tionalität an die konkreten Bedürfnisse anpassen lässt. So können zusätzliche Währungen, Konversionen, Rundungen, Formate oder Implementationen für Beträge ergänzt oder das Komponenten-Bootstrapping angepasst werden.

Fazit

JSR 354 definiert ein übersichtliches und zugleich mächtiges API, das den Umgang mit Währungen und Geldbeträgen stark vereinfacht, aber auch die teilweise widersprüchlichen Anforderungen. Auch fortgeschrittene Themen wie Runden und Währungsumrechnung sind adressiert und es ist möglich, Geldbeträge beliebig zu formatieren. Die bestehenden Erweiterungspunkte schließlich erlauben es, weitere Funktionen elegant zu integrieren. Diesbezüglich lohnt sich auch ein Blick in das OSS-Projekt [6], wo neben finanzmathematischen Formeln auch eine mögliche Integration mit CDI experimentell zur Verfügung steht.

JSR 354 sollte bis spätestens Ende des Jahres finalisiert sein. Für Anwender, die noch mit Java 7 arbeiten, wird zusätzlich ein vorwärtskompatibler Backport zur Verfügung stehen.

Weiterführende Links

[1] https://jcp.org/en/jsr/detail?id=354, https://java.net/projects/javamoney
[2] http://www.iso.org/iso/home/standards/currency_codes.htm
[3] http://stackoverflow.com/questions/3730019/why-not-use-double-or-float-to-represent-currency
[4] https://github.com/JavaMoney/jsr354-api
[5] https://github.com/JavaMoney/jsr354-ri
[6] http://javamoney.org

Anatole Tresch
anatole.tresch@credit-suisse.com

Nach dem Wirtschaftsinformatik-Studium an der Universität Zürich war Anatole Tresch mehrere Jahre lang als Managing-Partner und Berater tätig. Er sammelte weitreichende Erfahrungen in allen Bereichen des Java-Ökosystems vom Kleinunternehmen bis zu komplexen Enterprise-Systemen. Schwerpunkte sind verteilte Systeme sowie die effiziente Entwicklung und der Betrieb von Java EE in der Cloud. Aktuell arbeitet Anatole Tresch als technischer Architekt und Koordinator bei der Credit Suisse. Daneben ist er Specification Lead des JSR 354 (Java Money & Currency) und Co-Spec Lead beim Java EE Configuration JSR.

http://ja.ijug.eu/14/4/5

Scripting in Java 8

Lars Gregori, msgGillardon AG

Die Java Virtual Machine (JVM) unterstützt neben unterschiedlichen Programmiersprachen seit Längerem auch die Integration von Skriptsprachen. Mit Java 8 und der komplett überarbeiteten JavaScript-Engine „Nashorn" geht die JVM einen Schritt weiter in Richtung „Multi-Sprachen-Unterstützung". Für den Entwickler und Architekten lohnt es sich jetzt, sich mit dem Thema „Skriptsprachen" und der Motivation für deren Verwendung zu befassen. Die bisherige Performance galt für viele als Hinderungsgrund – Java 8 und die Einführung der Lambda-Ausdrücke nehmen hier positiven Einfluss.

Skriptsprachen werden für kleine und überschaubare Programme benutzt. Die Syntax der Programmiersprache verzichtet meistens auf bestimmte Sprachelemente wie zum Beispiel die der Variablen-Deklaration. Man spricht in diesem Fall von einer dynamischen Typisierung. Der Typ wird zur Laufzeit an den Wert gehängt. Dadurch lassen sich Prototypen und kleine Programme schnell erstellen. Java hingegen ist eine stark statisch typisierte Sprache. Der Compiler überprüft den Typ der Variable und somit benötigt jede Variable einen Typ. Zudem lassen sich lediglich Werte von diesem Typ zuweisen. Schwach typisierte Sprachen hingegen erlauben eine implizite Umwandlung unterschiedlicher Datentypen.

Die Betrachtung der daraus resultierenden Code-Qualität ist eher subjektiv. Skriptsprachen sind kompakter und lesbarer, da es weniger Codezeilen gibt. Zudem entfällt der zusätzliche Aufwand der Typ-Deklaration. Ob man sich der Argumentation hingeben soll, dass weniger Code und weniger Komplexität zu weniger Fehlern führen? Andersherum werden durch die Typ-Deklarationen zwar Schreibfehler beim Kompilieren des Quellcodes entdeckt, dadurch wird allerdings die Qualität der Anwendung nicht zwangsläufig besser. Qualität entsteht hauptsächlich durch Tests. Deren Erstellung ist weder durch eine dynamische noch durch eine statische Typisierung eingeschränkt. Dabei können aber die kleinen und kompakten

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("rhino");
Integer result = (Integer) engine.eval("6 * 7");
System.out.println(result);
```

Listing 1

Code-Stücke einer dynamischen Sprache das Schreiben von Tests fördern.

Bei der Performance sieht die Sache klarer aus. Java-Code, der als Byte-Code kompiliert ist, ist schneller als Skript-Code, der in Java ausgeführt wird. Jedoch hat die Einführung von „invokedynamic" in Java 7 zu einer deutlichen Verbesserung geführt. Dazu später.

Warum Skriptsprachen?

In „The Pragmatic Programmer" wird empfohlen, dass man jedes Jahr eine neue Sprache lernt. Skriptsprachen können etwas Neues und Anderes sein und auch Spaß machen. Mit ihnen lassen sich Programmteile in Konfigurationsdateien und Datenbanken auslagern und Anpassungen könnten zum Beispiel durch den Fachbereich erfolgen. Dadurch kann sich auch das Deployment der Anwendung erleichtern, indem sie bei kleineren Regeländerungen nicht komplett neu ausgerollt werden muss. Trotzdem sollte ein Prozess hinterlegt werden, der sicherstellt, dass Änderungen auf dem Produktivsystem nicht zu dessen Ausfall führen.

Trotz der vielen Java-Bibliotheken bieten verschiedene Programmiersprachen im Gegensatz zu Java die passendere Lösung zu einem Problem. Jython, die Java-Version von Python, und Sleep (Perl) haben ihre Stärke in der String-Manipulation. ABCL (Common Lisp) und Clojure bringen die „map"- und „reduce"-Methoden mit, die mit den Lambda-Ausdrücken nun allerdings auch in Java 8 Einzug halten. Groovy hat mit GroovySQL und GroovyMarkup eigene Klassen für die SQL- und HTML-Erzeugung.

Skriptsprachen bieten auch eine Vielzahl von Bibliotheken an, die verschiedene Aufgaben in bestehenden Anwendungen lösen. In einem nachfolgenden Beispiel wird auf das Template-Framework Mustache eingegangen. Man sollte aber trotzdem bedenken, dass Skriptsprachen mächtig sind und prinzipiell auf alles zugreifen können.

In Web- und Browser-Anwendungen ist JavaScript im Frontend als Standard gesetzt. Durch die Integration ins Backend, die einer der Gründe für den Erfolg von „Node.js" ist, erfordert die Frontend- und Backend-Entwicklung keine Trennung anhand der benutzten Programmiersprachen. Der Frontend-Entwickler kann Aufgaben aus dem Backend übernehmen, ohne dass er eine neue Sprache lernen muss. So lässt sich zum Beispiel die im Browser für die Überprüfung der Eingabefelder verwendete JavaScript-Datei auch im Backend nutzen. Änderungen bei der Eingabe-Validierung müssen nicht in Java nachgezogen werden. Es wird auch nicht automatisch das Backend vergessen, da zum Beispiel in einer Anforderungsänderung für die Altersprüfung nur von der Oberfläche gesprochen wird.

Skriptsprachen lassen sich auch in den Build- und Test-Prozess einbinden. Hier werden die notwendigen Bibliotheken meist einfacher hinzugefügt. Für die Verwendung in der eigentlichen Anwendung können bestimmte Freigaberegeln gelten.

Scripting in Java

Für die Integration von Skriptsprachen gibt es verschiedene Möglichkeiten. Groovy bietet zum einen mit „groovyc" einen Compiler, um „groovy"-Dateien in „class"-Dateien zu kompilieren. Auf diese Klassen kann wie in Java zugegriffen werden. Typenlose Rückgabewerte müssen aber in Java in einen entsprechenden Java-Typ umgewandelt werden. Zudem implementiert die kompilierte Klasse das „GroovyObject"-Interface, wodurch weiterhin eine Abhängigkeit zu Groovy bestehen bleibt.

Eine weitere Möglichkeit ist eine eigene Skript-Engine der Skript-Sprachen. Die „GroovyShell"-Klasse ermöglicht es, durch die „evaluate"-Methode Groovy-Skripte auszuführen. Java- und Groovy-Variablen können mit der „Binding"-Klasse gebunden werden. Dadurch lassen sich Werte in beide Richtungen austauschen.

Das „Bean Scripting"-Framework (BSF), ein Apache-Projekt, definiert eine Schnittstelle für Skriptsprachen, um in Java ausge-

führt zu werden und auf Java-Objekte und -Methoden zuzugreifen. Dabei wird beim „eval"-Methodenaufruf der „BSFManager"-Klasse der Name der Skript-Engine übergeben. Dieser ist innerhalb der BSF-Umgebung registriert und ruft dann die entsprechende Klasse mit dem übergebenen Skript auf.

Der Java Community Process, der Java Specification Requests (JSR) definiert, hat mit dem JSR 223 (Scripting for the Java Platform) ein ähnliches Prinzip wie BSF in Java integriert. Dadurch steht ab Java 6 das „javax.script"-Package zur Verfügung. Der „ScriptEngineManager" liefert mit der „getEngineByName"-Methode die entsprechende „ScriptEngine" zurück. Diese kann dann mit „eval" das Skript ausführen. Sowohl Skript- als auch Java-Variablen können gebunden werden und stehen beiden Seiten zur Verfügung. Die Java-Integration bietet auch für JVM-Skriptsprachen die Möglichkeit, auf andere JVM-Skriptsprachen direkt zuzugreifen. So kann zum Beispiel ein Groovy-Skript JavaScript-Code ausführen und ebenfalls auf die Variablen zugreifen.

Wer mehr über Scripting in Java erfahren möchte, dem sei das Buch „Scripting in Java: Languages, Frameworks, and Patterns" von Dejan Bosanac (ISBN 978-0321321930) empfohlen. Nachfolgend ist ein JSR-223-Beispiel aufgeführt, das eine Berechnung in JavaScript ausführt (siehe Listing 1).

Das Beispiel verwendet „Rhino" als Engine. Dabei handelt es sich um die in Java 6 integrierte JavaScript-Engine. Sie wurde in Java 8 komplett überarbeitet und heißt jetzt „Nashorn".

Nashorn

Die JavaScript-Engine „Nashorn" wurde im ECMAScript-Standard (siehe „http://www.ecma-international.org/ecma-262/5.1") implementiert. Der Unterschied zwischen ECMAScript und JavaScript besteht darin, dass JavaScript eine Implementierung des ECMAScript-Standards ist.

Nashorn ist die Referenz-Implementierung für den JSR 292 (Supporting Dynamically Typed Languages on the Java Platform), um dynamische Sprachen auf der Java-Plattform zu unterstützen. Der Weg dorthin wurde bereits mit Java 7 gelegt, indem mit der JVM der Byte-Code-Befehl „invokedynamic" eingeführt wurde. Dabei handelt es sich um eine Erweiterung der bereits bestehenden Invocation-Operatoren. So wird zum Beispiel „invokestatic" für statische Methodenaufrufe und „invokevirtual" für Methodenaufrufe einer Instanz-Klasse verwendet.

Im Gegensatz zu „invokedynamic" prüft der Compiler die Verwendung des richtigen Daten-Typs und bindet die entsprechenden Methodenaufrufe fest im erzeugten Byte-Code. Bei dynamisch typisierten Sprachen hingegen ermittelt „invokedynamic" den Typ zur Laufzeit und bindet die entsprechende Klasse und Methode dynamisch.

Beispiel „Mustache"

Mustache (siehe „http://mustache.github.io") ist ein Template-System, das für unterschiedliche Sprachen zur Verfügung steht. Da die Syntax auf „if" else"-Ausdrücke und Schleifen verzichtet, bezeichnet sich Mustache als „logic-less" Template-System. Der Name „Mustache", zu deutsch Oberlippenbart oder Schnauzbart, rührt daher, dass geschweifte Klammern für die Platzhalter verwendet werden. Dreht man die öffnende Klammer um 90 Grad, ähnelt dies einem Mustache. Das folgende Beispiel Template (template.mst) zeigt zu einem Kontakt den Namen und die dazugehörigen Telefonnummern (siehe Listing 2). Die Kontaktdaten sind als JSON-Daten in diesem Format hinterlegt (siehe Listing 3).

Wie im obigen Beispiel ermittelt der „ScriptEngineManager" die „ScriptEngine". Diese ist in dem Fall eine „NashornScript Engine"-Instanz, sie implementiert das „javax.script.Invocable"-Interface und stellt die Methode „invokeMethod" zur Verfügung. Zunächst wird das Template eingelesen und dann die Kontaktdaten. Diese liegen im JSON-Format in der Datei „contact.json" vor und lassen sich über das

```
Telefonnummer von {{contact.
name}}:
{{#contact.phone_numbers}}
* {{type}}: {{number}}
{{/contact.phone_numbers}}
```

Listing 2

```
{
 "contact":{
   "name":"Lars Gregori",
   "phone_numbers":[
     {
       "type":"daheim",
       "number":"+49 555 5555"
     },
     {
       "type":"privat",
       "number":"+49 555 2309"
     }
   ]
 }
}
```

Listing 3

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine nashorn = manager.getEngineByName("nashorn");
Invocable invocable = (Invocable) nashorn;

String template = new String(
         Files.readAllBytes(
                     Paths.get("template.mst")));

String contactJson = new String(
         Files.readAllBytes(
                     Paths.get("contact.json")));
Object json = nashorn.eval("JSON");
Object data = invocable.invokeMethod(
                     json, "parse", contactJson);

nashorn.eval(new FileReader("src/mustache.js"));
Object mustache = nashorn.eval("Mustache");

Object output = invocable.invokeMethod(
         mustache, "render", template, data);
System.out.println(output);
```

Listing 4

```
Telefonnummer von Lars Gregori:
* daheim: +49 555 5555
* privat: +49 555 2309
```

Listing 5

```
List<Object> someObjects = new ArrayList<Object>();
someObjects.add(22);
someObjects.add(„hallo“);
someObjects.forEach(obj -> System.out.println(obj));
```

Listing 6

```
...
30: invokedynamic #36,  0
// InvokeDynamic #0:accept:()Ljava/util/function/Consumer;
...
```

Listing 7

Nashorn-interne JSON-Objekt in Objektdaten umwandeln. Dazu wird mit der „eval“-Methode das interne JSON-Objekt geholt und der „invokeMethode“ übergeben. Als zweiter Parameter des „invokeMethode“-Aufrufs wird der aufzurufende Methoden-Name angegeben. In diesem Fall ist es die „parse“-Methode. Als Parameter für den „parse“-Aufruf werden die eingelesenen Kontaktdaten als String mitgegeben (siehe Listing 4).

Die „eval“-Methode liest die eigentliche Mustache-JavaScript-Datei „mustache.js“ (siehe „https://github.com/janl/mustache.js“) ein und führt sie intern aus. Durch den „eval(„Mustache“)“-Aufruf steht das Mustache-Objekt zur Verfügung. An diesem kann wieder mithilfe von „invokeMethod“ die „render“-Methode mit den Template- und Kontaktdaten als Parameter aufgerufen werden. Das „render“-Ergebnis lässt sich dann weiterverwenden. In diesem Fall wird es mit „println“ ausgegeben (siehe Listing 5).

Nashorn intern

Der Nashorn-Parser parst zunächst den JavaScript-Code und baut einen internen Baum auf, der das Programm repräsentiert. Innerhalb des Baums werden Ausdrücke durch Java-Aufrufe ersetzt und Aufrufe vereinfacht. Für die Daten werden die Verwendung, der Speicher und der Typ ermittelt. Im nächsten Schritt erzeugt der Code-Generator über die ASM-Bibliothek (siehe „http://asm.ow2.org“) Byte-Code. Dieser bleibt dann wie bei einem kompilierten Java-Programm im Speicher, wird mit dem Class Loader geladen und steht der Anwendung zur Verfügung. Er verhält sich wie kompilierter Java-Code, den die JVM automatisch optimieren kann.

Lambda-Ausdrücke

Das in Java 7 eingeführte „invokedynamic“ war zunächst nur eine Erweiterung der JVM für Skriptsprachen. Java als Sprache profitiert seit Java 8 mit den neu eingeführten Lambda-Ausdrücken auch davon. Dabei handelt es sich um anonyme Methoden, deren Typ erst zur Laufzeit ermittelt wird und die somit bestens für „invokedynamic“ geeignet sind (siehe Listing 6).

Im Beispiel wird eine Liste von unterschiedlichen Objekten angelegt. Mit der in Java 8 neu hinzugekommenen „forEach“-Methode wird über die einzelnen Elemente der Liste gegangen und an einen Lambda-Ausdruck übergeben. Die Syntax hierfür besteht aus einer Parameterliste, einem Pfeil-Operator und einem Ausdruck, der auf die Parameter zugreifen kann. Im Beispiel wird das Element als „obj“-Parameter übergeben und mit der „println“-Methode ausgegeben.

Mit dem im JDK enthaltenen Kommandozeilen-Tool „javap“ lässt sich eine kompilierte Klasse „disassembeln“ und deren Bytecode betrachten. Im Listing 7 ist ein stark gekürzter Ausschnitt des Lambda-Beispiels, das mit „javap -c -v SomeObjects.class“ aufgerufen wurde und in dem der „invokedynamic“-Aufruf zu sehen ist.

Fazit

Die JVM entwickelt sich mehr und mehr zu einer Multi-Sprachen-Umgebung, in der dynamische Skriptsprachen ihre Unterstützung finden. Zudem wurde in Java 8 mit den Lambda-Ausdrücken ein neues Sprachelement aus der funktionalen Programmierung übernommen. Der Schlüssel hierzu war die Einführung von „invokedynamic“ und die damit verbundene Steigerung der Laufzeit. Dabei handelt es sich aber nur um den ersten Schritt des noch von Sun gegründeten „Da Vinci Machine“-Projekts (siehe „http://openjdk.java.net/projects/mlvm“). Die Idee dahinter ist, dass die JVM den Entwickler einer Programmiersprache unterstützt, um bestimmte Konstrukte der Programmiersprache nicht mehr umständlich nachbauen zu müssen. Hier war „invokedynamic“ ein erster Schritt, um dynamische Aufrufe zur Laufzeit zu binden und so die Performance zu steigern. Mit Nashorn ist damit eine konkurrenzfähige JavaScript-Engine entstanden. Dies ermöglicht es, „Node.js“-Anwendungen mit „Avatar.js“ (siehe „https://avatar-js.java.net“) in Java integrieren zu können und somit die JVM weiter in Richtung „Multi-Sprachen-Umgebung“ zu bringen.

Lars Gregori
lars.gregori@msg-gillardon.de

Lars Gregori ist Lead IT Consultant bei der msgGillardon AG in Ismaning/München. Als Software-Architekt beschäftigt er sich im JEE- und mobilen Bereich für Finanzunternehmen. Im Center of Competence für IT-Architekturen hat er Technologien von heute und morgen im Fokus. Zudem gilt sein Interesse unterschiedlichen Programmiersprachen.

http://ja.ijug.eu/14/4/6