2 Was ist Groovy?

In diesem Kapitel bekommen Sie einen ersten Überblick über die Sprache Groovy und ihre Eigenschaften. Es klärt die Frage, warum eine weitere Sprache nicht überflüssig, sondern im Gegenteil höchst notwendig ist, und gibt erste Eindrücke der Eleganz und Einfachheit in der Benutzung von Groovy.

Eventuell stehen Sie gerade im Buchladen und blättern in diesem Buch, um zu entscheiden, ob es sich lohnt, es mitzunehmen. Dies ist eine Frage, die letztendlich natürlich nur Sie beantworten können, aber lesen Sie auf jeden Fall die Warnung am Schluss des Kapitels, und seien Sie versichert, dass sich die Beschäftigung mit Groovy auf jeden Fall lohnt.

2.1 Einführung

Vorkenntnisse: Objektorientierung und Java

Im Rahmen dieses Buches muss ich eine Grundannahme treffen: Ich gehe davon aus, dass Sie Objektorientierung und insbesondere Java kennen. Ich kann keine grundlegende Einführung in die objektorientierte Programmierung geben, und auch eine Einführung in Java würde den Rahmen des Buches sprengen. Leider. Es gibt aber ausgezeichnete Bücher zu diesem Thema. Falls Ihnen keines gefällt, sprechen Sie mit dem Verlag, vielleicht darf ich dann auch zu diesem Thema ein Buch schreiben.

Wenn Sie keine Vorkenntnisse in diesen Bereichen haben, heißt das nicht, dass Sie keine Freude an diesem Buch haben werden, aber wahrscheinlich werden Ihnen zwei Dinge passieren:

- Sie werden sich über den Enthusiasmus wundern, den die eine oder andere Spracheigenschaft bei mir auslöst, indem Sie etwas für völlig normal halten, was in Java einfach nicht geht und nur durch Groovy ermöglicht wird, beziehungsweise

- Sie werden an der einen oder anderen Stelle ein wenig mehr überlegen und/oder ein Java-Buch beziehungsweise ein Buch über Objektorientierung konsultieren müssen, um zu verstehen, was ein bestimmtes Konstrukt bewirkt.

In Summe werden Sie aber vielleicht sogar mehr Freude haben, die Eleganz von Groovy zu entdecken, als die Leute, die schon seit zehn Jahren in Java programmieren.

Lassen Sie sich also nicht durch geringe Vorkenntnisse abhalten von der Freude, die Sie mit Groovy erleben können.

2.2 Entstehungsgeschichte

James Strachan, Bob McWhirter

Gemessen an manchen anderen Sprachen ist Groovy noch relativ jung. Die ersten Ideen entstanden Anfang 2003 im Kopf von James Strachan, und das zugehörige Projekt wurde im August 2003 bei Codehaus (siehe [Codehaus]) ins Leben gerufen. Dort werden auch viele andere erstklassige Open-Source-Anwendungen veröffentlicht, aber der Hauptgrund für die Wahl von Codehaus war wahrscheinlich, dass der zweite Initiator von Groovy, Bob McWhirter, auch der Gründer von Codehaus ist.

Laut der Erzählungen von James Strachan entstanden viele der Ideen für Groovy dadurch, dass James die Sprache Python kennenlernte und überlegte, wie sich die Funktionalität dieser Sprache in Java realisieren ließe.

Der Name der Sprache

Die folgende Anekdote zur Entstehung des Namens der Sprache stammt von Guillaume LaForge, dem heutigen Leiter des Groovy-Projektes:

James Strachan hatte mit einigen Skriptsprachen herumgespielt, und er unterhielt sich mit Bob McWhirter. Die ganze Zeit wiederholte er: »Wäre es nicht groovy, wenn wir dies oder das in Java machen könnten?« Dabei benutzte er laufend das Adjektiv »groovy«.

Und als sie dann Ideen sammelten, die sie gerne in einer Skriptsprache für die JVM implementiert sehen würden, entschieden sie, diese Sprache »Groovy« zu nennen.

Gerade James Strachan stellte am Anfang eine treibende Kraft dar und sorgte für sehr viel Werbung. Das war nicht ausschließlich positiv, da Groovy zu Beginn einfach nicht reif war für die allgemeine Verwendung. Es ist aber auf jeden Fall James Strachan zu verdanken, dass Groovy überhaupt bekannt geworden ist und derart schnell zu einer tatsächlichen Alternative im Bereich der Skriptsprachen geworden ist.

JSR-241

Im März 2004 wurde Groovy als JSR-241 (Java Specification Request) in den Java Community Process eingebracht. Groovy erfährt

von Sun-Seite aus sehr starke Aufmerksamkeit. So sind auf der JavaOne-Konferenz 2007 extrem viele Vorträge zum Thema Groovy gehalten worden, und Sun stellt zum Beispiel auch Hardware für die Entwicklung von Groovy zur Verfügung.

Guillaume LaForge, Jochen Theodorou

Inzwischen haben aber sowohl James Strachan als auch Bob McWhirter das Projekt verlassen und haben neue Beschäftigungen gefunden. Seitdem sorgen Guillaume LaForge als der Projektleiter, Jochen Theodorou als der technische Leiter und eine rege Gruppe von Leuten, die sich intensiv mit der Weiterentwicklung der Sprache beschäftigen, dafür, dass Groovy nicht nur extrem stabil ist, sondern auch ständig neue interessante Fähigkeiten erhält.

2.3 Vorteile einer Skriptsprache

Grundsätzlich ist allen Skriptsprachen gemein, dass sie einen höheren Abstraktionsgrad besitzen als klassische Programmiersprachen und dass häufig verwendete Programmierkonstrukte weniger geschwätzig sind als Pendants in anderen Programmiersprachen. Funktionalität, die auszudrücken uns in einer »normalen« Programmiersprache durchaus ein Dutzend Zeilen oder mehr kosten kann, lässt sich häufig in einer Skriptsprache mit einer Zeile formulieren.

Schnelleres Programmieren

Dies hat mehrere Konsequenzen: Die Programmierung in einer Skriptsprache ist damit im Normalfall schneller erledigt. Auch die Wartung eines Programms in einer Skriptsprache ist im Normalfall effizienter. Nehmen wir das Standardbeispiel in Programmiersprachen seit Kernighan & Ritchie, die Ausgabe von »Hallo Welt« über den Standardausgabestrom (im Normalfall die Konsole). Wir betrachten die Realisierung in Java (Listing 2–1)

Listing 2–1
Das »Hallo Welt«-Programm in Java

```
package de.groovybuch.kap2;

public class HelloWorld {
   public static void main (String [] args) {
      String adressat = "Welt";
      System.out.println("Hallo, " + adressat);
   }
}
```

und die Realisierung in Groovy (Listing 2–2, diese Variante ignoriert allerdings die Package-Deklaration):

Listing 2–2
Das »Hallo Welt«-Programm in Groovy

```
adressat = "Welt"
println "Hallo, $adressat"
```

Um das Beispiel nicht zu einfach zu halten, fügen wir eine Variable hinzu, die den Adressat enthält. Wir sehen, wie wir in Java die Zeichenketten mit dem Gruß und dem Adressat explizit verknüpfen müssen, während in Groovy die direkte Referenzierung von Variablen innerhalb der Zeichenkette möglich ist.

Hier haben wir ein exzellentes Beispiel dafür, dass Groovy prägnantere Formulierung erlaubt (ohne Lesbarkeit aufzugeben), und es ist offensichtlich, dass die Groovy-Variante dieses Programmes einfacher zu erstellen und zu warten ist.

Ausführungsgeschwindigkeit

Eine negative Eigenschaft, die automatisch mit Skriptsprachen assoziiert wird, ist eine deutlich niedrigere Ausführungsgeschwindigkeit, etwas, das sich allerdings in der Wildnis der täglichen Programmierung eher selten beobachten lässt.

Erstens ist die Geschwindigkeit einzelner Befehle meist deutlich weniger bedeutend als normalerweise angenommen, wie jeder bestätigen kann, der schon einmal ein Programm mit einem Profiler optimiert hat. Zweitens ist in den meisten Fällen die Wahl eines besseren Algorithmus deutlich laufzeiteffektiver als der Wechsel der Sprache. Drittens treiben moderne Laufzeitumgebungen einen extremen Aufwand zur Optimierung eines Programms während der Ausführung. Als naheliegendes Beispiel sei hier Java genannt. Vor zehn Jahren hatte man im Vergleich zu C++ einen Geschwindigkeitsfaktor von ca. 1:100, während er heute bei den meisten Anwendungen bei 1:1 liegt. Und spätestens damit verliert das Geschwindigkeitsargument seine Stichhaltigkeit.

Keine expliziten Übersetzungsläufe

Ein früher häufig gehörtes Argument für Skriptsprachen ist, dass sie keinen expliziten Übersetzungslauf benötigen. Dies führt zu kürzeren Zyklen in der Entwicklung. Allerdings ist dieses Argument in Zusammenhang mit modernen Entwicklungsumgebungen nicht mehr stichhaltig. Betrachten wir das Beispiel Eclipse: Hier ist der Übersetzungslauf so elegant integriert, dass das Abspeichern eines veränderten Quelltextes automatisch zur inkrementellen Neuübersetzung führt. Dies ist – je nach Sprache – sogar zur Laufzeit während der Fehlerbeseitigung (Debugging) möglich.

Zusammenfassend lässt sich sagen, dass Skriptsprachen die Funktionalität normaler Programmiersprachen kombiniert mit zusätzlicher syntaktischer und semantischer Eleganz (auf Deutsch: coole zusätzliche Eigenschaften) anbieten, und dies im Allgemeinen ohne die negativen Eigenschaften, die ihnen zugesprochen werden.

2.4 Die Sprache Groovy

Wie positioniert sich hier Groovy? Auf der einen Seite verhält sich Groovy wie eine Skriptsprache, da zum einen kein expliziter Übersetzungslauf benötigt wird, zum anderen die Programmierung wie bei einer Skriptsprache durchgeführt werden kann. Auf der anderen Seite führt hinter den Kulissen jeder Aufruf eines Groovy-Skriptes zu einer Umwandlung in eine (oder mehrere) Java-Klassen.

Python, Ruby, Perl

Groovy ist mit seinem hohen Abstraktionsgrad, der impliziten Typisierung und den eleganten Sprachkonstrukten auf einer Ebene mit Skriptsprachen wie Python oder Ruby (und deutlich eleganter als Perl). Außerdem ist es eine Erweiterung von Java und als solche natürlich sehr Java-ähnlich.

Java

Dies bedeutet, dass jeder Java-Entwickler sofort Groovy-Programme lesen und nach ca. einer halben Stunde Einarbeitung auch schreiben kann. Anders als mit Python, Ruby oder Perl gibt es hier keine Einstiegshürde, kein Nachdenken und kein Nachschlagen unbekannter Funktionalität innerhalb von Bibliotheken, die man nicht kennt. Die größte Gefahr für Java-Entwickler (und übrigens auch für Groovy-Entwickler) ist die Frage, ob das aktuelle Problem nicht vielleicht »noch eleganter« zu lösen wäre. Das »Schlimme« ist, dass es in vielen Fällen tatsächlich noch eleganter geht ...

2.5 Interaktion mit Java

Nun stellt sich natürlich die Frage, ob Groovy irgendwie mit Java interagieren kann, wenn es schon von Java abstammt. Und tatsächlich kann es das. Überraschen mag allerdings die Vollständigkeit der Integration und die Tatsache, dass diese Integration in beide Richtungen nahezu perfekt funktioniert.

Java-Klassen für Groovy, Groovy-Programme für Java

Das bedeutet also nicht nur, dass Groovy sämtliche Java-Klassen verwenden kann, die im Klassenpfad existieren, sondern zusätzlich auch, dass Groovy-Programme von Java aus verwendet werden können!

Wie funktioniert dies? Jedes Groovy-Programm wird beim Aufruf automatisch in Bytecode übersetzt, der dann ausgeführt wird. Alternativ kann mithilfe des Groovy-Compilers auch explizit eine übersetzte Version (als Bytecode in einem Class-File) erzeugt werden. Die hierin enthaltene Klasse kann genau wie jede andere Java-Klasse verwendet werden.

Zum Beispiel kann eine langsame Groovy-Einführung in einem schon existierenden Projekt dadurch geschehen, dass das Groovy-

Archiv eingebunden wird und die Groovy-Klassen für den Aufruf von Java aus zur Verfügung gestellt werden.

Falls Groovy am Anfang nur dazu verwendet werden soll, die Tests elegant zu formulieren, so kann auch dies ohne Probleme geschehen. Die zu testenden Java-Klassen können direkt zum Test aufgerufen werden.

Die Verwendung von Java-Klassen innerhalb eines Groovy-Projektes, in denen die geschwindigkeitskritischen Programmteile abgehandelt werden, ist durch einfachen Aufruf möglich.

2.6 Erweiterungen gegenüber Java

Die verschiedenen Erweiterungen, die Groovy gegenüber Java bietet, sind an und für sich schon sehr hilfreich. Aber sie entfalten ihre volle Mächtigkeit dadurch, dass sie miteinander Hand in Hand arbeiten, um das Programmieren möglichst angenehm zu machen.

Manche Kombinationsmöglichkeiten ergeben sich erst mit zunehmender Erfahrung. Da dies jedes Mal mit einem Aha-Erlebnis verbunden ist, bleibt die Programmierung in Groovy immer eine Freude.

2.6.1 Erweiterte Spracheigenschaften

Alles ist ein Objekt

Beim Sprachdesign von Java wurde aus Effizienzgründen die Entscheidung getroffen, primitive Datentypen einzuführen. Diese Entscheidung hat aber weitreichende Konsequenzen darin, wie mit Werten innerhalb der Sprache umgegangen wird.

Um diese Probleme nicht zu komplex werden zu lassen, bietet Java seit jeher `Wrapper`-Klassen an, die einen primitiven Datentyp in einem Objekt kapseln. Beispiele sind die Klassen `Integer` für den Datentyp `int` oder `Double` für den Datentyp `double`. Man muss aber jetzt an vielen Stellen unterscheiden, ob man auf einem primitiven Datentyp operiert oder ein Objekt einer Klasse vor sich hat.

Bei Groovy gibt es keine Unterscheidung zwischen primitiven Datentypen und Objekten. Groovy verwendet ausschließlich Objekte. Wenn wir im Quelltext zum Beispiel die Zahl 3 notieren, dann wird diese sofort umgewandelt in ein Objekt der Klasse `Integer`. Wir können uns also immer darauf verlassen, mit Objekten umzugehen. Es gibt

auf Sprachebene keine primitiven Datentypen in Groovy. Das Beispiel in Listing 2–3 illustriert dies:

```
5.times { println "Hallo Welt" }
```

Listing 2–3
Methodenaufrufe auf einer Zahl

Wir rufen auf einer Zahl eine Methode auf `times()` auf, die den folgenden Anweisungsblock (eigentlich eine Closure, siehe nächster Abschnitt) in entsprechender Anzahl aufruft. Resultat ist die fünfmalige Ausgabe von »Hallo Welt«.

Da Groovy auf Java aufbaut und insbesondere die Java Virtual Machine (JVM) verwendet, wird natürlich unter der Haube auch mit primitiven Datentypen umgegangen, aber dies ist beim Programmieren völlig unsichtbar (und schlicht irrelevant).

Closures

Groovy bietet Methoden als Objekte erster (und höherer) Ordnung an. Dieses Konzept ist vor allem in der funktionalen Programmierung seit Langem bekannt und hat in den letzten Jahren mehr und mehr Einzug in imperative Programmiersprachen gefunden (Python und Ruby, aber auch zum Beispiel C# bieten dies an). Java verfügt zwar mit dem Sprachkonstrukt der anonymen inneren Klassen über etwas, das die Nachbildung dieses Konzeptes erlaubt. Aber der Nachteil der extremen Unbequemlichkeit in der Verwendung hat dazu geführt, dass anonyme Klassen sehr wenig verwendet werden. Ein einfaches Beispiel (Listing 2–4) illustriert dies:

Listing 2–4
Verwendung anonymer Klassen in Java

```
package de.groovybuch.kap2;

public class AnonymeKlasse {
   // Wir definieren ein Interface für unsere Methode.
   // Dies stellt die Methode call() zur Verfügung
   public interface UnsereMethode {     ❶
      public Object call();
   }

   public static void main(String[] args) {
      // Wir erzeugen zwei anonyme Klassen,
      // die das Interface implementieren
      UnsereMethode m1 = new UnsereMethode() {    ❷
         public Object call() {
            return "Hallo";
         }
      };
```

```
        UnsereMethode m2 = new UnsereMethode() {    ❸
            public Object call() {
                return "Welt";
            }
        };

        // Jetzt können wir Methoden als Objekte
        // erster Ordnung übergeben
        execMethod(m1);
        execMethod(m2);
    }

    /**
     * Wir führen eine Methode aus und geben das Resultat
     * auf die Konsole aus
     */
    public static void execMethod(UnsereMethode m) {    ❹
        System.out.println(m.call());
    }
}
```

Wir definieren ein Java-Interface, das ausschließlich eine Methode `call()` enthält ❶, und implementieren zwei anonyme Klassen, die diesem Interface genügen; `m1` liefert als Resultat die Zeichenkette »Hallo« ❷, `m2` liefert die Zeichenkette »Welt« ❸. Diese können wir danach als Parameter an eine Methode übergeben, die mit diesen Methodenobjekten umgehen kann ❺. Es ist offensichtlich, dass die Umständlichkeit in der Verwendung alles andere als elegant ist.

Betrachten wir die gleiche Funktionalität in Groovy (Listing 2–5):

Listing 2–5 *Verwendung von Closures, Funktionen erster Ordnung in Groovy*

```
m1 = { return "Hallo" }    ❶
m2 = { return "Welt" }    ❷

public execMethod(c) { println c() }    ❸

execMethod(m1)
execMethod(m2)
```

Wir erzeugen zwei anonyme Methoden `m1` ❶ und `m2` ❷, die »Hallo« beziehungsweise »Welt« zurückliefern. Außerdem definieren wir eine Methode, die diese aufruft und das Resultat ausgibt ❸, und rufen diese für `m1` und `m2` auf.

Listing 2–6 *Closures/Funktionen mit Parametern*

```
m3 = { name -> return "Hallo, $name" }
println m3("kleiner Prinz")
```

Damit Sie sich nicht fragen, wie diese Funktionen mit Parametern umgehen, definieren wir in Listing 2–6 die Funktion `m3`, die einen Parameter namens `name` nimmt und verarbeitet.

Diese Methodenobjekte erster (und höherer) Ordnung heißen in Groovy Closure. Wichtig bei Closures ist, dass sie nicht zum Zeitpunkt der Definition ausgeführt werden, sondern erst beim expliziten Aufruf.

Closures sind so allgegenwärtig, dass es nach kürzester Zeit absolut selbstverständlich ist, sie zu verwenden. Gerade in Kombination mit Erweiterungen der Java-Klassen sind sie so mächtig, dass es nahezu unmöglich ist, auf ihre Eleganz zu verzichten. Wir werden in Kapitel 6 im Detail auf ihre Verwendung eingehen.

Categories

Eine weitere faszinierende Funktionalität von Groovy ist die Möglichkeit, existierenden Klassen Methoden hinzuzufügen beziehungsweise existierende Methoden zu ändern. Dies geht so weit, dass es sogar möglich ist, die Klassen, die von Java zur Verfügung gestellt werden, zu modifizieren. Dies geschieht über ein Sprachkonstrukt, das »Category« heißt (eine der ersten Sprachen, die dieses Konstrukt verwendete, war die Sprache »Objective C«). Hierdurch wird die Funktionalität *zur Laufzeit* verändert.

Listing 2–7
Eine Category erweitert die Funktionalität der String-Klasse.

```
static class HalloCategory {    ❶
   public static hallo(String contents) {    ❷
      return "Hallo, $contents"
   }
}

testInt = 3
testString = "kleiner Prinz"

use(HalloCategory) {    ❸

   // gibt "Hallo, kleiner Prinz" aus
   println testString.hallo()    ❹
   // erzeugt eine Exception
   //println testInt.hallo()
}

// erzeugt eine Exception
// println testString.hallo()
```

In unserem Beispiel (Listing 2–7) definieren wir eine Category `HalloCategory` ❶. Die Definition enthält eine Methode `hallo()` ❷, die allen Objekten vom Typ String hinzugefügt wird (dies wird durch den ersten Parameter ausgesagt). Nachdem wir zwei Variablen `testInt` und `testString` Werte zugewiesen haben, wird die Category durch Verwendung des Schlüsselworts `use` verwendet ❸. Hierdurch wird zur Laufzeit innerhalb des nun folgenden Blocks die Klasse String so modifiziert, dass sie über eine Methode `hallo()` verfügt. Der Versuch, die Methode

`hallo()` auf `testString` aufzurufen ❹, funktioniert damit wie erwartet. Der Versuch, diese Methode auf einem Objekt aufzurufen, das nicht erweitert wurde, schlägt fehl. Da eine Category nur innerhalb des `use`-Bereichs gültig ist, ist auch der Versuch, die Methode `hallo()` auf dem String außerhalb des `use`-Bereichs aufzurufen, nicht erfolgreich.

Auf Categories werden wir in Kapitel 7 im Detail eingehen.

Meta Object Protocol

Das Meta Object Protocol (MOP) erlaubt es Objekten, zum Aufrufzeitpunkt einer Methode zu entscheiden, was sie mit diesem Aufruf machen. Da die Methode nicht notwendigerweise als Implementierung existieren muss, kann das Objekt zum Beispiel den Methodennamen untersuchen, um zu entscheiden, was es tun soll.

Groovy ist nicht die erste Sprache, die ein Meta Object Protocol zur Verfügung stellt. Die Wurzeln finden sich bei verschiedenen Lisp-Dialekten, viele objektorientierte Sprachen verfügen über Varianten, und die Idee taucht auch beim aspektorientierten Programmieren wieder auf.

Teil des Meta Object Protocol in Groovy ist es, bei Nichtexistenz einer aufgerufenen Methode eine spezielle Methode, die Methode `invokeMethod()`, des Objektes aufzurufen. Diese kann jetzt in geeigneter Weise reagieren.

Listing 2–8
Beispiel für die Verwendung des Meta Object Protocol

```
class MOPBeispiel {
   public invokeMethod(String method, Object params) {    ❶
      println "Ich wurde aufgerufen mit $method"
   }
}

bsp = new MOPBeispiel()
bsp.helloWorld()    ❷
```

In Listing 2–8 definieren wir eine Klasse `MOPBeispiel`, die ausschließlich die Methode `invokeMethod()` implementiert ❶. Der Aufruf einer nicht existierenden Methode `helloWorld()` auf einem Objekt dieser Klasse ❷ produziert aber aufgrund des Meta Object Protocol keine Fehlermeldung, sondern bewirkt den Aufruf der Methode `invokeMethod()` mit dem Methodennamen (und einer leeren Parametermenge).

Wir könnten jetzt natürlich argumentieren, dass das nichts anderes als eine akademische Spielerei ist, aber dieser Ansatz ist extrem mächtig. Insbesondere kann das Objekt hiermit auf Methodenaufrufe sinnvoll reagieren, die zum Programmierzeitpunkt nicht bekannt waren. Im simpelsten Fall kann eine einfache Fehlermeldung ausgegeben werden. Aber auch komplexe Reaktionsmöglichkeiten existieren: Wir

könnten zum Beispiel versuchen, den Methodennamen zu interpretieren, um entsprechend sinnvoll zu reagieren. Das kann so weit gehen, dass wir mit den Methodenaufrufen eine domänenspezifische Sprache (Domain Specific Language) auf unserem Objekt implementieren, zum Beispiel um über Webservices eine sinnvolle Antwort auf Anfragen zu geben.

Interessanterweise nutzt auch Groovy selbst intern das MOP in starkem Maße, um große Teile der Sprachfunktionalität zu implementieren.

Weitere Details und Anwendungsmöglichkeiten werden wir in Kapitel 7 kennenlernen.

2.6.2 Erweiterte Java-Klassen

In Groovy sind viele der Standard-Java-Klassen erweitert worden, um eine bequemere und elegantere Nutzung zu unterstützen und die Standard-Java-Klassen damit mehr in die Sprache einzugliedern. Als Beispiel sei hier die Klasse `java.io.File` genannt, die um über 40 Methoden erweitert worden ist. Viele dieser Methoden sind solche, die wir schon seit Jahren in Java vermissen beziehungsweise die den Zugriff auf die gekapselten Daten vereinfachen und verallgemeinern. Analog zum JDK wird die Gesamtheit dieser Methoden GDK genannt (Groovy Development Kit).

```
file = new File(args[0])
file.eachLine { line -> println line }
```

Listing 2–9
Demonstration der Erweiterung der Klasse java.io.File

Das Programm in Listing 2–9 implementiert die Funktionalität des Unix-Programms `cat` und des Windows-Programms `type` (ohne Implementierung der Kommandozeilenoptionen). Wir erzeugen ein neues Objekt vom Typ `File` (dies ist der Java-Typ), und rufen für jede Zeile der zugehörigen Datei eine Closure auf, die den übergebenen Parameter `line` auf die Konsole ausgibt.

Hier zeigt sich auch die Einfachheit, die durch Kombination verschiedener Sprachelemente von Groovy entsteht.

Wir werden viele der Spracherweiterungen im Laufe des Buches kennenlernen. Außerdem finden Sie eine Auflistung der Erweiterungen in Anhang A.

2.6.3 Bibliotheken (Beispiel Umgang mit XML)

Zusätzlich zu den anderen Erweiterungen von Java bietet Groovy auch Bibliotheken an, die viele Programmieraufgaben noch weiter vereinfachen. Ein Beispiel hierfür ist die Verarbeitung von XML, die allgemein in Groovy sehr elegant gelöst ist. Wir konzentrieren uns hier auf das Einlesen von XML-Daten mit der Klasse `XmlSlurper`.

Betrachten wir dazu die einfache XML-Struktur in Listing 2–10.

Listing 2–10 *XML-Struktur als Eingabe für XmlSlurper*

```
<groovy>
  <music>
    <tune>
      <title>The 59th Street Bridge Song</title>
      <artist>Simon and Garfunkel</artist>
    </tune>
    <tune>
      <title>Monday Monday</title>
      <artist>The Mamas and The Papas</artist>
    </tune>
    <tune>
      <title>Goodnight Moon</title>
      <artist>Shivaree</artist>
    </tune>
    <tune>
      <title>Suddenly</title>
      <artist>Soraya</artist>
    </tune>
  </music>
</groovy>
```

und ein Programm (in Listing 2–11), das dieses XML liest und die Titel mitsamt den Sängern ausgibt:

Listing 2–11 *Das Verarbeiten einer XML-Struktur in Groovy*

```
result = new XmlSlurper().parse(new File(args[0]))    ❶
result.music.tune.each {    ❷
   elem -> println "$elem.title, $elem.artist"
}
```

Ausgabe

```
The 59th Street Bridge Song, Simon and Garfunkel
Monday Monday, The Mamas and The Papas
Goodnight Moon, Shivaree
Suddenly, Soraya
```

In diesem Beispiel erzeugen wir ein Objekt der Klasse `XmlSlurper` ❶, einer Bibliotheksklasse, die XML-Dateien einliest und einen Objektbaum erzeugt, auf dem in Folge operiert werden kann (dieser wird im Beispiel der Variablen `result` zugewiesen). In der zweiten Zeile sehen wir ein Beispiel für die Verwendung der `GPath`-Notation ❷, einer Anfra-

genotation ähnlich der von XPath, die aber in der Sprache integriert ist (und mit Hilfe des Meta Object Protocol implementiert wurde). Jede ».«-Notation entspricht der Navigation zu einem oder mehreren Kindelementen. Den Zugriff auf die Inhalte der einzelnen Kindelemente zeigt der Ausgabebefehl.