writable` implementiert und alle Zeilen enthält, für die die angegebene Closure `true` ergibt.

`void filterLine (Writer writer) { String zeile -> boolean }`

Schreibt alle Zeilen der Datei in den angegebenen Writer, bei denen der Aufruf der Closure `true` ergibt.

Beide Methoden schließen den Stream am Ende.

getText()

```
String getText ()
String getText (String zeichensatz)
```

Liest den gesamten Inhalt des Streams in einen String. Wenn der Zeichensatz angegeben ist, wird dieser für die Kodierung verwendet, andernfalls wird die Standardkodierung benutzt. Der Stream wird danach geschlossen.

iterator()

```
Iterator<Character> iterator()
```

Liefert einen Iterator, mit dem die Daten Zeichen für Zeichen gelesen werden können.

newReader()

```
java.io.BufferedReader newReader()
```

Erzeugt einen `BufferedReader` zum Lesen der Daten als Text. Verwendet die Standardkodierung.

readLine()

```
String readLine()
```

Liest eine einzelne Textzeile aus dem Stream. Verwendet die Standardkodierung.

readLines()

```
List<String> readLines()
```

Liest die gesamten Daten zeilenweise in eine Liste von Strings. Danach wird der Stream geschlossen.

withReader()

```
void withReader { BufferedReader in -> void }
```

Erzeugt einen `BufferedReader` auf dem Stream und übergibt diesen an die angegebene Closure. Dabei wird sichergestellt, dass der Stream in jedem Fall auch wieder geschlossen wird. Es wird die Standardkodierung angewendet.

withStream()

```
void withStream { InputStream in -> void }
```

Übergibt den aktuellen Stream an die Closure und schließt ihn, nachdem sie aufgerufen worden ist.

Zu java.io.ObjectInputStream

eachObject()

```
void eachObject { Object obj -> void }
```

Liest den Stream Objekt für Objekt und ruft jeweils die angegebene Closure auf. Der Stream wird am Ende geschlossen.

Zu java.io.OutputStream

leftShift() – Operator <<

```
Writer leftShift (Object text)
```

Wandelt das Argument in einen String um und schreibt ihn in den Stream. Es wird die Standardkodierung verwendet. Als Ergebnis wird ein `java.io.Writer` geliefert, sodass diese Operation auch effizient verkettet werden kann.

```
OutputStream leftShift (InputStream in)
```

Leitet den angegebenen `InputStream` direkt in den `OutputStream`.

```
OutputStream leftShift (byte[] value)
```

Schreibt das `byte`-Array in den Stream.

withStream()

```
void withStream { OutputStream out -> void }
```

Übergibt den aktuellen Stream an die Closure und schließt ihn, nachdem sie aufgerufen worden ist.

withWriter()

```
void withWriter (String zeichensatz=null) { OutputStreamWriter out
-> void }
```

Erzeugt einen `OutputWriter` auf dem aktuellen Stream, übergibt ihn an die Closure und schließt ihn, nachdem sie aufgerufen worden ist. Wenn ein Zeichensatz angegeben ist, wird dieser für die Zeichenkodierung verwendet; andernfalls wird die Standardkodierung angewandt.

Zu java.io.Reader

eachLine()

```
void eachLine { String zeile -> void }
```

Liest den Reader zeilenweise und ruft mit jeder Zeile die angegebene Closure auf. Der Reader wird am Ende geschlossen.

filterLine()

```
Writable filterLine { String zeile -> boolean }
```

Erzeugt ein Objekt, das Interface `groovy.lang.Writable` implementiert und alle Zeilen enthält, für die die angegebene Closure `true` ergibt.

```
void filterLine (Writer writer) { String zeile -> boolean }
```

Schreibt alle Zeilen der Datei in den angegebenen Writer, bei denen der Aufruf der Closure `true` ergibt.

Beide Methoden schließen den Stream am Ende.

getText() `String getText ()`

Liest den gesamten Inhalt des Readers in einen String. Der Reader wird danach geschlossen.

iterator() `Iterator<String> iterator()`

Liefert einen Iterator, mit dem die Daten Zeile für Zeile gelesen werden können.

readLine() `String readLine()`

Liest eine einzelne Textzeile aus dem Reader.

readLines() `List<String> readLines()`

Liest die gesamten Daten zeilenweise in eine Liste von Strings. Danach wird der Reader geschlossen.

splitEachLine() `void splitEachLine (String regex) { String ... args -> void }`

Liest den Reader zeilenweise, zerteilt dabei jede Zeile anhand eines regulären Ausdrucks in Teilstrings und übergibt diese jeweils als getrennte Argumente an die Closure.

transform...()
```
void transformChar (Writer writer) { String einZeichen -> String }
void transformLine (Writer writer) { String eineZeile -> String }
```

Übergibt jedes aus dem Reader gelesene Zeichen bzw. jede aus dem Reader gelesene Zeile an die Closure und schreibt den Rückgabewert der Closure jeweils in den angegebenen Writer.

withReader() `void withReader { BufferedReader in -> void }`

Erzeugt einen `BufferedReader` auf dem aktuellen Reader und übergibt diesen an die angegebene Closure. Dabei wird sichergestellt, dass der Stream in jedem Fall auch wieder geschlossen wird.

Zu java.io.Writer

leftShift() – Operator <<

`Writer leftShift (Object text)`

Wandelt das Argument in einen String um und schreibt ihn in den Writer. Als Ergebnis wird der Writer selbst geliefert, so dass diese Operation auch verkettet werden kann.

withWriter() `void withWriter (String zeichensatz=null) { OutputWriter out -> void }`

Übergibt den aktuellen Writer an die Closure. Dabei wird sichergestellt, dass der Stream in jedem Fall auch wieder geschlossen wird.

write() `void write (groovy.lang.Writable writable)`

Diese Methode ermöglicht es, dass Objekte, die das Interface `groovy.langWritable` implementieren, effizient in den Writer geschrieben werden können.

Package java.lang

Zu java.lang.Boolean

and() – Operator: &&

`Boolean and (Boolean zweiterWert)`

Implementiert den logischen Und-Operator.

or() – Operator: ||

`Boolean or (Boolean zweiterWert)`

Implementiert den logischen Oder-Operator.

xor() `Boolean xor (Boolean zweiterWert)`

Führt die logische exklusive Oder-Operation aus.

Zu java.lang.Byte[]

Siehe auch oben unter »Array- und Listenmethoden«.

encodeBase64()

`Writable encodeBase64()`

Erzeugt ein Objekt, das das Interface `groovy.lang.Writable` implementiert und den in Base64 kodierten Inhalt des Arrays enthält. Dieser kann über seine Methode `write()` effizient in einen `Writer` geschrieben werden. Ein Aufruf von `toString()` an dem Ergebnis liefert den kodierten Wert als String.

Das Gegenstück zu dieser Methode ist die String-Methode `decodeBase64()`.

Zu java.lang.Character

Arithmetische Operationen

```
int compareTo (Number zweiterWert)
int compareTo (Character zweiterWert)
Number div (Number zweiterWert)
Number div (Character zweiterWert)
Number intdiv (Number zweiterWert)
Number intdiv (Character zweiterWert)
Number minus (Number zweiterWert)
Number minus (Character zweiterWert)
Number multiply (Number zweiterWert)
Number multiply (Character zweiterWert)
Number next ()
Number plus (Number zweiterWert)
Number plus (Character zweiterWert)
Number previous ()
```

Diese Methoden implementieren verschiedene arithmetische Operationen, so dass ein `Character`-Objekt auf beiden Seiten der entsprechenden Operatoren wie ein ganzzahliger Wert behandelt werden kann. Die Methoden sind unter `java.lang.Number` im Einzelnen erklärt.

Zu java.lang.CharSequence

getAt() – lesender Indexoperator [ ]

```
CharSequence getAt (int index)
CharSequence getAt (Collection indizes)
CharSequence getAt (Range indexIntervall)
```

Ermöglicht den Zugriff auf `String`-, `StringBuffer`- und sonstige Objekte, deren Klassen `CharSequence` implementieren, wie auf ein `Character`-Array. Allerdings ist das Ergebnis immer ein Objekt des Originaltyps. Dies gilt auch, wenn als Index eine einzelne Zahl angegeben ist. Der Ausdruck

```
"abc"[1]
```

ergibt also nicht etwa eine `Character`-, sondern eine `String`-Instanz mit dem Wert `"b"`.

Zu java.lang.Class

getMetaClass()

```
ExpandoMetaClass getMetaClass ()
```

Erzeugt eine `ExpandoMetaClass` zu der aktuellen Klasse, wenn sie nicht bereits existiert, registriert sie bei der `MetaClassRegistry` und gibt sie als Ergebnis zurück. Die `ExpandoMetaClass` erlaubt das nachträgliche Definieren von Methoden und Properties zu der aktuellen Klasse.

isCase() `boolean isCase (Object switchValue)`

Diese Methode ermöglicht die Verwendung von Klassen und Interfaces in `switch`-Verzweigungen. Wenn das Argument ebenfalls ein `Class`-Objekt ist, prüft sie, ob der bezeichnete Typ zuweisungskompatibel zu dem aktuellen Typ ist.

newInstance() `Object newInstance (Object ... args)`

Erzeugt eine neue Instanz der aktuellen Klasse durch den Aufruf des zu der angegebenen Argumentliste passenden Konstruktors. Nur anwendbar, wenn das aktuelle Class-Objekt eine instantiierbare Klasse benennt.

Zu java.lang.ClassLoader

getRootLoader()

`ClassLoader getRootLoader()`

Durchsucht den eigenen sowie alle übergeordneten Classloader nach einem Classloader, dessen Name `"org.codehaus.groovy.tools.RootLoader"` lautet, und liefert diesen als Ergebnis. Wenn ein solcher Classloader nicht gefunden wird, ist das Ergebnis `null`. Der Rootloader kann dazu verwendet werden, in einem Skript noch zur Laufzeit den Klassenpfad zu erweitern.

Zu java.lang.Double

Siehe auch die Methoden zu `java.lang.Number`.

round() `long round()`

Rundet auf einen ganzzahligen Wert. Die Rundung wird durch einen Aufruf von `java.lang.Math.round()` ausgeführt.

java.lang.Float

Siehe auch die Methoden zu `java.lang.Number`.

round() `long round()`

Rundet auf einen ganzzahligen Wert. Die Rundung wird durch einen Aufruf von `java.lang.Math.round()` ausgeführt.

Zu java.lang.Number

Die Groovy-Laufzeitbibliothek implementiert eine Reihe von arithmetischen Operationen auf dem abstrakten Typ `java.lang.Number`, die hier erläutert werden. Häufig gibt es

Varianten für konkrete numerische Typen, die in dieser Übersicht aber nicht aufgeführt werden, da sie – abgesehen vom Ergebnistyp – keine spezielle Semantik haben.

Diverse arithmetische Operationen sind so definiert, dass auf beiden Seiten des entsprechenden Operators auch ein `Character`-Objekt stehen kann. Siehe dazu auch die Anmerkung zu `Character`.

abs()

```
Number abs()
```

Liefert den Absolutwert der Zahl.

and() – Operator: &

```
Number and (Number zweiterWert)
```

Implementiert die bitweise Und-Operation.

compareTo() – Operatoren: <, <=, >, >=, ==, <=>

```
int compareTo (Character right)
int compareTo (right)
```

Implementiert diverse numerische Vergleichsoperatoren.

div() – Operator: /

```
java.lang.Number div (java.lang.Character right)
java.lang.Number div (java.lang.Number right)
```

Implementiert den Divisionsoperator.

downto()

```
void downto (java.lang.Number to) { Number zähler -> void }
```

Zählt abwärts vom eigenen bis zu dem angegebenen Wert und ruft bei jedem Durchgang die angegebene Closure auf.

intdiv()

```
java.lang.Number intdiv(java.lang.Character right)
java.lang.Number intdiv(java.lang.Number right)
```

Führt eine ganzzahlige Division zwischen dem eigenen und dem angegebenen Wert aus.

leftShift – Operator: <<

```
java.lang.Number leftShift (java.lang.Number anzahl)
```

Führt eine arithmetische Linksverschiebung um die angegebene Anzahl von Stellen aus. Diese Methode ist nur auf ganzzahlige Typen anwendbar.

minus() – Operator: –

```
java.lang.Number minus (java.lang.Character subtrahend)
java.lang.Number minus(java.lang.Number subtrahend)
```

Berechnet die Differenz zwischen dem eigenen Wert und dem angegebenen Subtrahenden.

mod() – Operator: %

```
java.lang.Number mod(java.lang.Number divisor)
```

Berechnet den Modulo-Wert zwischen dem eigenen Wert und dem angegebenen Divisor.

multiply() – Operator: *

```
java.lang.Number multiply(java.lang.Character multiplikator)
java.lang.Number multiply(java.lang.Number right)
```

Berechnet das Produkt aus dem eigenen Wert und dem angegebenen Multiplikator.

negate() – Operator: ~

```
java.lang.Number negate()
```

Berechnet den bitweisen Komplementärwert des eigenen Werts. Diese Methode ist nur auf ganzzahlige Typen anwendbar.

next() – Operator: ++ (Post- und Präinkrement)

```
java.lang.Number next()
```

Liefert einen um 1 erhöhten Wert.

or() – Operator: |

```
java.lang.Number or(java.lang.Number zweiterWert)
```

Führt ein bitweises Oder zwischen dem eigenen und dem angegebenen Wert aus.

plus() – Operator: +

```
java.lang.Number plus(java.lang.Character summand)
java.lang.Number plus(java.lang.Number summand)
```

Bildet die Summe aus dem eigenen Wert und dem Argument.

```
java.lang.String plus(java.lang.String string)
```

Wenn das Argument ein String ist, wird der aktuelle Wert in einen String umgewandelt und eine String-Verknüpfung mit dem Argument ausgeführt.

power() – Operator: **

```
java.lang.Number power (java.lang.Number exponent)
```

Berechnet die Potenz aus dem eigenen Wert und dem angegebenen Exponenten.

previous() – Operator: – – (Post- und Prädekrement)

```
java.lang.Number previous()
```

Liefert einen um 1 verminderten Wert.

rightShift() – Operator: >>

```
java.lang.Number rightShift(java.lang.Number right)
```

Führt eine arithmetische Rechtsverschiebung um die angegebene Anzahl von Stellen aus. Diese Methode ist nur auf ganzzahlige Typen anwendbar.

rightShiftUnsigned() – Operator: >>>

```
java.lang.Number rightShiftUnsigned(java.lang.Number right)
```

Führt eine arithmetische Rechtsverschiebung um die angegebene Anzahl von Stellen ohne Berücksichtigung eines Vorzeichens aus. Diese Methode ist nur auf ganzzahlige Typen anwendbar.

step()

```
void step (Number bis, Number schrittweite) { Number zähler -> void }
```

Zählt mit der angegebenen Schrittweite bis zu dem Wert `bis`. Bei jedem Durchgang wird die Closure mit dem jeweiligen Zählerstand aufgerufen. Der Zielwert muss größer als der eigene Wert sein.

times()

```
void times { Number zähler -> void }
```

Ruft die Closure entsprechend dem eigenen Wert mehrmals auf, dabei wird der Closure ein bei 0 beginnender ganzzahliger Zähler übergeben.

to...()

```
java.math.BigDecimal toBigDecimal()
java.math.BigInteger toBigInteger()
java.lang.Double toDouble()
java.lang.Float toFloat()
java.lang.Integer toInteger()
java.lang.Long toLong()
```

Wandelt den aktuellen Wert in ein Objekt des entsprechenden Typs um. Wenn es bereits ein Objekt des Zieltyps ist, wird das aktuelle Objekt selbst zurückgegeben.

upto()

```
void downto (java.lang.Number to) { Number zähler -> void }
```

Zählt aufwärts vom eigenen bis zu dem angegebenen Wert und ruft bei jedem Durchgang die angegebene Closure auf.

xor()

```
java.lang.Number xor(java.lang.Number zweiter Wert)
```

Führt ein bitweises exklusives Oder zwischen dem eigenen und dem angegebenen Wert aus.

java.lang.Object

Siehe oben unter »Allgemeine Hilfsmethoden« und »Für alle Objekte geltende Methoden«.

Zu java.lang.Object[]

Siehe auch oben unter »Array- und Listenmethoden«.

toArrayString() `java.lang.String toArrayString()`

Liefert den Inhalt des Arrays als String, eingeschlossen in geschweifte Klammern.

toSpreadMap() `groovy.lang.SpreadMap toSpreadMap()`

Erzeugt eine `SpreadMap`-Instanz; dabei dienen die Elemente des Arrays abwechselnd als Schlüssel und als Wert eines Map-Eintrags.

toString() `String toString()`

Ruft `toArrayString()` auf und liefert dadurch eine lesbare Form des Array-Inhalts.

Zu java.lang.Process

consumeProcessOutput()

```
void consumeProcessOutput()
```

Startet zwei Threads, die asynchron die Standardausgabe- und den Standardfehlerkanal des Prozesses auslesen, ohne die dabei erhaltenen Daten in irgendeiner Weise zu verarbeiten. Mithilfe dieser Methode kann verhindert werden, dass ein Prozess wegen eines vollen Ausgabezwischenspeichers stehen bleibt.

getErr(), getIn(), getOut()

```
java.io.OutputStream getErr()
java.io.InputStream getIn()
java.io.OutputStream getOut()
```

Liefert die Standardeingabe-, -ausgabe- und Standardfehlerkanäle des Prozesses als Streams. Dies sind einfache Alias-Methoden, die nur dazu dienen, die Streams wie Properties ansprechen zu können.

getText() `java.lang.String getText()`

Liest die gesamte Standardausgabe des Prozesses in einen String.

leftShift() – Operator <<

```
Writer leftShift (Object text)
```

Wandelt das Argument in einen String um und schreibt ihn in die Standardeingabe des Prozesses. Es wird die Standardkodierung verwendet. Als Ergebnis wird ein `java.io.Writer` geliefert, so dass diese Operation auch effizient verkettet werden kann.

```
OutputStream leftShift (InputStream in)
```

Leitet den angegebenen `InputStream` direkt in den `OutputStream`.

```
OutputStream leftShift (byte[] value)
```

Schreibt das `byte`-Array in die Standardeingabe des Prozesses.

waitForOrKill()

```
void waitForOrKill (long numberOfMillis)
```

Wartet auf die Beendigung des Prozesses und bricht ihn nach Ablauf der angegebenen Anzahl Millisekunden ab.

Zu java.lang.String

Siehe auch unter »`java.lang.CharSequence`«. Die vordefinierten Methoden für Strings haben zum Teil den Zweck, eine ähnliche Behandlung wie bei Listen und Arrays zu ermöglichen. Beachten Sie aber, dass auch hier – wie bei den Methoden zu `CharSequence` – immer Strings und keine `Character`-Werte zurückgeliefert werden.

contains()

```
Boolean contains (String text)
```

Prüft, ob der aktuelle String den angegebenen String enthält. Diese Methode ist ab Java 5.0 bereits in der Java-Standardbibliothek enthalten.

count()

```
int count (String s)
```

Stellt fest, wie oft der als Argument angegebene String in dem aktuellen String vorkommt. Überlappende Teilstrings werden mitgezählt.

decodeBase64()

```
byte[] decodeBase64 ()
```

Nimmt an, dass der aktuelle String in Base64 kodierte Daten sind, und wandelt diese in ein Byte-Array um. Es gibt eine korrespondierende vordefinierte Methode `encodeBase64()` für `byte`-Arrays.

eachMatch()

```
void eachMatch (String regex, Closure c)
```

Ruft für jedes Vorkommen des regulären Ausdrucks `regex` im aktuellen String die angegebene Closure auf. Dabei wird der Closure als Argument jeweils ein String-Array mit den Werten der einzelnen Match-Gruppen übergeben.

execute()

```
Process execute (String[] env, File dir)
Process execute (List env, File dir)
Process execute ()
```

Alle drei Methoden führen den Inhalt des aktuellen Strings als Betriebssystembefehl aus. Die Umgebungsvariablen können als String-Array oder String-Liste übergeben werden. Wenn der Parameter `dir` nicht `null` ist, enthält er das Arbeitsverzeichnis des zu startenden Prozesses; andernfalls dient das aktuelle Verzeichnis als Arbeitsverzeichnis. Folgendes Beispiel startet unter Windows den Explorer:

```
"explorer.exe".execute()
```

getAt() – Operator: [ ] (Index lesend)

```
String getAt (int n)
String getAt (Collection coll)
String getAt (Range r)
```

Implementiert den Indexoperator für einen einzelnen numerischen Indexwert, für eine Liste von Indexwerten oder für einen Bereich mit Indexwerten. Liefert einen String aus den Zeichen, die sich an den angegebenen Indexwerten befinden. Negative Indexwerte werden dabei vom String-Ende an rückwärts gezählt.

isCase()

```
Boolean isCase (Object kandidat)
```

Implementiert die `switch-case`-Verzweigung. Führt einen String-Vergleich mit `kandidat` durch.

leftShift() – Operator: <<

```
StringBuffer leftShift (Object obj)
```

Wandelt das aktuelle Objekt in einen `StringBuffer` um, hängt das übergebene Objekt an diesen und übergibt den `StringBuffer` als Rückgabewert. Da der `StringBuffer` ebenfalls den Operator << implementiert, können auf diese Weise mehrere String-Verkettungen in einem Stück durchgeführt werden. Das Ergebnis kann dann ohne explizite Umwandlung einem String zugewiesen werden.

minus() – Operator: –

```
String minus (Object obj)
```

Wandelt das übergebene Objekt in einen String und entfernt dessen erstes Vorkommen aus dem aktuellen String, sofern er in ihm enthalten ist.

multiply() – Operator: *

```
String multiply (Number n)
```

Vervielfacht den aktuellen String so oft wie angegeben. Das Argument wird in eine Ganzzahl umgewandelt und muss größer oder gleich 0 sein. Wenn es 0 ist, wird ein leerer String zurückgegeben.

negate() – Operator: ~

```
java.util.regex.Pattern negate()
```

Implementiert den unären Operator ~. Kompiliert den aktuellen String, der einen regulären Ausdruck enthält, zu einem Pattern-Objekt.

next() – Operator: ++ (Post- und Präinkrement)

```
String next()
```

Erhöht das letzte Zeichen des aktuellen Strings um 1 und liefert das Ergebnis als neues `String`-Objekt. Wenn das letzte Zeichen gleich `Character.MAX_VALUE` ist, wird es auf `Character.MIN_VALUE` gesetzt. Wenn der aktuelle String leer ist, wird `Character.MIN_VALUE` zurückgegeben.

pad...()

```
String padLeft (Number länge, String füllzeichen=' ')
String padRight (Number länge, String füllzeichen=' ')
```

Verlängert den aktuellen String auf die angegebene Größe. Bei `padLeft()` wird er links, bei `padRight()` wird er rechts, bei Bedarf wiederholt, mit `füllzeichen` aufgefüllt. Wenn die vorherige Länge bereits größer oder gleich `länge` ist, bleibt er unverändert. Das Argument `füllzeichen` muss mindestens ein Zeichen lang sein; standardmäßig dienen Leerzeichen als Füllzeichen.

plus() – Operator: +

```
String plus (Object obj)
```

Wandelt das Argument in einen String, hängt diesen an den aktuellen String an und übergibt das Ergebnis als neues `String`-Objekt.

previous() – Operator: – – (Post- und Prädekrement)

```
String previous ()
```

Implementiert den unären Operator --. Vermindert das letzte Zeichen des aktuellen Strings um 1 und liefert das Ergebnis als neues `String`-Objekt. Wenn das letzte Zeichen gleich `Character.MIN_VALUE` ist, wird es aus dem String entfernt. Der aktuelle String darf nicht leer sein.

replaceAll()

```
String replaceAll (String regex, Closure c)
```

Ruft für alle Übereinstimmungen des regulären Ausdrucks `regex` im aktuellen String die angegebene Closure auf und fügt das jeweilige Ergebnis anstelle

des übereinstimmenden String-Abschnitts in den aktuellen String ein. Dabei werden alle Match-Gruppen der Reihe nach als Closure-Argumente übergeben.

reverse() `String reverse ()`

Gibt einen String mit den gleichen Zeichen, wie sie im aktuellen String enthalten sind, aber in umgekehrter Reihenfolge zurück.

size() `int size ()`

Gibt die Anzahl der im aktuellen String enthaltenen Zeichen zurück. Diese Methode ist gleichwertig mit der String-Methode `length()` und soll einen einheitlichen Zugriff auf die Eigenschaften eines Objekts über verschiedene Typen hinweg ermöglichen.

to...()

```
BigDecimal toBigDecimal ()
BigInteger toBigInteger ()
Boolean toBoolean ()
Character toCharacter ()
Double toDouble ()
Float toFloat ()
Integer toInteger ()
Long toLong ()
```

Diese Methoden wandeln den aktuellen String in den durch den Methodennamen bezeichneten Typ um. Dazu werden in der Regel die String-Konstruktoren der jeweiligen Zielklassen verwendet.

tokenize() `List tokenize (String trennzeichen=' \t\n\r\f')`

Verwandelt den aktuellen String mithilfe des `java.util.StringTokenizer` in eine Liste von Teilstrings. Die in dem als Argument übergebenen String enthaltenen Zeichen dienen dabei als Trennzeichen. Standardmäßig verwendet `StringTokenizer` Leerzeichen, Tabulatorzeichen, Zeilenwechsel, Rücklauf- und Seitenwechselzeichen als Trenner.

toList() `List toList ()`

Erzeugt eine Liste mit Strings, die jeweils ein einzelnes Zeichen des aktuellen Strings enthalten.

toURI(), toURL()

```
java.net.URI toURI ()
java.net.URL toURL ()
```

Wandelt den aktuellen String in ein `URI`- bzw. ein `URL`-Objekt.

Zu java.lang.String[]

execute() `java.lang.Process execute()`

Führt einen Betriebssystembefehl aus. Dabei bildet das erste Array-Element (mit dem Index 0) den eigentlichen Befehl, und die übrigen Elemente werden als Parameter übergeben.

Zu java.lang.StringBuffer

Siehe auch unter »`java.lang.CharSequence`«. Während die Methoden für den Typ `String` das aktuelle Objekt zwangsläufig nicht verändern können, nehmen folgende Methoden, die für die Klasse `java.lang.StringBuffer` vordefiniert sind, Modifikationen am aktuellen Objekt vor. Die unsynchronisierte Klasse `java.lang.StringBuilder`, die ab Java 1.5 in den meisten Fällen anstelle des langsameren `StringBuffer` verwendet werden kann, wird durch Groovy 1.0 noch nicht unterstützt.

leftShift() – Operator: <<

`StringBuffer leftShift (Object obj)`

Wandelt das Argument in einen String, hängt ihn an den aktuellen `StringBuffer` und liefert einen Verweis auf diesen als Ergebnis. Dadurch können mehrere Verkettungsoperationen verknüpft werden. Siehe auch die `leftShift()`-Methode zur Klasse `String`.

putAt() – Operator: [ ] (Index schreibend)

`void putAt (IntRange intervall, Object obj)`

Implementiert den schreibenden Indexoperator mit einem Intervall. Der durch das Intervall gekennzeichnete Teil des aktuellen `StringBuffer` wird durch das in einen String umgewandelte zweite Argument ersetzt.

size() `int size ()`

Gibt die Anzahl der im aktuellen `StringBuffer` enthaltenen Zeichen zurück. Diese Methode ist gleichwertig mit der `StringBuffer`-Methode `length()` und soll einen einheitlichen Zugriff auf die Eigenschaften eines Objekts über verschiedene Typen hinweg ermöglichen.

Package java.net

java.net.ServerSocket

accept() `java.net.Socket accept { Socket socket -> void }`

Startet beim Eingang einer neuen Verbindung einen Thread, der die angegebene Closure ausführt. Die Closure erhält den Socket der Verbindung als Argument übergeben. Nachdem die Closure ausgeführt worden ist, wird die Verbindung geschlossen.

java.net.Socket

leftShift() – Operator <<

```
java.io.Writer leftShift (Object value)
```

Erzeugt einen Writer auf dem Socket und schreibt das Argument als String in den Writer. Der Writer wird als Ergebnis zurückgegeben, so dass die Operation effizient verkettet werden kann.

```
java.io.OutputStream leftShift (byte[] value)
```

Erzeugt einen `OutputStream` auf dem Socket und schreibt das Argument in den Stream. Der Stream wird als Ergebnis zurückgegeben, so dass die Operation effizient verkettet werden kann.

withStreams()

```
void withStreams { InputStream in, OutputStream out -> void }
```

Erzeugt einen `InputStream` und einen `OutputStream` auf dem Socket und übergibt beide an die Closure. Beide Streams werden in jedem Fall danach geschlossen.

java.net.URL

eachByte()

```
void eachByte { Byte byte -> void }
```

Erzeugt eine Verbindung zu der URL und ruft die angegebene Closure für jedes Byte des Streams auf. Der Stream wird am Ende geschlossen.

eachLine()

```
void eachLine { String zeile -> void }
```

Erzeugt eine Verbindung zu der URL, liest diese zeilenweise und ruft mit jeder Zeile die angegebene Closure auf. Verwendet die Standardkodierung. Der Stream wird am Ende geschlossen.

getText()

```
String getText ()
String getText (String zeichensatz)
```

Erzeugt eine Verbindung zu der URL und liest den gesamten Inhalt in einen String. Wenn der Zeichensatz angegeben ist, wird dieser für die Kodierung verwendet, andernfalls wird die Standardkodierung benutzt. Der Stream wird danach geschlossen.

withReader() `void withReader { BufferedReader in -> void }`

Erzeugt einen `BufferedReader` zum Lesen der URL und übergibt diesen an die angegebene Closure. Dabei wird sichergestellt, dass der Reader in jedem Fall auch wieder geschlossen wird. Es wird die Standardkodierung angewendet.

Package java.sql

java.sql.Date

Siehe `java.utilDate`.

Package java.util

Zu java.util.Collection

Siehe auch oben »Iterative Methoden«.

asImmutable() `java.util.Collection asImmutable()`

Übergibt die Collection als unveränderbare Collection.

asList() `java.util.List asList()`

Erzeugt eine neue `List`-Instanz mit dem Inhalt der Collection. Wenn die Collection bereits eine Liste ist, wird sie selbst zurückgegeben.

asSynchronized()

`java.util.Collection asSynchronized()`

Übergibt die Collection als synchronisierte (Thread-sichere) Collection.

count() `int count (Object value)`

Zählt, wie viele Elemente gleich dem Argument sind.

disjoint() `boolean disjoint(java.util.Collection right)`

Prüft, ob sich die aktuelle und die als Argument übergebene Collection nicht überschneiden, also keine gleichen Elemente haben.

getAt() – Operator: [ ] (Index lesend)

`java.util.List getAt (String property)`

Durchläuft alle Elemente der Collection, fragt bei jedem Element die Property mit dem angegebenen Namen ab und liefert eine Liste mit den Werten dieser Properties.

groupBy()

```
java.util.Map<Object,List> groupBy { Object element -> Object }
```

Gruppiert die Elemente der Collection entsprechend den Ergebnissen, die jeder Aufruf der Closure liefert.

intersect()

```
java.util.List intersect (java.util.Collection zweiteCollection)
```

Erstellt eine Liste mit allen Elementen, die sowohl in der aktuellen als auch in der als Argument übergebenen Collection vorkommen.

isCase()

```
boolean isCase(java.lang.Object kandidat)
```

Implementiert die `switch-case`-Verzweigung. Prüft, ob `kandidat` in der aktuellen Collection enthalten ist..

leftShift – Operator: <<

```
java.util.Collection leftShift (Object value)
```

Fügt das Argument der aktuellen Collection hinzu. Die aktuelle Collection wird als Ergebnis zurückgeliefert, so dass sich die Operation leicht verketten lässt.

max()

```
Object max()
Object max(java.util.Comparator comparator)
Object max { Object element -> Object }
Object max { Object e1, Object e2 > int }
```

Liefert den größten Wert in der Collection. Wenn ein `Comparator` angegeben ist, wird dieser für den Vergleich verwendet. Wenn eine Closure angegeben ist, wird diese für den Vergleich verwendet. Hat die Closure einen Parameter, wird das Ergebnis der Closure für den Vergleich verwendet. Hat sie zwei Parameter, muss sie einen Vergleich zwischen diesen beiden Objekten durchführen und das Ergebnis liefern (–1, 0, +1). Bei der argumentlosen Variante von `max()` wird ein typspezifischer Vergleich durchgeführt.

min()

```
Object min()
Object min(java.util.Comparator comparator)
Object min { Object element -> Object }
Object min { Object e1, Object e2 > int }
```

Liefert den kleinsten Wert in der Collection. Wenn ein `Comparator` angegeben ist, wird dieser für den Vergleich verwendet. Wenn eine Closure angegeben ist, wird diese für den Vergleich verwendet. Hat die Closure einen Parameter, wird das Ergebnis der Closure für den Vergleich verwendet. Hat sie zwei Parameter, muss sie einen Vergleich zwischen diesen beiden Objekten durchführen und das Ergebnis liefern (–1, 0, +1). Bei der argumentlosen Variante von `min()` wird ein typspezifischer Vergleich durchgeführt.

multiply() – Operator: *

```
List multiply (Number faktor)
```

Erzeugt eine Liste, in der alle Elemente der Collection entsprechend dem angegebenen Faktor mehrfach vorhanden sind.

plus() – Operator: +

```
Collection plus (Collection right)
Collection plus (Object right)
```

Erzeugt eine neue Collection, in der die Elemente der aktuellen Collection sowie das Argument bzw. – wenn das Argument ebenfalls eine Collection ist – die Elemente des Arguments enthalten sind. Wenn das aktuelle Objekt ein Set ist, ist auch das Ergebnis ein Set. Andernfalls ist das Ergebnis eine Liste.

sort()

```
List sort()
List sort(java.util.Comparator comparator)
List sort { Object element -> Object }
List sort { Object e1, Object e2 > int }
```

Liefert eine Liste, in der die Elemente der aktuellen Collection sortiert enthalten sind. Wenn ein `Comparator` angegeben ist, wird dieser für den Vergleich verwendet. Wenn eine Closure angegeben ist, wird diese für den Vergleich verwendet. Hat die Closure einen Parameter, wird das Ergebnis der Closure für den Vergleich verwendet. Hat sie zwei Parameter, muss sie einen Vergleich zwischen diesen beiden Objekten durchführen und das Ergebnis liefern (–1, 0, +1). Bei der argumentlosen Variante von `sort()` wird ein typspezifischer Vergleich durchgeführt.

Wenn die aktuelle Collection ein `SortedSet` ist, wird sie selbst als Ergebnis geliefert, denn in diesem Fall ist das Sortieren überflüssig.

sum()

```
Object sum()
java.lang.Object sum { (groovy.lang.Closure closure)
```

Summiert die Elemente der Collection. Wenn alle Elemente Zahlen sind, werden sie arithmetisch addiert, andernfalls werden alle Elemente in Strings umgewandelt und zusammengefügt.

Wenn eine Closure angegeben ist, werden nicht die Elemente der Collection, sondern die Ergebniswerte der Closure-Aufrufe für jedes Element summiert.

toList()

```
List toList()
```

Liefert eine neue Liste mit allen Elementen der Collection.

toListString()

```
String toListString()
```

Liefert den Inhalt der Collection als String, eingeschlossen in eckige Klammern.

toString()

```
String toString()
```

Ruft `toListString()` auf und liefert dadurch eine lesbare Form des Inhalts.

unique()

```
Collection unique()
Collection unique (java.util.Comparator comparator)
Collection unique { Object element -> Object }
Collection unique { Object e1, Object e2 > int }
```

Erzeugt eine neue Collection aus den Elementen der aktuellen Collection, in der alle doppelten Elemente entfernt sind. Wenn ein `Comparator` angegeben ist, wird dieser für den Vergleich verwendet. Wenn eine Closure angegeben ist, wird diese für den Vergleich verwendet. Hat die Closure einen Parameter, wird das Ergebnis der Closure für den Vergleich verwendet. Hat sie zwei Parameter, muss sie einen Vergleich zwischen diesen beiden Objekten durchführen und das Ergebnis liefern (–1, 0, +1). Bei der argumentlosen Variante von `unique()` wird ein typspezifischer Vergleich durchgeführt.

Wenn die aktuelle Collection ein Set ist, wird sie selbst als Ergebnis geliefert (ein Set kann keine doppelten Elemente enthalten.)

Zu java.util.Date

Alle für `java.util.Date` vordefinierten Methoden sind auch auf `java.sql.Date` anwendbar.

minus() – Operator: –

```
Date minus(int days)
```

Subtrahiert die angegebene Anzahl von Tagen und gibt das Ergebnis als neues `Date`-Objekt zurück.

next() – Operator: ++ (Post- und Präinkrement)

```
Date next()
```

Erhöht das Datum um einen Tag und gibt das Ergebnis als neues `Date`-Objekt zurück.

plus() – Operator: +

```
Date plus(int days)
```

Addiert die angegebene Anzahl von Tagen und gibt das Ergebnis als neues `Date`-Objekt zurück.

previous() – Operator: – – (Post- und Präinkrement)

```
Date previous()
```

Vermindert das Datum um einen Tag und gibt das Ergebnis als neues `Date`-Objekt zurück.

Zu java.util.Enumeration

iterator()

```
java.util.Iterator iterator()
```

Liefert einen Iterator über die Elemente der Enumeration. Damit können die iterativen Methoden und die `for`-Schleife auch auf Enumerationen angewendet werden.

Zu java.util.Iterator

iterator()

```
java.util.Iterator iterator()
```

Liefert den Iterator selbst als Ergebnis. Damit können die iterativen Methoden und die `for`-Schleife auch direkt auf Iteratoren angewendet werden.

Zu java.util.List

asImmutable()

```
java.util.List asImmutable()
```

Übergibt die Liste als unveränderbare Liste.

asSynchronized()

```
Map asSynchronized()
```

Übergibt die Liste als synchronisierte (Thread-sichere) Liste.

execute()

```
Process execute()
```

Führt einen Betriebssystembefehl aus. Dabei bildet das erste Listenelement (mit dem Index 0) den eigentlichen Befehl, und die übrigen Elemente werden als Parameter übergeben.

flatten()

```
List flatten()
```

Erzeugt eine neue Liste, die alle Elemente der aktuellen Liste enthält, dabei werden diejenigen Elemente, die Collections oder Maps sind, rekursive durch deren einzelne Elemente ersetzt.

minus()

```
List minus (Object operand)
List minus (Collection zuEntfernen)
```

Entfernt alle Elemente, die gleich dem Argument bzw. gleich einem in dem Argument enthaltenen Element sind, aus der aktuellen Liste und übergibt das Ergebnis als neue Liste.

pop()

```
Object pop()
```

Entfernt das letzte Element aus der Liste und übergibt es als Ergebnis. Dadurch lässt sich jede Liste wie ein Stack behandeln.

putAt() – Operator: [ ] (Index schreibend)

```
void putAt (Range range, Object value)
```

Entfernt die durch den Bereich indizierten Elemente aus der Liste und ersetzt sie durch das zweite Argument. Wenn Letzteres eine Collection ist, werden stattdessen die Elemente dieser Collection einzeln eingefügt.

reverse()

```
List reverse()
```

Erzeugt eine neue Liste, die die gleichen Elemente wie die aktuelle Liste enthält, jedoch in umgekehrter Reihenfolge.

reverseEach()

```
void reverseEach { Object element -> void }
```

Durchläuft alle Elemente der Liste in umgekehrter Reihenfolge und übergibt sie jeweils der Closure.

Zu java.util.Map

asImmutable()

```
Map asImmutable()
```

Übergibt die Map als unveränderbare Map. Wenn die aktuelle Map eine `SortedMap` ist, ist auch das Ergebnis eine `SortedMap`.

asSynchronized()

```
Map asSynchronized()
```

Übergibt die Map als synchronisierte (Thread-sichere) Map. Wenn die aktuelle Map eine `SortedMap` ist, ist auch das Ergebnis eine `SortedMap`.

asType() – Operator: as

```
<T> T asType (Class<T> interface)
```

Diese Methode ermöglicht es, mit der aktuellen Map ein beliebiges Interface dynamisch zu implementieren. Alle Methodenaufrufe an dem resultierenden Objekt werden in Aufrufe von Closures umgewandelt, die unter dem Namen der Methode in der Map gespeichert sein müssen.

collect()

```
List collect { Map.Entry entry -> Object }
List collect (List list) { Map.Entry entry -> Object }
List collect { Object key, Object value -> Object }
List collect (List list) { Object key, Object value -> Object }
```

Liefert eine Collection aus den Ergebnissen, die von der angegebenen Closure bei dem Aufruf für jedes Element geliefert werden. Wenn eine Liste als Argument übergeben worden ist, werden die Closure-Ergebnisse dieser hinzugefügt, andernfalls legt die Methode ein neues `List`-Objekt an.

Wenn die Closure einen Parameter hat, wird ihr der jeweilige Map-Eintrag übergeben, andernfalls erhält sie den Schlüssel und den Wert als einzelne Argumente übergeben.

each()

```
void each { Map.Entry entry -> Object }
void each { Object key, Object value -> Object }
```

Führt die angegebene Closure für jedes Element aus. Wenn die Closure einen Parameter hat, wird ihr der jeweilige Map-Eintrag übergeben, andernfalls erhält sie den Schlüssel und den Wert als einzelne Argumente übergeben.

find()

```
Object find { Object elementwert -> boolean }
```

Ermittelt das erste Element, für das die angegebene Closure `true` ergibt. Der Closure werden die Werte der Map übergeben, nicht die Schlüssel.

findAll()

```
Map findAll { Map.Entry eintrag -> boolean }
Map findAll { Object schlüssel, Object wert -> boolean }
```

Erzeugt eine neue Map aus allen Elementen, für die der Closure-Aufruf `true` ergibt. Wenn die Closure einen Parameter hat, wird ihr der jeweilige Map-Eintrag übergeben, andernfalls erhält sie den Schlüssel und den Wert als einzelne Argumente übergeben.

get()

```
Object get (Object schlüssel, Object vorgabeWert)
```

Zusätzliche Variante der bei Maps ohnehin vorhandenen `get()`-Methode. Wenn in der Map kein Wert mit dem angegebenen Schlüssel vorhanden ist, wird stattdessen der angegebene Vorgabewert geliefert.

getAt() – Operator: [ ] (Index lesend)

```
Object getAt (Object schlüssel)
```

Leitet den Aufruf an die normale `get()`-Methode der Map weiter. Diese Methode ermöglicht es, die Werte einer Map auch mit der Indexnotation (eckige Klammern) auszulesen.

putAt() – Operator: [ ] (Index schreibend)

```
Object putAt (Object schlüssel, Object wert)
```

Leitet den Aufruf an die normale `put()`-Methode der Map weiter. Diese Methode ermöglicht es, die Werte einer Map auch mit der Indexnotation (eckige Klammern) zu setzen.

spread()

```
groovy.lang.SpreadMap spread()
```

Erzeugt eine `java.lang.SpreadMap` mit dem Inhalt der aktuellen Map.

subMap()

```
Map subMap (Collection schlüssel)
```

Erzeugt eine neue Map, die alle Werte der aktuellen Map umfasst, deren Werte in der als Argument erhaltenen Collection vorkommen.

toMapString()

```
String toMapString()
```

Liefert den Inhalt der Map als String, eingeschlossen in eckigen Klammern.

toSpreadMap()

```
groovy.lang.SpreadMap toSpreadMap()
```

Erzeugt eine `java.lang.SpreadMap` mit dem Inhalt der aktuellen Map.

toString()

```
String toString()
```

Ruft `toMapString()` auf und liefert dadurch eine lesbare Form des Inhalts.

java.util.Set

asImmutable()

```
Set asImmutable()
```

Übergibt das Set als unveränderbares Set. Wenn das aktuelle Set ein `SortedSet` ist, ist auch das Ergebnis ein `SortedSet`.

asSynchronized()

```
Set asSynchronized()
```

Übergibt das Set als synchronisiertes (Thread-sicheres) Set. Wenn das aktuelle Set ein `SortedSet` ist, ist auch das Ergebnis ein `SortedSet`.

flatten()

```
List flatten()
```

Erzeugt ein neues Set, das alle Elemente des aktuellen Sets enthält, dabei werden diejenigen Elemente, die Collections oder Maps sind, rekursive durch deren einzelne Elemente ersetzt.

minus()

```
Set minus (Object operand)
Set minus (Collection zuEntfernen)
```

Entfernt alle Elemente, die gleich dem Argument bzw. gleich einem in dem Argument enthaltenen Element sind, aus dem aktuellen Set und übergibt das Ergebnis als neues Set.

Zu java.util.Timer

runAfter() `void runAfter (int millis) { null -> void }`

Beauftragt den Timer, die angegebene Closure nach einer bestimmten Anzahl von Millisekunden auszuführen.

Package java.util.regex

Zu java.util.regex.Matcher

each() `void each { String[] matchgroups -> void }`

Durchläuft alle Treffer eines Mustervergleichs. Der Closure werden jeweils die einzelnen Match-Gruppen als Argumente übergeben.

getAt() – Operator: [ ] (Index lesend)

`Object getAt (int index)`

Liefert den Treffer mit dem angegebenen Index aus einem Mustervergleich. Bei einem einfachen Vergleichsmuster ist das Ergebnis ein String, bei einem gruppierten Vergleichsmuster ist das Ergebnis eine Liste mit Match-Gruppen.

`String getAt (Collection indices)`

Liefert die Treffer mit den angegebenen Indizes aus einem Mustervergleich als verketteten String. Die Elemente der Collection können einzelne Indexzahlen sein; in diesem Fall werden die gesamten Übereinstimmungen hinzugefügt. Es können aber auch ihrerseits wieder Collections oder Intervalle sein; dann werden die entsprechenden Match-Gruppen jedes Treffers hinzugefügt.

getCount() `int getCount()`

Liefert die Anzahl der Übereinstimmungen nach einem Mustervergleich.

hasGroup() `boolean hasGroup()`

Prüft, ob der `Matcher` Gruppierungen enthält.

iterator() `java.util.Iterator<String> iterator()`

Liefert einen Iterator über die Treffer des Mustervergleichs. Damit können die iterativen Methoden und die `for`-Schleife auch auf `Matcher` angewendet werden.

setIndex() `void setIndex (int index)`

Setzt die Position des `Matcher` auf den angegebenen Index.

size() `long size()`

Liefert die Anzahl der Übereinstimmungen nach einem Mustervergleich.

java.util.regex.Pattern

isCase() `boolean isCase (Object kandidat)`

Implementiert die `switch-case`-Verzweigung. Prüft, ob der Mustervergleich zwischen `kandidat` und dem aktuellen Pattern ein positives Ergebnis hat.

Package org.w3c.dom

org.w3c.dom.NodeList

iterator() `java.util.Iterator iterator()`

Liefert einen Iterator über die `NodeList`. Damit können die iterativen Methoden und die `for`-Schleife auch auf `NodeList` angewendet werden.

ANHANG B

Wichtige Klassen und Interfaces

Die Groovy-eigenen Bibliotheken definieren insgesamt über 600 verschiedene Typen – wobei eingebettete Typen nicht mitgerechnet sind. Wer mit Groovy arbeitet, muss sie natürlich nicht alle kennen. Viele von ihnen sind nur für den Compiler interessant, haben nur interne Bedeutung oder einen sehr speziellen Zweck. Ein paar Klassen und Methoden – vor allem in den Packages `groovy.lang` und `groovy.util` – sollte man aber durchaus kennen, die wichtigsten sind daher zusammenfassend dokumentiert, wobei wir einige für Anwendungsentwickler weniger interessante Dinge auslassen. Weitere Informationen zu den Klassen können der Originaldokumentation – oder, da letztere leider sehr lückenhaft ist, dem Quellcode entnommen werden.

Die Deklaration der Klassen und Interfaces ist hier, obwohl sie alle in Java implementiert sind, wie bei Groovy-Typen dargestellt und stimmt nicht immer mit der Orginaldeklaration überein. Insbesondere sind Typinformationen häufig spezifischer angegeben, um deutlich zu machen, was für Datentypen von Methoden, Konstruktoren und Properties tatsächlich erwartet bzw. geliefert werden. Außerdem sind die deklarierten Exceptions der Methoden nicht immer aufgeführt.

Package groovy.inspect

Dieses Package enthält gegenwärtig nur eine einzige relevante Klasse, nämlich den Groovy-Inspector.

Klassen

Inspector

```
class Inspector
```

Der Groovy-Inspector ermöglicht es, alle möglichen Informationen über ein Objekt in einer einheitlichen Weise zu ermitteln. Alle Ergebnisse sind in Textform dargestellt, und es werden auch dynamisch zugeordnete Methoden berücksichtigt. Daher kann der `Inspector` auch nur im Zusammenhang mit einem konkreten Objekt verwendet werden.

Konstruktor

`Inspector (Object dasObject)`

Erzeugt eine neue `Inspector`-Instanz für das angegebene Objekt.

Properties

`String[] classProps`

(Nur zu lesen.) Die Eigenschaften der Klasse, die das Objekt implementiert, in der Form eines String-Array. Die Elemente des String-Arrays können mit Hilfe einer Reihe von Integer-Konstanten der Form `CLASS_`*`XXX`*`_IDX` indiziert werden, die in `Inspector` definiert sind.

`boolean groovy`

(Nur zu lesen.) Ist `true`, wenn das zugeordnete Objekt ein Groovy-Objekt ist (also das Interface `groovy.lang.GroovyObject` implementiert).

`String[][] methods`

(Nur zu lesen.) Die Eigenschaften aller normalen Java-Methoden und -Konstruktoren der Klasse in der Form eines Arrays von String-Arrays. Die Elemente des String-Arrays können mit Hilfe einer Reihe von Integer-Konstanten der Form `MEMBER_`*`XXX`*`_IDX` indiziert werden, die in `Inspector` definiert sind.

`String[][] metaMethods`

(Nur zu lesen.) Die Eigenschaften aller dynamisch hinzugefügten Methoden der Klasse bzw. des Objekts in der Form eines Arrays von String-Arrays. Die Elemente des String-Arrays können mit Hilfe einer Reihe von Integer-Konstanten der Form `MEMBER_`*`XXX`*`_IDX` indiziert werden, die in `Inspector` definiert sind.

`Object object`

(Nur zu lesen.) Das dem `Inspector` zugeordnete Objekt.

`String[][] publicFields`

(Nur zu lesen.) Die Eigenschaften aller öffentlichen Felder und Konstanten der Klasse in der Form eines Arrays von String-Arrays. Die Elemente des String-Arrays können mit Hilfe einer Reihe von Integer-Konstanten der Form `MEMBER_`*`XXX`*`_IDX` indiziert werden, die in `Inspector` definiert sind.

`String[][] propertyInfo`

(Nur zu lesen.) Die Eigenschaften aller statischen und dynamisch hinzugefügten Properties der Klasse bzw. des Objekts in der Form eines Arrays von String-Arrays. Die Elemente des String-Arrays können mit Hilfe einer Reihe von Integer-Konstanten der Form `MEMBER_XXX_IDX` indiziert werden, die in `Inspector` definiert sind.

Methoden

`static void print (String[][] memberInfo)`

Gibt den Inhalt einer `Inspector`-Property mit Member-Informationen formatiert in der Standardausgabe aus.

`static String shortName (Class eineKlasse)`

Ermittelt den einfachen Klassennamen (ohne Package-Pfad) der übergebenen Klasse.

`static List<String[]> sort (List<String[]> memberInfo)`

Sortiert eine Liste mit Member-Informationen, wie sie vom Inspector geliefert werden, alphabetisch, und liefert dieselbe Liste als Ergebnis zurück.

Package groovy.lang

Das Package `groovy.lang` enthält ungefähr analog zu `java.lang` jene Klassen, die von der Sprache Groovy und der zugehörigen Laufzeitumgebung benötigt werden, um überhaupt funktionieren zu können, oder die ansonsten von grundlegender Bedeutung sind.

Interfaces

GroovyInterceptable

`interface GroovyInterceptable extends GroovyObject`

Marker-Interface ohne eigene Properties und Methoden. Bei Groovy-Klassen, die dieses Interface implementieren, werden alle Methodenaufrufe über die Methode `invokeMethode()` geleitet, so dass sie innerhalb der Klasse abgefangen werden können.

GroovyObject

`interface GroovyObject`

Dieses Interface implementieren alle vom Groovy-Compiler übersetzten Klassen. Es kann auch von Java-Klassen implementiert werden, die wie Groovy-Objekte behandelt werden sollen; allerdings bietet sich in diesem Fall die Ableitung von der abstrakten Klasse `GroovyObjectSupport` an, die bereits eine Standardimplementierung von `GroovyObject` enthält.

Properties

`MetaClass metaClass`

Referenz auf die für die implementierende Klasse zuständige Metaklasse. Normalerweise ist dies die in der `MetaClassRegistry` für diese Klasse registrierte Metaklasse; indem Sie hier eine andere Metaklasse zuordnen, können Sie das Verhalten des Objekts maßgeblich beeinflussen.

Methoden

`Object getProperty (String name)`

Liefert den Wert der Property mit dem angegebenen Namen.

`Object invokeMethod (String name, Object args)`

Diese Methode wird innerhalb eines Groovy-Programms immer dann aufgerufen, wenn für den jeweiligen Aufruf keine statische Implementierung vorhanden ist, so dass der Methodenaufruf dynamisch behandelt werden kann. Wenn das Objekt zusätzlich das Marker-Interface `GroovyInterceptable` implementiert, wird jeder Methodenaufruf an `invokeMethod()` geleitet. Der Parameter `name` enthält den Methodennamen und der Parameter `args` die Argumente – normalerweise in der Form eines `Object`-Arrays.

`void setProperty (String name, Object wert)`

Setzt die Property mit dem angegebenen Namen auf den angegebenen Wert.

GroovyResourceLoader

`interface GroovyResourceLoader`

Mit Hilfe einer Implementierung dieses Interface kann die Art und Weise beeinflusst werden, wie Groovy die Quelldateien eines Programms lädt. Dazu muss eine Instanz dieser Implementierung beim `GroovyClassLoader` gesetzt werden. Standardmäßig ist das Interface im `GroovyClassLoader` intern implementiert.

Methoden

`URL loadGroovySource (String filename) throws MalformedURLException`

Die zu implementierende Methode erhält den Namen einer Quelldatei und liefert die URL, die von Groovy zum Einlesen der Quelle verwendet werden kann. Das Ergebnis ist `null`, wenn die Quelldatei nicht existiert oder wenn die Rechte zum Lesen der Datei nicht ausreichen.

Interceptor

`interface Interceptor`

Implementierungen dieses Interface können im Zusammenhang mit der `ProxyMetaClass` dazu verwendet werden, Methoden- und Property-Aufrufe abzufangen.

Methoden

`Object afterInvoke (Object object, String name, Object[] args, Object result)`

Wird nach der Originalmethode aufgerufen, wenn `doInvoke()` das Ergebnis `true` hat, und andernfalls direkt nach `beforeInvoke()`. Die Parameter sind das Objekt, an dem die Originalmethode aufgerufen wird, der Name der Methode, die Argumente des Aufrufs sowie das Ergebnis der Originalmethode bzw. das Ergebnis von `beforeInvoke()`.

`Object beforeInvoke (Object object, String name, Object[] args)`

Diese Methode wird vor dem Aufruf der Zielmethode aufgerufen. Das Ergebnis ersetzt das Ergebnis der Originalmethode, wenn `doInvoke()` den Wert `false` liefert.

`boolean doInvoke()`

Das Ergebnis dieser Methode entscheidet darüber, ob die Originalmethode überhaupt aufgerufen wird. Ist es `false`, unterbleibt der Aufruf der Originalmethode, und deren Ergebnis wird durch das Ergebnis von `beforeInvoke()` ersetzt.

MetaClass

`interface MetaClass extends MetaObjectProtocol`

Interface für Metaklassen, die das Verhalten von Groovy- und Java-Objekten wesentlich beeinflussen. In der Regel wird die Standardimplementierung `MetaClassImpl` angewendet.

Dieses Interface definiert zahlreiche Properties und Methoden, von denen viele nur für die interne Implementierung wesentlich sind; der vom Anwendungsprogrammierer sinnvoll nutzbare Teil des Interface ist in `MetaObjectProtocol` definiert und dort erläutert.

MetaClassRegistry

`interface MetaClassRegistry`

Interface für ein Singleton-Objekt, bei dem zu jeder im Programm vorkommenden Klasse die zugehörige Metaklasseninstanz registriert ist. Die im aktuellen Programm gültige Instanz kann mithilfe der statischen Methode `GroovySystem.getMetaClassRegistry()` ermittelt werden.

Methoden

`MetaClass getMetaClass (Class klasse)`

Liefert die zu einer bestimmten Klasse registrierte Metaklasseninstanz.

`void removeMetaClass (Class klasse)`

Entfernt die der angegebenen Klasse aktuell zugeordnete Metaklasseninstanz. Danach muss ein Aufruf von `getMetaClass()` wieder die Standard-Metaklasse liefern.

`void setMetaClass (Class klasse, MetaClass metaKlasse)`

Setzt die Metaklasseninstanz zu einer bestimmten Klasse.

MetaObjectProtocol

`interface MetaObjectProtocol`

Dieses Interface definiert die Methoden und Properties einer Metaklasse, die für Anwendungsprogrammierer sinnvoll nutzbar sind.

Properties

`Class theClass`

Nur lesend. Die Klasse, für die die aktuelle Metaklasseninstanz zuständig ist.

Methoden

`Object getAttribute (Object objekt, String name)`

Ruft das Attribut mit dem angegebenen Namen bei dem Objekt ab. Das Objekt muss eine Instanz der Klasse sein, der die aktuelle Metaklasse zugeordnet ist. Was geschieht, wenn das Attribut nicht existiert, hängt von der zugeordneten Klasse ab.

`Object getProperty (Object objekt, String name)`

Ruft die Property mit dem angegebenen Namen bei dem Objekt ab. Das Objekt muss eine Instanz der Klasse sein, der die aktuelle Metaklasse zugeordnet ist. Was geschieht, wenn die Property nicht existiert, hängt von der zugeordneten Klasse ab.

`Object invokeConstructor (Object[] args)`

Ruft den Konstruktor der zugeordneten Klasse mit den im Array `args` enthaltenen Argumenten auf. Das Ergebnis ist eine Instanz dieser Klasse.

`Object invokeMethod (Object objekt, String name, Object[] args)`

Ruft die Methode mit dem angegebenen Namen mit den im Array angegebenen Argumenten an dem Objekt auf. Das Objekt muss eine Instanz der Klasse sein, der die aktuelle Metaklasse zugeordnet ist. Wenn die Methode nicht existiert, wird eine `MissingMethodException` ausgelöst.

`Object invokeMethod (Object objekt, String name, Object arg)`

Ruft die Methode mit dem angegebenen Namen mit einem einzelnen Argument an dem Objekt auf. Das Objekt muss eine Instanz der Klasse sein, der die aktuelle Metaklasse zugeordnet ist. Wenn die Methode nicht existiert, wird eine `MissingMethodException` ausgelöst.

`Object invokeStaticMethod(Object objekt, String name, Object[] args)`

Ruft die statische Methode mit dem angegebenen Namen mit den im Array angegebenen Argumenten auf. Wenn die Methode nicht existiert, wird eine `MissingMethodException` ausgelöst.

`void setAttribute (Object objekt, String name, Object wert)`

Setzt das Attribut mit dem angegebenen Namen bei dem Objekt. Das Objekt muss eine Instanz der Klasse sein, der die aktuelle Metaklasse zugeordnet ist. Was geschieht, wenn das Attribut nicht existiert, hängt von der zugeordneten Klasse ab.

`void setProperty (Object objekt, String name, Object wert)`

Setzt die Property mit dem angegebenen Namen bei dem Objekt. Das Objekt muss eine Instanz der Klasse sein, der die aktuelle Metaklasse zugeordnet ist. Was geschieht, wenn die Property nicht existiert, hängt von der zugeordneten Klasse ab.

MutableMetaClass

```
interface MutableMetaClass
```

Definiert zusätzliche Methoden für eine Metaklasse, der dynamisch Methoden hinzugefügt werden können. Dies sollte allerdings nur so lange geschehen, wie die Methode `initialize()` der Metaklasse nicht aufgerufen worden ist – andernfalls müssen besondere Maßnahmen bezüglich Thread-Sicherheit getroffen werden.

Methoden

`void addMetaBeanProperty (MetaBeanProperty metaBeanProperty)`

Fügt der Metaklasse eine neue Meta-Property hinzu. Die Meta-Property ist ein von der abstrakten Klasse `MetaBeanProperty` abgeleitetes Objekt, das eine Property mit ihren Getter- und Setter-Methoden beschreibt.

`void addMetaMethod (MetaMethod metaMethode)`

Fügt der Metaklasse eine neue Metamethode hinzu. Die Metamethode ist ein Objekt, mit dem eine konkrete Methode beschrieben wird.

`void addNewInstanceMethod (Method methode)`

Fügt der Metaklasse eine neue Instanzmethode hinzu. Diese Methode kann eine Originalmethode der zugeordneten Klasse überschreiben.

`void addNewStaticMethod(Method method)`

Fügt der Metaklasse eine neue statische Methode hinzu. Diese Methode kann eine Originalmethode der zugeordneten Klasse überschreiben.

PropertyAccessInterceptor

```
interface PropertyAccessInterceptor extends Interceptor
```

Erweitert das Interface `Interceptor` um Methoden, mit deren Hilfe das Auslesen und Setzen von Properties bei einer `ProxyMetaClass` abgefangen werden kann.

Methoden

`public Object beforeGet (Object objekt, String name)`

Wird vor dem Auslesen einer Property mit dem angegebenen Namen bei dem Objekt ausgeführt. Das Ergebnis dieser Methode ersetzt den Wert der Original-Property, wenn `doGet()` den Wert `false` liefert.

`public void beforeSet (Object objekt, String name, Object wert)`

Wird vor dem Setzen einer Property mit dem angegebenen Namen bei dem Objekt ausgeführt.

Range

```
public interface Range extends List
```

Interface zur Repräsentation eines Wertebereichs, also einer Zahlenfolge mit dem Abstand 1, die durch ihre Unter- und Obergrenze definiert ist. Das Interface wird durch

verschiedene Arten von Wertebereichen, z.B. `IntRange`, `EmptyRange` und `ObjectRange`, implementiert.

Properties

`Comparable from`

Nur zu lesen. Definiert den Startwert des Wertebereichs.

`Comparable To`

Nur zu lesen. Definiert den Endwert des Wertebereichs.

`boolean reverse`

Nur zu lesen. Hat den Wert `true`, wenn der Wertebereich umgekehrt ist, also wenn der Startwert größer als der Endwert ist.

Methoden

`String inspect()`

Liefert eine textuelle Darstellung des Wertebereichs in der Form, wie es in einem Programm definiert würde.

Writable

`interface Writable`

Interface für Klassen, die eine effiziente direkte Ausgabe ihres Inhalts ermöglicht. Damit kann bei komplexeren Klassen der Aufbau langer Strings mit der `toString()`-Methode vermieden werden, da die Objekte direkt als Text in den Writer geschrieben werden.

Methoden

`Writer writeTo (Writer out)`

Schreibt den Inhalt des Objekts in den angegebenen Writer.

Klassen

BenchmarkInterceptor

`class BenchmarkInterceptor implements Interceptor`

Beispiel für die Implementierung eines `Interceptor` für die `ProxyMetaClass`. Speichert die Ausführungszeiten von Methodenaufrufen und ermöglicht einfache Auswertungen.

Properties

`Map calls`

Nur zu lesen. Eine Map mit den gespeicherten Laufzeiten.

Methoden

`List statistics()`

Liefert die Ergebnisse in tabellarischer Anordnung.

`void reset()`

Setzt die gespeicherten Laufzeitwerte zurück auf `null`.

Binding

```
class Binding extends GroovyObjectSupport
```

Jedes Skript verfügt über ein Behälterobjekt vom Typ `Binding` zum Speichern von Skriptvariablen. Alle undeklarierten, nicht lokalen Variablen eines Skripts (also jene Variablen, die weder durch eine Typangabe noch durch das Schlüsselwort `def` zu lokalen Variablen der Skriptmethode erklärt sind) werden über das Binding aufgelöst und bei Bedarf neu angelegt. Das Binding kann außerhalb des Skripts von einem aufrufenden Programm bereits mit Werten belegt worden sein, und nach der Beendigung des Skriptlaufs ist es zum Auslesen der Variablen verfügbar.

Da Binding die Klasse `GroovyObjectSupport` erweitert, stehen alle Properties und Methoden eines standardmäßigen Groovy-Objekts zur Verfügung.

Konstruktoren

```
Binding ()
Binding (Map<String,Object> variablen)
```

Legt eine neue `Binding`-Instanz an. Wenn eine Map übergeben wird, sollte diese nur Werte mit String-Schlüssel enthalten, da dies die Namen der Variablen sind. Der gesamte Inhalt der Map wird in das Binding übernommen. Wird keine Map übergeben, ist das neue Binding leer.

```
Binding (String [] args)
```

Legt eine neue Binding-Instanz an, die das String-Array als Variable mit dem Namen `args` enthält.

Properties

Neben den Properties des Groovy-Objekts und den im Folgenden aufgeführten spezifischen Properties sind nachrangig auch alle im Binding gespeicherten Variablen als Properties verfügbar.

```
Map<String,Object> variables
```

Nur zu lesen. Interne Map, die zum Speichern der Binding-Variablen verwendet wird.

Methoden

```
Object getProperty (String name)
```

Überschreibt die standardmäßige Methode so, dass die Binding-Variablen auch als Properties der `Binding`-Instanz oder mit dem Indexoperator ausgelesen werden können.

```
Object getVariable (String name)
```

Liefert den Wert der Binding-Variablen mit dem angegebenen Namen. Wenn es keine Variable mit diesem Namen gibt, ist das Ergebnis `null`.

```
void setProperty (String name)
```

Überschreibt die standardmäßige Methode so, dass die Binding-Variablen auch als Properties der `Binding`-Instanz oder mit dem Indexoperator gesetzt werden können.

`void setVariable (String name, Object wert)`

Setzt eine Binding-Variable mit dem angegebenen Namen und dem angegebenen Wert. Wenn bereits eine Variable mit dem Namen vorhanden ist, wird sie überschrieben.

Closure

Abstrakte Basisklasse für alle vom Compiler übersetzten Closures. In der abgeleiteten Klasse muss die eigentliche Funktionalität der Closure in einer Methode namens `doCall()` mit beliebigen Parametern implementiert sein. Die Member der Closure-Instanz stehen innerhalb der `doCall()`-Methode im Allgemeinen nicht zur Verfügung, da diese im Kontext des Eigentümer- bzw. Delegate-Objekts läuft.

Deklaration

```
public abstract class Closure
extends GroovyObjectSupport
implements Cloneable, Runnable
```

Properties

`Object delegate`

Delegate-Objekt. An dieses Objekt werden alle Methodenaufrufe innerhalb der Closure weitergeleitet, die es nicht schon im Eigentümer-Objekt gibt. Ist anfänglich identisch mit dem Eigentümer-Objekt.

`int directive`

Property zum Steuern des Verhaltens in Schleifen. Es kann einer der Konstanten `DONE` und `SKIP` gesetzt werden. Die Auswertung dieser Flags findet in den rekursiven Methoden statt.

`int maximumNumberOfParameters`

Nur zu lesen. Anzahl der deklarierten Parameter der Closure, dieser Wert ist immer mindestens 1.

`Object owner`

Nur zu lesen. Eigentümer-Objekt der Closure. Im Kontext dieses Objekts ist die Closure erzeugt worden, und an dieses Objekt werden alle Methodenaufrufe geleitet. Wenn die Closure in einer statischen Methode erzeugt worden ist, ist das Eigentümer-Objekt die `Class`-Instanz.

`Class[] parameterTypes`

Nur zu lesen. Typen der deklarierten Parameter der Closure.

Methoden

`Writable asWritable ()`

Liefert ein Wrapper-Objekt zu dieser Closure, die das Interface `Writable` implementiert. Die zugehörige `writeTo()`-Methode ruft die Closure auf und übergibt ihr den Writer als Argument.

```
Object call ()
Object call (Object args)
Object call (Object[] args)
```

Ruft die Closure ohne Argumente, mit einem Argument oder mit einem Array von Argumenten auf.

```
Closure curry (Object[] args)
```

Liefert eine neue Closure mit derselben Funktionalität, allerdings ist ein Teil der Parameter der ursprünglichen Closure mit den Argumenten der `curry()`-Methode belegt, so dass die neue Closure nur noch die unbelegten Parameter annimmt.

```
Object getProperty (String name)
```

Versucht, die Property zuerst beim Eigentümer-Objekt und, wenn dies nicht möglich ist und es ein Delegate-Objekt gibt, beim Delegate-Objekt zu lesen.

```
boolean isCase (Object kandidat)
```

Implementierung der `switch-case`-Verzweigung. Ruft die Closure-Methode mit dem Argument auf und wandelt das Ergebnis in einen Booleschen Wert.

```
void run ()
```

Implementierung des Interface `Runnable`; ermöglicht es, eine Closure auch als eigenen Thread zu starten. Der Aufruf wird einfach an die Closure-Methode weitergeleitet.

```
void setProperty (String name, Object wert)
```

Versucht, die Property zuerst beim Eigentümer-Objekt und, wenn dies nicht möglich ist und es ein Delegate-Objekt gibt, beim Delegate-Objekt auf einen neuen Wert zu setzen.

DelegatingMetaClass

```
class DelegatingMetaClass implements MetaClass, MutableMetaClass
```

Implementierung einer Metaklasse, die alle Methodenaufrufe an eine andere, als Delegate dienende Metaklasse weiterleitet. Sie können von dieser Klasse eine eigene Klasse ableiten, um darin einzelne Methoden zu überschreiben und die Aufrufe abzufangen. Welche Methoden überschrieben werden können, ist den Erläuterungen zu dem Interface `MetaObjectProtokol` zu entnehmen.

Konstruktoren

```
def DelegatingMetaClass (MetaClass delegate)
```

Erzeugt eine neue `DelegatingMetaClass`, die alle Aufrufe an die als Argument übergebene Metaklasse weiterleitet.

```
def DelegatingMetaClass (Class klasse)
```

Erzeugt eine neue `DelegatingMetaClass`, die alle Aufrufe an die Metaklasse weiterleitet, die der als Argument übergebenen Klasse zugeordnet ist.

Methoden `public void setAdaptee (MetaClass delegate)`

Setzt die Metaklasse, an die alle Aufrufe weitergeleitet werden.

Weitere Methoden sind bei dem Interface `MetaObjectProtocol` zu finden.

ExpandoMetaClass

`class ExpandoMetaClass extends MetaClassImpl implements GroovyObject`

Erweiterung der standardmäßigen Metaklasse, die es erlaubt, im laufenden Betrieb neue Methoden und Properties hinzuzufügen; dies muss jedoch vor dem Aufruf von `initialize()` geschehen, erst danach kann die `ExpandoMetaClass` verwendet werden.

Methoden und Properties können der `ExpandoMetaClass` folgendermaßen zugewiesen werden:

```
def mc = new ExpandoMetaClass(MeineKlasse)
// Methode neu zuweisen
mc.methode1 << { String s -> println "Methode 1, arg=$s" }
// Methode neu zuweisen oder überschreiben
mc.methode2 = { int x, int y -> println "Methode 2, arg=$y,$y" }
// Statische Methode neu zuweisen
mc.'static'.methode3 << { String s -> println "Stat. Methode 3, arg=$s" }
// Statische Methode neu zuweisen oder überschreiben
mc.'static".methode4 = { int x, int y -> println "Stat. Methode 4, arg=$x,$y" }
// Konstruktor neu zuweisen
mc.constructor << { String s -> println "Konstruktor, arg=$s" }
// Konstruktor neu zuweisen oder überschreiben
mc.constructor = { int x, int y -> println "Konstruktor, arg=$x,$y" }
// Property mit Initialwert zuweisen
mc.property1 = "Initialwert der Property 1"
```

Siehe auch Interfaces `MetaObjectProtocol` und `GroovyObject`.

Konstruktoren `ExpandoMetaClass (Class dieKlasse, boolean registrieren=false)`

Erzeugen eine neue Instanz der `ExpandoMetaClass` für den angegebenen Typ. Wenn das Flag registrieren angegeben und auf `true` gesetzt ist, wird die neue Instanz automatisch in der `MetaClassRegistry` angemeldet.

Properties `boolean allowChangesAfterInit`

(Nur schreibend.) Legt fest, ob auch nach dem `initialize()`-Aufruf der Metaklasse noch Änderungen vorgenommen werden dürfen.

`List<MetaMethod> expandoMethods`

(Nur lesend.) Alle dynamisch hinzugefügten Methoden, beschrieben in der Form einer Liste von `MetaMethod`-Objekten.

`List<MetaBeanProperty> expandoProperties`

(Nur lesend.) Alle dynamisch hinzugefügten Properties, beschrieben in der Form einer Liste von `MetaBeanProperty`-Objekten.

`Class javaClass`

(Nur lesend.) Klasse, für diese `ExpandoMetaClass` gilt. Ist im Konstruktor übergeben worden.

Methoden

`static void disableGlobally()`

Macht den globalen Gebrauch aller `ExpandoMetaClass`-Instanzen unmöglich.

`static void enableGlobally()`

Ermöglicht den globalen Gebrauch aller `ExpandoMetaClass`-Instanzen. Dieser Aufruf ist Voraussetzung dafür, dass diese auch für Objekte abgeleiteter oder implementierender Klassen gültig sind und dass sie somit auch für abstrakte Klassen und Interfaces verwendbar sind.

```
MetaMethod getMetaMethod (String name, Class argTypen)
MetaMethod getMetaMethod (String name, Object args)
```

Liefert eine Beschreibung der Methode mit dem angegebenen Namen und die angegebenen Argumente-Typen bzw. Argumente als `MetaMethod`-Instanz.

`MetaProperty getMetaProperty (String name)`

Liefert eine Beschreibung der Property mit dem angegebenen Namen als `MetaProperty`-Instanz.

GroovyClassLoader

`class GroovyClassLoader extends URLClassLoader`

Groovy-eigener Classloader, der Groovy-Klassen aus Quelltexten lesen und übersetzen kann. Der `GroovyClassLoader` kann auch in eigenen Programmen verwendet werden, um Groovy-Klasse dynamisch zu laden.

Konstruktoren

```
GroovyClassLoader()
GroovyClassLoader(ClassLoader parent)
GroovyClassLoader(ClassLoader parent, CompilerConfiguration config)
GroovyClassLoader(ClassLoader parent, CompilerConfiguration config,
    boolean useConfigClasspath)
GroovyClassLoader(GroovyClassLoader parent)
```

Als Konstruktor-Parameter kann ein Java- oder Groovy-Classloader angegeben werden, der als Parent-Classloader dient. Ist kein Classloader angegeben, so wird der Classloader des aktuellen Thread-Kontext als Parent benutzt.

Die Übergabe eines Konfigurationsobjekts vom Typ `org.codehaus.groovy.control.CompilerConfiguration` ermöglicht die Angabe verschiedener Optionen. Wenn das Flag `useConfigClasspath` gesetzt ist, wird auch der Klassenpfad aus der Konfiguration benutzt.

Properties

`Class[] loadedClasses`

(Nur lesend.) Array mit allen Klassen, die von dieser Classloader-Instanz geladen worden sind.

`Boolean shouldRecompile`

Flag, das festlegt, ob die Klassen neu kompiliert werden sollten. Wenn der Wert `null` ist, wird die Einstellung aus der `CompilerConfiguration` verwendet.

`GroovyResourceLoader resourceLoader`

Implementierung des Interface `GroovyResourceLoader`, die zum Lokalisieren von Quelltexten anhand der Klassennamen dient. Kann durch eine eigene Implementierung ersetzt werden, die beispielsweise die Programmquellen aus einer Datenbank liest.

Methoden

```
void addClasspath (String path)
```

Diese Methode ermöglicht es, dem Klassenpfad zur Laufzeit ein Pfadelement hinzuzufügen.

```
void addURL (URL url)
```

Fügt zur Laufzeit eine URL hinzu.

```
Class loadClass (String name, boolean lookup,
    boolean preferClass, boolean resolve=false)
```

Lädt die Klasse mit dem angegebenen Namen entweder selbst oder über den Parent-Classloader. Die Flags haben folgende Bedeutung: `lookup` legt fest, ob nach Skriptdateien gesucht werden soll; `preferClass` bestimmt, ob reguläre Klassen Vorrang vor Skripten haben sollen und `resolve` besagt, dass die Klasse durch den Aufruf der `resolve()`-Methode des Classloader eingebunden werden soll.

```
Class parseClass (File sourceFile)
Class parseClass (GroovyCodeSource codeSource)
Class parseClass (InputStream in, String fileName=null)
Class parseClass (String text, String fileName=null)
```

Kompiliert den angegebenen Groovy-Quelltext und liefert die Hauptklasse als Ergebnis zurück. Der Quelltext kann als Datei, als `GroovyCodeSource`-Objekt, als `InputStream` und als Klartext angegeben werden. In den letzten beiden Fällen kann ein beliebiger Dateiname als Grundlage für die Bildung von Klassennamen bei Skripten angegeben werden. Ist kein Dateiname gegeben, wird der Klassenname generiert.

GroovyCodeSource

`class GroovyCodeSource`

Klasse, die die Herkunft eines Groovy-Quellprogramms beschreibt. Wird in Zusammenhang mit dem `GroovyClassLaoder` verwendet und insbesondere dann benötigt, wenn ein Groovy-Programm mit den Mitteln der Java-Sicherheitsarchitektur abgesichert werden soll, obwohl es keine Quelladresse (URL oder Dateiname) gibt.

Konstruktoren

```
GroovyCodeSource (File file)
GroovyCodeSource (URL url)
```

Erzeugt eine `GroovyCodeSource` für eine Datei oder eine URL. In diesem Fall sind bereits alle erforderlichen Informationen in den Zielnamen enthalten.

```
GroovyCodeSource (InputStream in, String name, String codeBase)
GroovyCodesSource (String skript, String name, String codeBase)
```

Wenn das Quellprogramm aus einer unbenannten Quelle kommt, wie hier aus einem InputStream oder direkt aus einem String, müssen ein Dateiname und eine Code-Base (Verzeichnis oder Basis-URL) angegeben werden.

Properties

```
File file
```

(Nur lesend.) `File`-Objekt, wenn die Programmquelle eine Datei ist, sonst `null`.

```
InputStream inputStream
```

(Nur lesend.) `InputStream`-Objekt, wenn die Programmquelle ein Eingabestrom ist, sonst `null`.

```
String name
```

(Nur lesend.) Name der Programmquelle.

```
boolean cachable
```

Flag legt fest, ob die Programmquelle im Cache gehalten werden kann.

GroovyObjectSupport

```
abstract class GroovyObjectSupport implements GroovyObject
```

Hilfreiche Basisklasse für in Java geschriebene Klassen, die sich wie Groovy-Objekte verhalten sollen. Enthält Standardimplementierungen aller Properties und Methoden des Interface `GroovyObject`.

GroovyShell

```
class GroovyShell extends GroovyObjectSupport
```

Mit Hilfe dieser Klasse können Groovy-Skripte innerhalb eines Java- (oder Groovy-)Programms sehr einfach dynamisch übersetzt und ausgeführt werden.

Konstruktoren

```
GroovyShell (ClassLoader parentLoader, Binding binding, CompilerConfiguration config)
```

Erzeugt eine `GroovyShell` unter Angabe eines definierten Parent-Classloader, eines Binding-Objekts und einer `CompilerConfiguration`. Der Konstruktor ist mit sechs Varianten überladen, die es ermöglichen, einen oder mehrere der Argumente wegzulassen.

`GroovyShell (GroovyShell parentShell)`

Erzeugt eine `GroovyShell` mit einem eigenen Classloader, dabei wird der Classloader der Parent-Shell als Parent-Classloader verwendet.

Properties

`Binding contevxt`

Das Binding-Objekt, das den von der `GroovyShell` ausgeführten Skripten übergeben wird.

Die Methoden `getProperty()` und `setProperty()` der `GroovyShell` sind so modifiziert, dass sie zunächst die Werte im Binding lesen bzw. setzen.

Methoden

```
Object evaluate (File sourceFile)
Object evaluate (GroovyCodeSource codeSource)
Object evaluate (InputStream in, String fileName=null)
Object evaluate (String scriptText, String fileName=null)
```

Führt das in einer Datei, einem GroovyCodeSource-Objekt, einem InputStream oder einem String angegebene Groovy-Skript aus und liefert das Ergebnis zurück. Wenn die Quelle ein Strom oder ein String ist, kann ein gedachter Dateinamen angegeben werden, mit dessen Hilfe der Klassenname gebildet wird; andernfalls wird der Klassenname generiert.

```
Object getVariable (String name)
```

Liest eine Variable aus dem Binding.

```
void initializeBinding ()
```

Initialisiert das Binding. In der Standard-Implementierung wird nur eine Binding-Variable `shell` mit der aktuellen `GroovyShell`-Instanz belegt.

```
Script parse (File file)
Script parse (GroovyCodeSource codeSource)
Script parse (InputStream in, String fileName=null)
Script parse (String scriptText, String fileName=null)
```

Entspricht den `evaluate()`-Methoden, jedoch wird das übersetzte Skript nicht ausgeführt sondern als Methodenergebnis zurückgegeben. Das Skript kann mit seiner Methode `run()` ausgeführt werden und ist wiederverwendbar.

```
void resetLoadedClasses ()
```

Entfernt alle geladenen Klassen aus dem Cache des eigenen Classloader.

```
Object run (File scriptFile, List args<String>)
Object run (File scriptFile, String[] args)
Object run (InputStream in, String fileName, String[] args)
Object run (String scriptText, String fileName, List<String> args)
Object run (String scriptText, String fileName, String[] args)
```

Führt das angegebene Quellprogramm als Skript, Main-Klasse oder JUnit-Testfall aus. Das jeweils letzte Argument wird dem Programm als Array von Befehlszeilenargumenten mitgegeben.

`void setVariable (String name, Objekt wert)`

Setzt eine Variable im Binding auf den angegebenen Wert.

GroovySystem

`final class GroovySystem`

Statische Klasse mit einigen systemweit gültigen Informationen.

Properties

`static MetaClassRegistry metaClassRegistry`

(Nur lesend.) Verweist auf die in diesem System gültige `MetaClassRegistry`.

`static boolean useReflection`

(Nur lesend.) Flag, das festlegt, ob Metaklassen standardmäßig Methoden- und Property-Namen grundsätzlich mit Hilfe von Java-Reflection auflösen.

GString

`abstract class GString implements Comparable, CharSequence, Writable, Buildable`

Abstrakte Basisklasse GStrings. Für jeden GString generiert der Compiler eine eigene, von dieser Klasse abgeleitete Klasse.

Properties

`String[] strings`

(Nur lesend.) Array mit den Strings, die zusammen den Text des GString ergeben.

`Object[] valuesObject[] values`

(Nur lesend.) Array mit den Objekten, die zwischen den Textsegmenten eingefügt werden.

`int valueCount`

(Nur lesend.) Anzahl der Objekte, die zwischen den Textsegmenten einzufügen sind.

Methoden

`void build (GroovyObject builder)`

Ermöglicht es, GStrings in Builder einzufügen. Der betreffende Builder muss jedoch die Methoden

`Object getValue (int index)`

Ermittelt den Wert mit dem angegebenen Index.

An einem GString können alle Methoden eines `String`-Objekts aufgerufen werden, in der Regel werden sie einfach an eine durch `toString()` erzeugte String-Repräsentation des GString weitergeleitet.

ProxyMetaClass

```
class ProxyMetaClass extends MetaClassImpl implements AdaptingMetaClass
```

Implementierung einer Metaklasse, die alle Aufrufe an eine andere Metaklasse weiterleitet, wobei aber alle Methodenaufrufe abgefangen werden können. Der `ProxyMetaClass`-Instanz muss dazu eine Implementierung des Interface `Interceptor` zugewiesen werden, deren Methoden dann vor und nach jedem Methodenaufruf konsultiert werden.

Instanzen dieser Metaklasse sollten immer nur auf einzelne Objekten angewendet werden, da die Klasse nicht threadsicher ist.

Konstruktor

`ProxyMetaClass (MetaClassRegistry reg, Class dieKlasse, MetaClass adaptee)`

Erzeugt eine neue `ProxyMetaClass`-Instanz für die angegebene Klasse, die alle Aufrufe an die ebenfalls anzugebende eigentlich zuständige Metaklasse weiterreicht. Der erste Parameter ist die systemweit gültige `MetaClassRegistry`.

Properties

`MetaClass adaptee`

Metaklasse, an die alle Aufrufe weitergeleitet werden.

`Interceptor interceptor`

Objekt, das bei jedem Methodenaufruf konsultiert wird.

Methoden

`static ProxyMetaClass getInstance (Class eineKlasse)`

Factory-Methode, die den Normalfall einer Instanziierung abdeckt. Es genügt die Angabe einer Klasse, alle anderen Angaben ermittelt die Methode selbst.

```
void use (Closure closure)
void use (GroovyObject objekt, Closure closure)
```

Registriert die aktuelle `ProxyMetaClass`, führt die Closure aus, und deregistriert sie anschließend. Wenn ein `GroovyObject` als Argument angegeben ist, wird die Metaklasse nicht bei der `MetaClassRegistry` angemeldet, sondern nur an dem genannten Objekt gesetzt. Diese Methoden erleichtern den Einsatz der `ProxyMetaClass` für eine definierte Phase.

Script

```
abstract class Script extends GroovyObjectSupport
```

Basisklasse für alle Groovy-Skripte, also alle Programme ohne explizite Klassendefinition.

Konstruktor

`Script (Binding binding=null)`

Erzeugt eine neue Instanz der Skript-Klasse und übergibt das angegebene Binding. Wenn kein Binding-Objekt genannt ist, legt die Klasse ein eigenes an.

Properties

`Binding binding`

Binding des Skripts, mit dem alle Referenzen auf nicht deklarierte Variablen aufgelöst werden.

Die Methoden `getProperty()` und `setProperty()` im Skript sind so angepasst, dass alle Zugriffe auf Properties vorrangig an das Binding weitergeleitet werden.

Methoden

```
Object evaluate (File quelldatei)
Object evaluate (String quelltext)
```

Führt ein Groovy-Skript dynamisch unter Verwendung des aktuellen Bindings aus.

```
abstract Object run()
```

Diese Methode wird in einer konkreten, abgeleiteten Klasse durch das auszuführende Skript überschrieben.

```
void run (File quelldatei, String[] args)
```

Führt ein Groovy-Skript dynamisch unter Verwendung des aktuellen Bindings und der angegebenen Befehlszeilenargumente aus.

TracingInterceptor

```
class TracingInterceptor implements Interceptor
```

Implementierung des `Interceptor`, die alle Methodenaufrufe protokolliert. Kann in Zusammenhang mit der `ProxyMetaClass` verwendet werden.

Properties

`Writer writer`

Writer-Instanz, in die das Protokoll der Methodenaufrufe geschrieben wird. Wenn der Writer nicht explizit gesetzt wird, schreibt er in die Standardausgabe.

Package groovy.util

Interfaces

ResourceConnector

```
interface ResourceConnector
```

Durch eine Implementierung dieses Interface kann ein Programm, das die `GroovyScriptEngine` verwendet, festlegen, wie die Ressourcen gefunden werden.

Methoden

```
java.net.URLConnection getResourceConnection (String name)
```

Erzeugt ein `URLConnection`-Objekt, das den Zugriff auf die durch den Namen beschriebene Ressource ermöglicht.

Klassen

BuilderSupport

```
class BuilderSupport extends GroovyObjectSupport
```

Basisklasse für eigene Builder-Klassen.

Konstruktoren

```
BuilderSupport ()
BuilderSupport (BuilderSupport proxy)
BuilderSupport (Closure nameMappingClosure, BuilderSupport proxy)
```

Dem Konstruktor kann eine weitere `BuilderSupport`-Instanz übergeben werden, an die er dann alle nicht selbst behandelten Methodenaufrufe weiterleitet. Außerdem kann eine Closure angegeben werden, die alle Methodennamen in entsprechende andere Objekte übersetzt.

Methoden

Es sind hier nur diejenigen Methoden aufgeführt, die typischerweise durch eigene Builder-Implementierungen überschrieben oder von ihnen verwendet werden.

```
protected abstract Object createNode (Object name)
protected abstract Object createNode (Object name, Map attrib)
protected abstract Object createNode (Object name, Object value)
protected abstract Object createNode (Object name, Map attrib, Object
value)
```

Wird aufgerufen, wenn ein neuer Knoten anzulegen ist. Der Name ist normalerweise ein String mit dem Namen der aufgerufenen Methode, kann aber bei Einsatz einer Closure zur Übersetzung der Namen auch ein anderes Objekt sein.

Entsprechend dem Methodenaufruf erhält `createNode()` die übergebenen Attribute und gegebenenfalls ein Extra-Argument, jedoch nicht eine zur Kennzeichnung einer neuen Ebene verwendete Closure.

```
protected Object getCurrent()
```

Liefert den aktuellen Knoten; innerhalb von `createNode()` ist dies das Parent-Objekt des aktuell anzulegenden Knotens. Beim Anlegen des Wurzelobjekts ist der aktuelle Knoten `null`.

```
protected Object getName (String methodeName)
```

Kann überschrieben werden, um einen Methodennamen in ein beliebiges anderes Objekt zu übersetzen (alternativ zur Angabe einer Closure im Konstruktor).

```
protected void nodeCompleted (Object parent, Object knoten)
```

Wird aufgerufen, wenn die Bearbeitung eines Knotens (mit allen untergeordneten Knoten) beendet ist.

`protected Object postNodeCompletion (Object parent, Object knoten)`

Wird nach `nodeCompleted()` aufgerufen und ermöglicht es, das zurückzugebende Knotenobjekt zu modifizieren.

`protected void setParent (Object parent, Object child)`

Wird aufgerufen, um das Parent-Objekt eines Knotens zu setzen.

CharsetToolkit

`class CharsetToolkit`

Hilfsklasse, die es erlaubt, den in einer Datei verwendeten Zeichensatz zu ermitteln, und die einige weitere Zeichensatz-bezogene Funktionen aufweist.

Konstruktor

`CharsetToolkit (File file)`

Erzeugt ein neues `CharsetToolkit` für die angegebene Textdatei.

Properties

`static Charset[] availableCharsets`

(Nur lesend.) Array mit allen in der JVM bekannten Zeichensätzen.

`Charset charset`

(Nur lesend.) Für die angegebene Datei ermittelter Zeichensatz.

`Charset defaultCharset`

Standardmäßig zu verwendender Zeichensatz, wenn die Datei keine Anhaltspunkte für den verwendeten Zeichensatz liefert.

`Charset defaultSystemCharset`

(Nur lesend.) Standard-Zeichensatz der JVM.

`boolean enforce8Bit`

Legt fest, dass ein 7-Bit-Zeichensatz (US-ASCII) ausgeschlossen sein soll. (Vorgabewert ist `true`.)

Methoden

`BufferedReader getReader()`

Erzeugt einen Reader für die zugeordnete Datei unter Verwendung des ermittelten Zeichensatzes.

ClosureComparator

`class ClosureComparator implements Comparator`

Einfache, universell zu verwendende `Comparator`-Implementierung, die eine Closure für den Vergleich von zwei Objekten benutzt.

Konstruktoren

```
ClosureComparator (Closure closure)
```

Der Konstruktor übernimmt eine Closure mit zwei Parametern, die den eigentlichen Vergleich vornimmt.

Methoden

```
int compare (Object o1, Object o2)
```

Führt den Vergleich mit Hilfe der Closure aus.

Eval

```
class Eval
```

Klasse mit einigen statischen Methoden, die das Einbinden von Groovy-Code in Java-Programme erleichtern sollen.

Methoden

```
Object me (String ausdruck)
```

Führt das im Argument angegebene Stück Groovy-Code aus und gibt das Ergebnis zurück.

```
Object me (String symbol, Object  object, String ausdruck)
```

Führt den Groovy-Ausdruck aus, dieser erhält aber zusätzlich das Objekt als Variable mit dem als `symbol` angegebenen Variablennamen.

```
Object x (Object x, String ausdruck)
```

Führt den Groovy-Ausdruck aus, dieser erhält zusätzlich das Objekt `x` als Variable mit dem Namen »x«.

```
Object xy (Object x, Object y, String ausdruck)
```

Führt den Groovy-Ausdruck aus, dieser erhält zusätzlich die Objekte `x` und `y` als Variablen mit den Namen »x« und »y«.

```
Object xyz (Object x, Object y, Object z, String ausdruck)
```

Führt den Groovy-Ausdruck aus, dieser erhält zusätzlich die Objekte `x`, `y` und `z` als Variablen mit den Namen »x«, »y« und »z«.

GroovyScriptEngine

```
class GroovyScriptEngine implements ResourceConnector
```

Führt Groovy-Skripte aus, lädt dabei veränderte Skripte neu und kann auch Aufrufe von Skripten aus Skripten verarbeiten.

Konstruktoren

```
GroovyScriptEngine (ResourceConnector rc, ClassLoader parentLoader=null)
GroovyScriptEngine (String url, ClassLoader parentLoader=null)
GroovyScriptEngine (String[] urls, ClassLoader parentLoader=null)
GroovyScriptEngine (URL[] urls, ClassLoader parentLoader=null)
```

Erzeugt eine neue `GroovyScriptEngine` unter Angabe eines oder mehrerer URLs, unter denen die Quellprogramme zu finden sind, oder eines `Resource-`

Connector, der mit der Lokalisierung der Quellprogramme beauftragt werden kann. Wenn URLs angegeben sind, müssen die Skriptdateien analog zu Klassendateien unter Java angeordnet sein, so dass derDateipfad unterhalb der jeweiligen URL jeweils den Package-Pfad widerspiegelt.

In allen Fällen kann optional ein Parent-Classloader angegeben werden, wenn nicht der Standard-Classloader des aktuellen Thread verwendet werden soll.

Properties

```
CloassLoader parentClassLoader
```

(Nur lesend.) Aktueller Parent-Classloader.

Methoden

```
java.net.URLConnection getResourceConnection (String name)
```

Liefert den Zugriff auf eine Quelldatei in Form eines `URLConnection`-Objekts.

```
Object run (String skriptName, Binding binding)
```

Führt das Skript mit dem angegebenen Namen aus und gibt das Ergebnis des Skriptlaufs zurück.

```
String run (String skriptName, String argument)
```

Führt das Skript mit dem angegebenen Namen aus und gibt das Ergebnis als String zurück. Das Skript erhält ein neues Binding mit dem angegebenen Argument als Variable namens »arg«.

GroovyTestCase

```
class GroovyTestCase extends junit.framework.TestCase
```

Basisklasse für Testfälle unter Groovy. Enthält einige zusätzliche Prüfmethoden. Abgeleitete Groovy-Klassen können direkt als Skript ausgeführt werden. Es sind hier nur die zusätzlichen Methoden aufgeführt.

Methoden

```
protected void assertArrayEquals (Object[] erwartet, Object[] array)
```

Prüft ob ein Array die erwartete Länge und den erwarteten Inhalt hat.

```
protected void assertContains (char erwartet, char[] array)
protected void assertContains (int erwartet, int[] array)
```

Prüft, ob ein Array ein bestimmtes Element enthält.

```
protected void assertInspect (Object wert, String erwartet)
```

Prüft, ob der Aufruf der `inspect()`-Methode des angegebenen Wertes das erwartete Ergebnis hat.

```
protected void assertLength (int erwartet, char[] array)
protected void assertLength (int erwartet, int[] array)
protected void assertLength (int erwartet, Object[] array)
```

Prüft, ob das angegebene Array die erwartete Länge hat.

`protected void assertScript (String skriptText)`

Prüft, ob das in `skriptText` übergebene Groovy-Skript ohne Exception ausgeführt werden kann.

`protected void assertToString (Object wert, String erwartet)`

Prüft, ob der Aufruf von `toString()` bei dem übergebenen Wert das erwartete Ergebnis liefert.

`protected String fixEOLs (String wert)`

Ersetzt alle Zeilenenden im übergebenen String durch `\n`-Zeichen. Dies erleichtert den Vergleich von mehrzeiligen Ergebnissen mit String-Literalen.

`String getMethodName()`

Liefert den Namen der aktuellen Testmethode (wie `getName()` im JUnit-`TestCase`).

`String getName()`

Liefert den Namen des aktuellen Tests in etwas verschönerter Form.

`boolean notYetImplemented()`

Führt den aktuellen Test noch einmal aus und meldet einen Fehler, wenn dabei *kein* Fehler auftritt. Diese Methode erleichtert den Umgang mit noch nicht fertig gestellten Anwendungsteilen. Um sie zu nutzen, kann man folgende Zeile am Anfang eines normalen Tests einfügen:

```
if (notYetImplemented()) return
```

`static boolean notYetImplemented (junit.framework.TestCase aufrufer)`

Statische Variante von `notYetImplemented()`, erlaubt den Aufruf aus Klassen, die nicht von `GroovyTestCase` abgeleitet sind.

```
protected String shouldFail (Class exceptionTyp, Closure code)
protected String shouldFail (Closure code)
```

Prüft, ob die angegebene Closure eine Exception auslöst. Die erste Variante der Methode prüft darüber hinaus, ob die ausgelöste Exception vom angegebenen Typ ist.

GroovyTestSuite

`class GroovyTestSuite extends junit.framework.TestSuite`

Basisklasse für in Groovy geschriebene Test-Suites. Kann auch als eigenständiges Programm gestartet werden; in diesem Fall muss das Skript mit dem Testfall als Argument angegeben werden.

Die `GroovyTestSuite` kann auch direkt in einem Testrunner ausgeführt werden, wenn der auszuführende Testfall als System-Property namens »test« definiert ist.

Methoden

`Class compile (String dateiName) {`

Kompiliert die angegebene Skript-Datei mit Hilfe des `GroovyClassLoader` und gibt die erzeugte Klasse als Ergebnis zurück.

`static void main (String args[])`

Main-Methode für den Start der Test-Suite als eigenständiges Programm. Wenn ein Befehlszeilenargument angegeben ist, wird dieses als Name eines Skripts interpretiert, das den Testfall enthält.

`void loadTestSuite ()`

Lädt die zur Suite gehörenden Tests aus der Skriptdatei, die entweder als Befehlszeilenargument oder als System-Property genannt ist.

`static Test suite ()`

Erzeugt eine neue `GroovyTestSuite`-Instanz, lädt die auszuführenden Tests und gibt sie als Ergebnis zurück.

IndentPrinter

`class IndentPrinter`

Hilfsklasse zur Ausgabe von hierarchisch eingerückten Testen. Wird beispielsweise vom NodePrinter verwendet.

Konstruktoren

`IndentPrinter (PrintWriter out=System.out, String indent="  ")`

Erzeugt einen neuen `IndentPrinter` mit dem angegebenen `PrintWriter` zur Ausgabe und dem zum Einrücken zu verwendenden String.

Methoden

`void decrementIndent()`

Vermindert die Einrückungstiefe um eine Stufe.

`void flush()`

Leert den Ausgabepuffer.

`void incrementIndent()`

Erhöht die Einrückungstiefe um eine Stufe.

`void print (String text)`

Gibt den angegebenen Text ohne Zeilenvorschub aus.

`void printIndent()`

Gibt den String für die Einrückung entsprechend der aktuellen Tiefe aus.

`void println()`

Gibt einen Zeilenvorschub aus.

`void println (String text)`

Gibt den angegebenen Text mit anschließendem Zeilenvorschub aus.

`void setIndentLevel (int level)`

Setzt die aktuelle Einrückungstiefe auf den angegebenen Wert.

Node

`class Node implements Serializable`

Diese Klasse stellt einen Knoten in einer beliebigen Baumstruktur analog zu einem XML-Dokument dar. Sie wird unter anderem vom `NodeBuilder` verwendet.

Konstruktoren

`Node (Node parent, Object name, Map attribute, Object inhalt)`

Erzeugt einen neuen Knoten mit dem angegebenen Parent-Knoten, dem Namen, den Attributen und dem Inhalt. Bei einem Wurzelknoten ist der Parent-Knoten `null`. Wenn ein Parent-Knoten angegeben ist, trägt sich der neue Knoten automatisch bei diesem als neuer Kindknoten ein. Der Name ist üblicherweise ein String, kann aber auch ein beliebiges Objekt sein. Als Inhalt sollte entweder ein String oder ein `List`-Objekt mit Unterknoten verwendet werden.

Die Parameter `attribute` und `inhalt` können einzeln oder zusammen weggelassen werden.

Properties

`Object attribute (Object name)`

Liefert das Attribut mit dem angegebenen Namen. Wenn kein Attribut mit diesem Namen existiert, ist das Ergebnis `null`.

`List<Node> breadthFirst ()`

Liefert eine flache Liste aller im (Teil-)Baum enthaltenen Knoten. Dabei werden die Geschwisterknoten vor den Kindknoten ausgewertet.

`List<Node> children()`

Liefert eine Liste mit allen direkt untergeordneten Kindknoten.

`List<Node> depthFirst()`

Liefert eine flache Liste aller im (Teil-)Baum enthaltenen Knoten. Dabei werden die Kindknoten vor den Geschwisterknoten ausgewertet.

`Object get (String name)`

Liefert einen Wert mit dem angegebenen Namen. Ermöglicht den Zugriff auf diese Werte in Groovy über die Property-Notation. Abhängig von der Form des Arguments werden folgende Ergebnisse geliefert.

`@xxx` – das Attribut mit dem Namen »xxx«.

`..` (zwei Punkte) – der Parent-Knoten.

* (ein Stern) – Liste aller direkten Kindknoten.

** (zwei Sterne) – Liste aller Knoten im Baum wie bei `depthFirst()`.

`xxx` – `NodeList`-Objekt aller direkten Kindknoten mit dem Namen »xxx«.

`NodeList getAt (groovy.xml.QName qname)`

Ermittelt alle direkten Kindknoten, deren Name dem angegebenen `QName`-Objekt entspricht.

`Iterator<Node> iterator()`

Liefert einen Iterator über alle direkten Kindknoten.

`Object name()`

Gibt den Namen des Knotens zurück.

`Node parent()`

Gibt den Parent-Knoten zurück bzw. `null`, wenn der aktuelle Knoten ein Wurzelknoten ist.

`void print (PrintWriter out)`

Gibt den mit dem aktuellen Knoten beginnenden (Teil-)Baum mit Hilfe eines `NodePrinter` in strukturierter Weise auf dem angegebenen `PrintWriter` aus.

`viud setValue (Object inhalt)`

Setzt den Inhalt des aktuellen Knotens. Dabei sollte es sich entweder um einen String oder um eine Liste von Kindknoten handeln.

`String text()`

Gibt den Inhalt des aktuellen Knotens in Textform zurück.

`Object value()`

Liefert den Inhalt des aktuellen Objekts in unveränderter Form. Dies ist in der Regel ein String oder eine Liste mit Kindknoten.

NodeList

```
class NodeList extends ArrayList<Node>
```

Mengenabfragen von Knoten bei einem `Node`-Objekt liefert im allgemeinen ein Objekt dieses Typs, das eine Liste von Knotenobjekten enthält.

Konstruktoren

```
NodeList()
NodeList(Collection<Node> collection)
NodeList(int size)
```

Die Konstruktoren entsprechen den analogen Konstruktoren der `ArrayList`.

Methoden

```
NodeList getAt (groovy.xml.QName qname)
NodeList getAt (String name)
```

Liefert eine Liste mit allen enthaltenen Knoten, deren Namen dem angegebenen `QName`-Objekt bzw. dem im String angegebenen Namen entspricht.

`String text ()`

Liefert die zusammengefassten Ergebnisse des Aufrufs von `text()` bei jedem enthaltenen Knotenobjekt.

NodePrinter

`class NodePrinter`

Gibt einen aus `Node`-Objekten bestehenden Baum oder Teilbaum mit Hilfe eines `IndentPrinter` formatiert aus.

Konstruktoren

```
NodePrinter()
NodePrinter(IndentPrinter out)
NodePrinter(PrintWriter out)
```

Erzeugt einen neuen `NodePrinter`, der den angegebenen `IndentPrinter` oder den angegebenen `PrintWriter` verwendet. Ist nichts anderes gesetzt, wird der Baum in die Standardausgabe geschrieben.

Methoden

`void print (Node knoten)`

Gibt den mit dem angegebenen Knoten beginnenden Baum oder Teilbaum aus.

OrderBy

`class OrderBy implements Comparator`

`Comparator`-Implementierung, die den Vergleich mit Hilfe von Closures durchführt, die beim Instantiieren anzugeben sind. Die Closures haben jeweils einen Parameter.

Konstruktoren

```
OrderBy (Closure closure)
OrderBy (List<Closure> closureList)
```

Erzeugt ein neues `OrderBy`-Objekt anhand der übergebenen Closures.

Methoden

`void add (Closure closure)`

Fügt eine weitere Closure hinzu.

`int compare (Objekt o1, Object o2)`

Vergleicht die beiden Werte anhand des Ergebnisses durchgeführt, den der Aufruf der Closure für jeden der Werte liefert. Wenn mehrere Closures angegeben sind, werden sie nacheinander solange aufgerufen, bis eine Closure ein Ergebnis ungleich 0 liefert.

Proxy

```
class Proxy extends GroovyObjectSupport
```

Basisklasse für in Groovy geschriebene Proxy-Klassen. Diese leiten alle Methodenaufrufe an ein anderes Objekt (den »Adaptee«) weiter. Bei Bedarf können die Methodenaufrufe in abgeleiteten Klassen abgefangen werden.

Properties

`Object adaptee`

Objekt, an das die Methodenaufrufe weitergeleitet werden sollen. Muss gesetzt werden, bevor eine `Proxy`-Instanz verwendet werden kann.

Methoden

`Object invokeMethod (String name, Object args)`

Leitet die Methodenaufrufe an das Adaptee-Objekt weiter. Diese Methode muss in abgeleiteten Klassen überschrieben werden, um einzelne Methodenaufrufe abfangen zu können.

`Iterator iterator()`

Liefert einen Iterator über das Adaptee-Objekt.

`Proxy wrap (Object adaptee)`

Setzt das Adaptee-Objekt und gibt den aktuellen `Proxy` als Ergebnis zurück. Mit Hilfe dieser Methode kann ein Proxy leicht instanziiert werden, ohne dass in abgeleiteten Klassen entsprechende Konstruktoren geschrieben werden müssen. Das Instantiieren kann vielmehr so erfolgen:

```
def proxy = new MeinProxy().wrap(adapteeObjekt)
```

XmlNodePrinter

```
class XmlNodePrinter
```

Mit Hilfe dieser Klasse können (Teil-)Baumstrukturen aus `Node`-Objekten in der Form von XML-Dokumenten ausgegeben werden.

Konstruktoren

```
XmlNodePrinter ()
XmlNodePrinter (IndentPrinter out, String quote="\"")
XmlNodePrinter (PrintWriter out, String indent="  ", String quote="\"")
```

Erzeugt einen neuen `XmlNodePrinter` anhand des angegebenen `IndentPrinter` oder `PrintWriter`. Es können zusätzlich das zu verwendende `Anführungszeichen` für Attribute und der zum Einrücken zu benutzende String angegeben werden.

Methoden

`void print (Node knoten)`

Gibt den mit dem angegebenen Wurzelknoten beginnenden Baum oder Teilbaum aus.

XmlParser

```
class XmlParser implements org.xml.sax.ContentHandler
```

Ermöglicht das Einlesen eines kompletten, in Form eins Textes vorliegenden XML-Dokuments in eine aus Node-Objekten bestehende Baumstruktur.

Konstruktoren

```
XmlParser (boolean validating=false, boolean namespaceaware=true)
```

Erzeugt einen neuen XmlParser mit den angegebenen Eigenschaften bezüglich Validierung und Berücksichtigung von Namensräumen.

```
XmlParser (javax.xml.parsers.SAXParser parser)
XmlParser (org.xml.sax.XMLReader reader)
```

Bei diesen alternativen Konstruktoren kann ein SAX-Parser bzw. ein XML-Reader angegeben werden, die zu verwenden sind.

Properties

```
org.xml.sax.Locator documentLocator
```

Objekt zur Lokalisierung von Dokumenten.

```
org.xml.sax.DTDHandler dTDHandler
```

Objekt zur Behandlung von DTDs.

```
org.xml.sax.EntityResolver  entityResolver
```

Objekt zur Auflösung von XML-Entities.

```
org.xml.sax.ErroHandler
```

Objekt zur Behandlung von Parser-Fehlern.

Methoden

```
Node parse (File datei)
Node parse (org.xml.sax.InputSource input)
Node parse (InputStream input)
Node parse (Reader reader)
Node parse (String uri)
```

Wandelt das aus verschiedenen Quellen eingelesene XML-Dokument in eine Baumstruktur um und liefert den Wurzelknoten als Node-Objekt.

```
Node parseText (String text)
```

Wie parse(), jedoch wird das XML-Dokument direkt als String-Parameter übergeben.

ANHANG C
SwingBuilder-Methoden

Die folgende Tabelle listet alle virtuellen Methoden des `SwingBuilder`, geordnet nach Bereichen, sowie die jeweils zugrunde liegenden Java- oder Groovy-Klassen auf. Es ist jeweils nur der Name der Methode angegeben und nicht die vollständige Signatur, da diese immer dasselbe Muster hat. Argumente können Strings und Maps sein, Komponenten, in denen weitere Elemente geschachtelt sein können, zusätzlich auch eine Closure für die untergeordneten Komponenten. Näheres zur Funktionsweise des `SwingBuilder` können Sie in Kapitel 6 nachlesen.

Fensterkomponenten

Methodenname	Klasse
`colorChooser`	`javax.swing.JColorChooser`
`dialog`	`javax.swing.JDialog`
`fileChooser`	`javax.swing.JFileChooser`
`frame`	`javax.swing.JFrame`
`optionPane`	`javax.swing.JOptionPane`
`window`	`javax.swing.JWindow`

Containerkomponenten

Methodenname	Klasse
`box`	`javax.swing.JBox`
`container`	(Platzhalter für externe Containerkomponente)
`desktopPane`	`javax.swing.JDesktopPane`
`internalFrame`	`javax.swing.JInternalFrame`
`layeredPane`	`javax.swing.JLayeredPane`

Methodenname	Klasse
panel	javax.swing.JPanel
scrollPane	javax.swing.JScrollPane
splitPane	javax.swing.JSplitPane
tabbedPane	javax.swing.JTabbedPane
toolBar	javax.swing.JToolBar
viewport	javax.swing.JViewPort

Menükomponenten

Methodenname	Klasse
checkBoxMenuItem	javax.swing.JCheckBoxMenuItem
menu	javax.swing.JMenu
menuBar	javax.swing.JMenuBar
menuItem	javax.swing.JMenuItem
popupMenu	javax.swing.JPopupMenu
radioButtonMenuItem	javax.swing.JRadioButtonMenuItem

Elementare Komponenten

Methodenname	Klasse
button	javax.swing.JButton
checkBox	javax.swing.JCheckBox
comboBox	javax.swing.JComboBox
editorPane	javax.swing.JEditorPane
formattedTextField	javax.swing.JFormattedTextField
label	javax.swing.JLabel
list	javax.swing.JList
passwordField	javax.swing.JPasswordField
progressBar	javax.swing.JProgessBar
radioButton	javax.swing.JRadioButton
scrollBar	javax.swing.JScrollBar
separator	javax.swing.JSeparator
slider	javax.swing.JSlider
spinner	javax.swing.JSpinner
table	javax.swing.JTable
textArea	javax.swing.JTextArea
textPane	javax.swing.JTextPane
textField	javax.swing.JTextField
toggleButton	javax.swing.JToggleButton

Methodenname	Klasse
tree	javax.swing.JTree
widget	(Platzhalter für externe Containerkomponente)

Layouts und Constraints

Methodenname	Klasse
borderLayout	java.awt.BorderLayout
box	javax.swint.Box
boxLayout	javax.swing.BoxLayout
cardLayout	java.awt.CardLayout
flowLayout	java.awt.FlowLayout
gbc	(Kurzform für gridBagLayout)
glue	(Ruft Box.createGlue() auf.)
gridBagLayout	java.awt.GridBagLayout
gridBagConstraints, gbc	java.awt.GridBagConstraints
gridLayout	java.awt.GridLayout
hbox	(Ruft Box.createHorizontalBox() auf.)
hglue	(Ruft Box.createHorizontalGlue() auf.)
hstrut	(Ruft Box.createHorizontalStrut() auf.)
overlayLayout	javax.swing.OverlayLayout
rigidArea	(Ruft Box.createRigidArea() auf.)
springLayout	java.awt.SpringLayout
tableLayout	groovy.swing.impl.TableLayout
td	groovy.swing.impl.TableLayout.TD
tr	groovy.swing.impl.TableLayout.TR
vbox	(Ruft Box.createVerticalBox() auf.)
vglue	(Ruft Box.createVerticalGlue() auf.)
vstrut	(Ruft Box.createVerticalStrut() auf.)

Modelle

Methodenname	Klasse
boundedRangeModel	javax.swing.DefaultBoundedRangeModel
buttonGroup	javax.swing.ButtonGroup
closureColumn	groovy.swing.impl.TableModel.ClosureColumn
propertyColumn	groovy.swing.impl.TableModel.PropertyColumn
spinnerDateModel	javax.swing.SpinnerDateModel
spinnerListModel	javax.swing.SpinnerListModel

Methodenname	Klasse
spinnerNumberModel	javax.swing.SpinnerNumberModel
tableModel	javax.swing.table.TableModel
tableColumn	javax.swing.table.TableColumn

Sonstiges

Methodenname	Klasse
action	javax.swing.Action
actions	java.util.Collection<javax.swing.Action>
map	java.util.Map<String,Object>

ANHANG D
Groovy-Tools

Dieser Anhang enthält eine Übersicht der zu Groovy gehörenden Werkzeugprogrammen mit ihren jeweiligen Optionen.

groovy – der Groovy-Starter

Der Kommandozeilenbefehl `groovy` dient dazu, als Quelldatei vorliegende Groovy-Programme direkt auszuführen. Standardmäßig wird er folgendermaßen aufgerufen:

```
> groovy optionen programmname programmargumente
```

In *programmname* kann die Dateiendung weggelassen werden, wenn die auszuführende Datei eine der Endungen *.groovy*, *.gvy*, *.gy* *.gsh* oder gar keine Endung hat. Die Datei muss entweder ein Groovy-Skript (ohne explizite Klassendefinition) oder eine für Groovy ausführbare Klasse beinhalten (also eine Java-Main-Klasse oder eine Klasse, die `java.lang.Runnable`, `groovy.util.GroovyTestCase` oder `groovy.util.GroovyTestSuite` implementiert. Alle nach dem Programmnamen angegebenen Argumente werden dem Programm übergeben. Tabelle D-1 zeigt die Optionen, die zusätzlich *vor* dem Programmnamen angegeben werden können.

Tabelle D-1: Optionen für den Befehl groovy

Option	Bedeutung
`-a` *muster* `--autosplit` *muster*	Trennt die Eingabezeilen mit dem als *muster* angegebenen regulären Ausdruck in Teilstücke auf und übergibt diese dem Programm einzeln. Wenn kein Muster angegeben ist, wird an Leerzeichen getrennt. Gilt nur in Verbindung mit den Optionen `-l`, `-n` oder `-p`.
`-c` *charset* `--encoding` *charset*	Gibt das Encoding der Eingabedateien an. Als Argument muss die standardmäßige Bezeichnung eines Java bekannten Zeichensatzes angegeben werden.
`-d` `--debug`	Gibt im Fall einer Exception einen vollständigen Stacktrace aus.

Tabelle D-1: Optionen für den Befehl groovy (Fortsetzung)

Option	Bedeutung
`-e` *`script`*	Bei Angabe dieser Option wird das als Klartext angegebene Skript direkt ausgeführt.
`-h` `--help`	Gibt eine Übersicht der unterstützten Optionen aus. Alle weiteren Optionen und Argumente bleiben unbeachtet.
`-i` *`extension`*	Schreibt die Standardausgabe direkt in die als Argument angegebene Datei zurück. Das Original der Eingabedatei wird in einer Datei mit der Erweiterung *`extension`* gesichert. Gilt nur in Verbindung mit den Optionen `-n` und `-p`.
`-l` *`portnummer`*	Groovy horcht an dem Port mit der angegebenen Nummer und ruft das Skript bei jeder eingehenden Zeile auf. Die Zeile wird dem Skript als Binding-Variable `line` übergeben. Wenn keine Portnummer angegeben ist, wird Port 1960 verwendet.
`-n`	Interpretiert die Befehlsargumente als Namen von Eingabedateien und übergibt diese zeilenweise dem Skript. Wenn keine Argumente angegeben sind, wird die Standardeingabe zeilenweise übergeben. Diese Option wird typischerweise in Verbindung mit der Option `-e` verwendet.
`-p`	Wie Option `-n`, jedoch wird zusätzlich der Ergebniswert des Skripts nach jedem Aufruf in die Standardausgabe geschrieben.
`-v` `--version`	Gibt die Versionsnummern von Groovy und Java aus. Alle weiteren Optionen und Argumente bleiben unbeachtet.

groovyc – der Groovy-Compiler

Der Befehl groovyc dient dazu, Groovy-Quellprogramme in Java-Klassendateien zu übersetzen, die wie normale Java-Programme ausgeführt werden können. Der Aufruf lautet:

```
> groovyc optionen quelldateien
```

Als Quelldateien sind ein oder mehrere Dateinamen von Groovy-Quellprogrammen anzugeben. Zusätzlich können die in Tabelle D-2 aufgeführten Optionen angegeben werden.

Tabelle D-2: Optionen für den Befehl groovyc

Option	Bedeutung
`-cp` *`pfad`* `--classpath` *`pfad`*	Gibt den Klassenpfad in der üblichen Form an, damit der Compiler referenzierte Klassen finden kann.
`-d verzeichnis`	Gibt das Verzeichnis an, in das die übersetzten Klassendateien geschrieben werden sollen.
`--encoding` *`zeichensatz`*	Gibt den für die Quelldateien zu verwendenden Zeichensatz an.
`-e` `--exception`	Weist den Compiler an, im Fall eines Fehlers einen vollständigen Stacktrace auszugeben.
`-h` `--help`	Gibt eine Übersicht der unterstützten Optionen aus. Alle weiteren Optionen und Argumente bleiben unbeachtet.
`-j` `--jointCompilation`	Leitet alle Dateien mit der Endung *.java* an den Java-Compiler weiter.
`-Joption=wert`	Definiert eine Option mit Wert, die an den Java-Compiler weitergegeben werden soll. Gilt nur, wenn die Option `-j` gesetzt ist.

Tabelle D-2: Optionen für den Befehl groovyc (Fortsetzung)

Option	Bedeutung
`-Loption`	Definiert eine Option ohne Wert, die an den Java-Compiler weitergegeben werden soll. Gilt nur, wenn die Option `-j` gesetzt ist.
`-v` `--version`	Gibt zu Beginn die Versionsnummern von Groovy und Java aus.

groovysh – die Groovy-Shell

Der Befehl `groovysh` ermöglicht die interaktive Eingabe von Groovy-Skripten in der Konsole. Dabei gilt das Prinzip, dass die Eingaben nach der Eingabeaufforderung `groovy>` so lange gesammelt werden, bis der Benutzer den Befehl `go` eingibt. Dies ermöglicht die Eingabe mehrzeiliger Skripte. Die Groovy-Shell wird ohne Optionen und Argumente gestartet:

```
> groovysh
Let's get Groovy!
===============
Version: 1.1 JVM: 1.6.0_01-b06
Type 'exit' to terminate the shell
Type 'help' for command help
Type 'go' to execute the statements
groovy>
```

Neben `go` können nach der Eingabeaufforderung noch einige weitere Befehle eingegeben werden, die in Tabelle D-3 aufgelistet sind.

Tabelle D-3: Von der Groovy-Shell unterstützte Befehle

Befehl	Bedeutung
`binding`	Zeigt die Namen und die Werte der aktuell im Binding gespeicherten Variablen an.
`discard`	Entfernt das aktuelle Skript aus dem Speicher, ohne es auszuführen.
`discardclasses`	Entfernt alle zuvor eingegebenen Klassendefinitionen, sofern sie nicht durch Binding-Variablen oder andere Klassen referenziert werden.
`display`	Zeigt das aktuelle Skript zeilenweise an.
`execute` `go`	Führt das aktuelle Skript aus.
`exit` `quit`	Beendet die Groovy-Shell.
`explain`	Zeigt den Syntaxbaum des aktuellen Skripts an.
`help`	Gibt eine Zusammenfassung der unterstützten Befehle aus.
`inspect`	Öffnet in einem getrennten Fenster den Groovy-Objektbrowser, der alle Eigenschaften des letzten Skript-Ergebniswerts anzeigt.

GroovyConsole – die interaktive Konsole

Die interaktive Konsole ist eine minimale grafische Entwicklungsumgebung für Groovy-Programme. Sie besteht aus zwei Teilfenstern. Das eine von ihnen dient dazu, ein Groovy-Skript einzugeben, während im anderen die Ergebnisse der Ausführung dieses Skripts angezeigt werden. Der Aufruf erfolgt ohne Parameter und ohne Optionen:

```
> GroovyConsole
```

Die Anwendung hat eine Reihe von Menüoptionen, die in Tabelle D-4 erläutert werden.

Tabelle D-4: Menüoptionen der Groovy-Konsole

Befehl	Taste	Bedeutung
File→new File		Legt eine neue Quelldatei an.
File→new Window		Erzeugt ein neues Konsolenfenster.
File→Open		Öffnet eine vorhandene Quelldatei zur Bearbeitung.
File→Save		Speichert das aktuelle Programm als Datei.
File→Exit	Alt+F4	Beendet die Groovy-Konsole.
Edit→Next	Strg+N	Zeigt eine frühere Version des Skripts an.
Edit→Previous	Strg+P	Zeigt die nächste Version des Skripts an.
Edit→Clear Output	Strg+W	Löscht das Ausgabefenster.
Actions→Run	Strg+R	Führt das Programm im Eingabefenster aus.
Actions→Inspect last	Strg+I	Öffnet in einem getrennten Fenster den Groovy-Objektbrowser, der alle Eigenschaften des letzten Skript-Ergebniswerts anzeigt.
Actions→Inspect variables	Strg+J	Öffnet in einem getrennten Fenster den Groovy-Objektbrowser, der alle Binding-Variablen mit ihren Eigenschaften anzeigt.
Actions→Capture Stdout		Legt fest, ob die Standardausgabe des Skripts abgefangen und im Ausgabefeld dargestellt werden soll. Standardmäßig eingeschaltet.
Actions→Larger Font	Alt+Umschalt+L	Stellt die Anzeige auf eine größere Schriftart um.
Actions→Smaller Font	Alt+Umschalt+S	Stellt die Anzeige auf eine kleinere Schriftart um.
Help→About		Zeigt die Groovy-Version an.

Index

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Über den Autor

Jörg Staudemeyer, von Hause aus Wirtschaftswissenschaftler und eigentlich per Zufall an die »Computerei« geraten, ist als Senior-Consultant bei der European IT Consultancy EITCO GmbH tätig. Während seiner fast 20jährigen Beratertätigkeit im Kontext kommerzieller Großprojekte hatte er Gelegenheit, diverse Bereiche der Informationstechnik kennenzulernen. Seine langjährige Tätigkeit als Fachübersetzer und Autor für den O'Reilly Verlag (unter anderem zu Java und anderen Enterprise-Computing-Technologien) nutzt er als willkommene Gelegenheit, sprachliche und technische Interessen zu vereinen. Seine Partnerin, die Germanistin und Kreativ-Schreib-Expertin Brigitte Schulte, unterstützt ihn dabei, Texte in korrektem und verständlichem Deutsch zu produzieren. Beide lieben es, auf ausgedehnten Fahrradtouren körperliche Bewegung mit Natur- und Kulturgenuss zu kombinieren.

Kolophon

Das Tier auf dem Cover von *Groovy für Java-Entwickler* ist ein Doktorfisch (Acanthurida). Die 72 Arten der Doktorfische sind in ihrer Farbgebung, Form und Größe sehr unterschiedlich. Alle Fische dieser Familie haben allerdings skalpellartige Stacheln an den Seiten der Schwanzwurzeln, denen sie ihren Namen verdanken, da sie an Skalpelle von Ärzten erinnern. Diese messerscharfen Schneiden dienen jedoch nur der Verteidigung und werden bei Angriffen klappmesserartig vom Körper abgespreizt. Bei vielen Arten sind sie kräftig gefärbt.

Abgesehen von dieser gefährlichen Waffe sind die meisten Doktorfische aber sehr friedfertige Tiere, die sich vor allem vegetarisch von Algen und Plankton ernähren. Sie leben in den warmen, flachen Gewässern der Tropen, vor allem im Indopazifik, in unmittelbarer Nähe von Riffen. Tagsüber raspeln sie hier ihre Nahrung von Korallen und Felsen ab, nachts schlafen sie versteckt im Korallenriff. Die meisten Arten leben einzelgängerisch, doch es gibt auch Arten, die Schwärme bilden, um im offenen Wasser gemeinsam nach Plankton zu fischen. Die Gruppe bietet ihnen dabei Schutz vor Fressfeinden. Doktorfische sind langsame Schwimmer, da sie nur die Brustflossen benutzen. Dies verleiht ihnen ihre typische auf- und abwärtsgerichtete Schwimmbewegung.

Bei Vollmond kommen Männchen und Weibchen für die Balz und die Eiablage zusammen. Dafür schwimmt das Pärchen ins offene Wasser, wo Eier und Spermien abgegeben werden. Die befruchteten Eier werden durch die Strömung ins Meer getrieben, wo die Larven heranwachsen und erst im ausgewachsenen Stadium zum Riff zurückkehren.

Manche Doktorfischarten sind beliebte Speisefische. Bei ihrer Nahrungsaufnahme nehmen die Tiere allerdings oftmals in Algen vorhandene Gifte wie Maitotoxin und Ciguatoxin zu sich, die für die Tiere selbst ungefährlich sind, die sich im Körper aber anreichern. Der Verzehr von Doktorfischen kann daher zu Vergiftungserscheinungen, Ciguatera genannt, führen.

Der Umschlagsentwurf dieses Buchs basiert auf dem Reihenlayout von Edie Freedman und stammt von Michael Oreal, der hierfür einen Stich des *Dover Pictorial Archive* aus dem 19. Jahrhundert verwendet hat. Als Textschrift verwenden wir die Linotype Birka, die Überschriftenschrift ist die Adobe Myriad Condensed und die Nichtproportionalschrift für Codes ist LucasFont's TheSans Mono Condensed. Geesche Kieckbusch hat das Kolophon geschrieben.