



Mehr Dynamik bitte!

Einbindung von Skriptsprachen in Java

Dirk Weil

In Softwareentwicklungsprojekten stellt sich häufig die Aufgabe, Teile der Gesamtanwendung so an die gewünschte Umgebung anpassbar zu machen, dass algorithmische Änderungen auch ohne ein neues Deployment in Betrieb genommen werden können. Wurde ein solches Customizing früher beispielsweise durch die Entwicklung einer eigenen Formelsprache gelöst, steht seit Java 6 die Möglichkeit zur Einbindung von Skripten in Java-Anwendungen zur Verfügung. Wir möchten im Folgenden darstellen, wie die Skripte mit der Java-Umgebung interagieren können und für welchen Einsatzzweck sich die verfügbaren Sprachen eignen. Und nicht zuletzt werfen wir einen Blick auf die Performanz der Skripte.

Dynamische Anpassung einer Java-Anwendung

Der konkrete Anlass für unsere Aufgabenstellung stammt aus einem Projekt, in dem wir für einen unserer Kunden eine Anwendung zur Betriebsdatenerfassung und Leistungslohnermittlung entwickeln. Gerade in letzterem Bereich sind die Regeln zur Lohnermittlung sehr vielfältig und häufigen Änderungen unterworfen, sodass eine Programmierung innerhalb der Geschäftslogik der Java-Anwendung ungünstig erscheint – jede Anpassung der Berechnungsregeln hätte dann ein erneutes Deployment der Anwendung zur Folge. Zudem sollte eine Bearbeitung der Lohnermittlungsalgorithmen auch für Mitarbeiter der jeweiligen Fachabteilung möglich sein. Hier können wir aber nicht davon ausgehen, dass die notwendigen Kenntnisse und Werkzeuge zur Programmierung mit Java und zum Deployment auf einem Java-EE-Server vorhanden sind.

Gesucht ist also zunächst eine Programmiersprache, die keine besonderen Entwicklungswerkzeuge wie Compiler, Entwicklungsumgebung usw. benötigt, sodass der Entwickler idealerweise nur einen einfachen Texteditor benötigt. Die damit erstellten Programme – nennen wir sie schon mal Skripte – sollen dann leicht mit einer Java-EE-Anwendung verbunden und deployt werden können. Schön wäre für diesen Aspekt, wenn keine spezielle Ablaufumgebung benötigt wird, die Skripte also auf der Java-VM ablaufen können. Eine weitere wesentliche Anforderung ist eine enge Integration in die Java-Anwendung: Java-Laufzeitobjekte müssen Skript-Funktionen aufrufen können und umgekehrt.

Scripting for the Java™ Platform

Seit Java 6 gibt es nun die standardmäßige Möglichkeit, Skripte verschiedener Sprachen einzubinden. Dies ist das Ergebnis des JSR 223 „Scripting for the Java™ Platform“ und manifestiert sich im Paket `javax.script` der Standardbibliothek. Im Lieferumfang ist dabei mit Rhino eine Engine und Sprachimplementierung für JavaScript enthalten. Andere Skriptsprachen lassen sich auf einfache Weise hinzu konfigurieren. Als ersten Einstieg empfiehlt sich die Homepage des Skripting-Projektes [Scripting]. Hier findet man auch die Datei `jsr223-engines.zip` zum Download, die die sogenannten Engines für etwa zwei Dutzend Skriptsprachen enthält. Diese stellen die Anbindungsadapter gemäß JSR 223 dar. Zusätzlich werden die eigentlichen



Implementierungen der Skriptsprachen benötigt, die man sich separat besorgen muss.

Die auf der genannten Seite enthaltene Liste von Skriptsprachen erhebt keinen Anspruch auf Vollständigkeit: Auch an anderen Stellen werden JSR-233-kompatible Sprachen gepflegt, darunter durchaus auch solche, die eine native Implementierung benötigen. Aus den oben genannten Gründen werden wir uns im Folgenden aber nur mit Sprachen beschäftigen, die die Java-Laufzeitumgebung nutzen.

Schauen wir uns zunächst einmal die grundsätzliche Vorgehensweise zur Ausführung eines Skriptes an:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

...

// Manager fuer Script Engines erzeugen
ScriptEngineManager sem = new ScriptEngineManager();

// Engine fuer JavaScript laden
ScriptEngine engine = sem.getEngineByName("javascript");

// Script inline ausfuehren
engine.eval("print('Hello, JavaScript!');");

// Script aus Datei ausfuehren
engine.eval(new FileReader("hello.js"));

...
```

Mit Hilfe des `ScriptEngineManagers` erhält man Zugriff zur gewünschten Engine – hier zur eingebauten Rhino-Engine. Die Methode `eval` akzeptiert als Parameter ein Skript in Form einer Zeichenkette oder eines Eingabestroms und führt es aus.

Die Verwendung einer anderen Sprache anstelle von JavaScript gestaltet sich denkbar einfach. Soll beispielsweise Groovy eingesetzt werden, benötigt man die Bibliotheken `groovy-engine.jar` aus der oben erwähnten Datei `jsr223-engines.zip` sowie `groovy-all.jar` aus der Groovy-Installation im Classpath. Die erste Bibliothek ist der JSR-223-Adapter zur Sprachimplementierung in der zweiten Bibliothek. Solch ein Bibliothekspaar wird für jede Skriptsprache benötigt, die aus der Java-Anwendung heraus aufgerufen werden soll. Wir betrachten hier, wie bereits erwähnt, nur Sprachen, die auf der Java-VM laufen. Andere Sprachen benötigen gegebenenfalls eine umfangreichere Installation.

Der `ScriptEngineManager` kann befragt werden, welche `ScriptEngines` verfügbar sind:

```
ScriptEngineManager sem = new ScriptEngineManager();
List<ScriptEngineFactory> factories = sem.getEngineFactories();

for (ScriptEngineFactory scriptEngineFactory : factories) {
    System.out.printf("Script Engine: %s (%s)\n",
        scriptEngineFactory.getEngineName(),
        scriptEngineFactory.getEngineVersion());
    System.out.printf("  Language: %s (%s)\n",
        scriptEngineFactory.getLanguageName(),
        scriptEngineFactory.getLanguageVersion());

    for (String alias : scriptEngineFactory.getNames()) {
        System.out.printf("    Alias: %s\n", alias);
    }
}
```

Das gezeigte Programmsegment erzeugt etwa diese Ausgabe:

```
Script Engine: Pnuts (1.0)
Language: Pnuts (1.2)
Alias: pnuts
Script Engine: Mozilla Rhino (1.6 release 2)
Language: ECMAScript (1.6)
Alias: js
Alias: rhino
Alias: JavaScript
Alias: javascript
Alias: ECMAScript
Alias: ecmaascript
Script Engine: groovy (1.5.6)
Language: groovy (1.5.6)
Alias: groovy
```

Interaktion mit der Java-Umgebung

Nun ist das Starten von Skripten zwar interessant, verlöre aber schnell seinen Reiz, wenn man nicht zwischen dem aufrufenden Java-Programm und dem Skript Werte transferieren könnte. Dies ist zunächst einmal möglich mit Hilfe der sogenannten **Bindings**, die als Implementierung von `Map<String, Object>` in der Lage sind, Key-Value-Paare aufzunehmen. Es gibt je ein `Bindings`-Objekt auf globaler Ebene und in der genutzten Engine, zugreifbar mit der Methode `getBindings` in `ScriptEngineManager` bzw. `ScriptEngine`. Für den einfachen Zugriff auf die Key-Value-Paare gibt es in beiden Interfaces die von `Map` bekannten Methoden `get` und `put`.

Diese `Bindings`-Einträge lassen sich nun innerhalb des Skriptes verwenden. Umgekehrt werden vom Skript erzeugte Werte im `Bindings`-Objekt der Engine abgelegt. Damit ist ein Austausch von Werten zwischen Java-Programm und Skript möglich:

```
// Script-Variablen besetzen
engine.put("netto", 100);
engine.put("steuersatz", 0.19);

// Script ausführen
engine.eval("brutto = netto * steuersatz");

// Script-Variablen auslesen
Object result = engine.get("brutto");
```

Das gezeigte Beispiel arbeitet mit dem `Bindings`-Objekt der `ScriptEngine`; die Methoden `get` und `put` sind Convenience-Methoden für den Aufruf der jeweils gleichnamigen Methode in `getBindings` (`ScriptContext.ENGINE_SCOPE`). Die in einem `Bindings`-Objekt enthaltenen Werte sind aus Sicht des Skriptes globale Daten vergleichbar mit den im Application- bzw. Session-Scope einer Webanwendung abgelegten Werten.

Eine weitere Möglichkeit besteht darin, Skripte bzw. Teile daraus als Funktionen aufzurufen. Dazu wird das Skript zunächst einmalig ausgeführt und damit quasi kompiliert. Anschließend können im Skript definierte Funktionen aus dem Java-Programm heraus aufgerufen werden:

```
// Script compilieren
engine.eval(
    "function factorial(n) { return n==1 ? 1 : n*factorial(n-1) };");

Invocable invocable = (Invocable) engine;

// Funktion aufrufen
Object result = invocable.invokeFunction("factorial", 5);
```

In den Beispielen wurden nur primitive Daten bzw. Objekte der entsprechenden Wrapper-Klassen (`int` bzw. `java.lang.Integer`) zwischen Java und Skript ausgetauscht. Darauf ist man allerdings nicht eingeschränkt; es können vielmehr beliebige Objekte übergeben werden:

```
// ShowCurrentDate.java
// Script Engine laden
ScriptEngineManager sem = new ScriptEngineManager();
ScriptEngine engine = sem.getEngineByName("groovy");

// Script öffnen
InputStream scriptInputStream = ClassLoader.getResourceAsStream(
    "scripts/showdate.groovy");

// Script compilieren
engine.eval(new InputStreamReader(scriptInputStream));

// Funktion aufrufen
Invocable invocable = (Invocable) engine;
invocable.invokeFunction("showDate", Calendar.getInstance());

// showdate.groovy
import java.util.Calendar;

def showDate(timestamp) {
    System.out.println("Jahr: " + timestamp.get(Calendar.YEAR))
    System.out.println("Monat: " + timestamp.get(Calendar.MONTH))
    System.out.println("Tag: " + timestamp.get(Calendar.DAY_OF_MONTH))
}
```

Das Beispiel – diesmal mit Hilfe von Groovy als Skriptsprache formuliert – zeigt eindrücklich die enge Integration mit dem Java-Laufzeitsystem: Es kann problemlos ein Objekt des Typs `java.util.Calendar` übergeben und im Skript verarbeitet werden. Ebenso nahtlos erscheinen andere Java-Klassen – hier beispielsweise `java.lang.System` – wie integrale Bestandteile der Skriptsprache.

Skriptsprachen-Vorteile

Was bringt uns denn nun der Einsatz einer Skriptsprache? Einerseits natürlich die eingangs erwähnten Eigenschaften wie einfache Bearbeitung und unkompliziertes Deployment der Skripte. Andererseits sind Skriptsprachen vielfach mit einem Typsystem versehen, das nicht so strikt wie das von Java ist, wo beispielsweise Objekte vor ihrer Verwendung deklariert werden müssen oder Methoden nur aufgerufen werden können, wenn der deklarierte Typ es zulässt. Dieses starre Typsystem ist zweifellos eine der starken Eigenschaften der Sprache Java und sorgt für eine hohe Stabilität insbesondere von umfangreichen Java-Anwendungen.

Für kleinere Baustellen – und dies ist der Anwendungsbereich für Skripte – wünscht man sich manchmal aber etwas mehr Laissez-faire. So erlauben die betrachteten Skriptsprachen die Verwendung von Variablen ohne vorherige Deklaration. Die typsyste-



zifischen Eigenschaften der Objekte sind aber dennoch benutzbar: Im zuvor gezeigten Groovy-Code ist die Parametervariable `timestamp` ohne Typ deklariert, vergleichbar einer Deklaration als `Object`. Trotzdem kann darin die Methode `get` (der Klasse `Calendar`) aufgerufen werden.

Weiterhin bieten Skriptsprachen gegenüber Java einige interessante Ergänzungen in der Behandlung von häufig genutzten Datenstrukturen wie Listen und Maps:

```
myList = [1776, -1, 33, 99, 0, 928734928763]
println myList[0] // 1776
println myList.size() // 6
println myList // [1776, -1, 33, 99, 0, 928734928763]

scores = [ "Brett":100, "Pete":"Did not finish", "Andrew":86.87934 ]
println scores["Pete"] // Did not finish
println scores.Andrew // 86.87934
```

Closures – also Funktionsobjekte wie für Java angekündigt – sind in den angesprochenen Skriptsprachen bereits enthalten:

```
3.times { println 'Hi' }

[1, 2, 3].each { number ->
    println number
}

["eins", "zwei", "drei"].each { println it }

def printit = { println it }
["one", "two", "three"].each printit
```

Zielgruppe

Die Beispiele zeigen Groovy-Code. Die Syntax anderer Skriptsprachen ist ähnlich, wenn nicht häufig sogar deckungsgleich. Das verwundert auch nicht wirklich, haben wir uns doch in der Auswahl der Sprachen auf die beschränkt, die auf der Java-VM laufen. Diese Sprachen haben daher natürlicherweise eine Ähnlichkeit untereinander und auch zu Java. Dies ist für diejenigen mit Entwicklerhintergrund eine sehr angenehme Eigenschaft, die eine schnelle Einarbeitung in die Skriptsprache erlaubt. Hier wird aber auch deutlich, für wen diese Sprachen gemacht sind: für Entwickler eben. Ein Anwender dagegen, der sich beispielsweise im Controlling durchaus virtuos mit umfangreichen Excel-Spreadsheets herumschlägt, aber nur gelegentlich Algorithmen (z. B. Makros) programmiert, wird sich mit der Programmierung von Skripten nicht leicht tun.

Performanz

Für die Auswahl der Skriptsprache ist natürlich auch interessant, wie schnell ihre Skripte abgearbeitet werden. Um dazu einen ers-

ten Eindruck zu erhalten, haben wir die im Beispiel oben gezeigte Version von `factorial` in Java und in verschiedenen Skriptsprachen programmiert, damit 100 000 mal die Fakultät von 20 berechnet und die Laufzeiten gemessen. Den Programmcode haben wir hier aus Platzgründen nicht abgedruckt. Sie können ihn sich inklusive der anderen Beispiele von [SRC] herunterladen.

Abbildung 1 zeigt, dass man natürlich damit rechnen muss, dass Skripte deutlich langsamer ablaufen als ein Java-Programm. Die Faktoren von ca. 18 im Falle von Pnuts oder selbst gut 200 bei Groovy sind aber nicht grundsätzliche Show-Stopper. Auffallend ist allerdings, dass die Performanz der verschiedenen Sprachen recht deutlich voneinander abweicht. Danach wäre für eine laufzeitintensive Anwendung am ehesten Pnuts geeignet.

Unser kleiner Benchmark prüft natürlich nur bestimmte wenige Aspekte der Sprachen ab und ist damit nicht generell aussagekräftig. Im Internet veröffentlichte Messungen bestätigen allerdings die grobe Tendenz unseres Ergebnisses (s. z. B. [Pnuts]).

Fazit

Mit der Integration von Skriptsprachen gewinnt die Java-Plattform eine weitere leistungsfähige Eigenschaft hinzu. Damit lassen sich auf einfache Weise Teile des Gesamtalgorithmus so auslagern, dass sie ohne aufwändige Entwicklungs- und Deploymentumgebung bearbeitet werden können. Es stehen viele Sprachen zur Verfügung, die auf der Java-VM ablaufen und damit die Plattformneutralität einer Java-Anwendung nicht beschädigen. Vor die intensive Nutzung von Skripten muss allerdings eine Laufzeitprüfung gestellt werden.

Ausblick

Skripte öffnen nicht nur die Tür für ein bequemes Customizing von Anwendungen, sondern auch für böswillige Eingriffe in die Anwendungslogik. Mit welchen Risiken man dabei rechnen muss, wäre einen eigenen Artikel wert. Ein weiteres interessantes Thema ist die Ausgestaltung von Skriptumgebungen, sodass sie für eine bestimmte Fachanwendung gut einsetzbar wird – Stichwort Domänenspezifische Sprachen.

Links

[Pnuts] N. Sweet, Pnuts Highlights,

<http://pnuts.org/articles/pnutsHighlights.html>

[Scripting] scripting: Project Home, <https://scripting.dev.java.net>

[SRC] Programmcode von factorial und der übrigen Beispiele des Artikels, <http://www.gedoplan.de/mehrdynamikbitte>



Abb. 1: Laufzeit von factorial in verschiedenen Skriptsprachen



Dirk Weil ist seit mehr als zehn Jahren als Berater im Bereich Java tätig. Als Geschäftsführer der GEDOPLAN GmbH in Bielefeld ist er für die Konzeption und Realisierung von Informationssystemen auf Basis von Java EE verantwortlich. Er ist Autor von Fachartikeln, hält Vorträge und leitet Seminare aus einem umfangreichen, eigenen Java-Curriculum.
E-Mail: dirk.weil@involva-gruppe.de.