JShell: REPL in Java 9

Thorsten Ludwig, inovex GmbH

Mit JShell hält endlich auch eine REPL in Java Einzug. Wer bereits aus anderen Sprachen die Vorzüge des interaktiven Entwickelns kennengelernt hat, wird das Release von Java 9 kaum noch abwarten können. Der Artikel zeigt, wie die REPL funktioniert, welche Möglichkeiten und Einschränkungen vorhanden sind, und gibt einen Ausblick, was mit passendem Tool-Support möglich sein wird.

Das Release von Java 9 steht kurz bevor, da lohnt sich ein Blick auf die neuen Features. Neben dem neuen Modulsystem namens „Jigsaw" ist JShell die interessanteste Neuerung. Damit gibt es endlich eine REPL in Java. REPL steht für „Read Eval Print Loop". Wer schon einmal Bekanntschaft mit funktionalen Programmiersprachen gemacht hat, wird das Konzept einer REPL kennen. Es handelt sich in der Regel um ein Command-Line-Tool, bei dem man Code eingibt, woraufhin dieser ausgewertet und das Ergebnis ausgegeben wird.

Damit lassen sich vor allem zwei Dinge erledigen: Zum einen eigener oder fremder Code erkunden, etwa das neue Projekt, das man übernimmt oder eine Bibliothek, die man noch nicht kennt; zum anderen kann man direkt in der REPL entwickeln, hier hat sich die Bezeichnung „REPL-Driven-Development" (RDD) etabliert. Der Vorteil: Man bekommt rasend schnell Rückmeldung, ob der Code das macht, was man sich vorgestellt hat. Mit einer guten REPL entfällt also die Notwendigkeit, eine Main-Methode oder Testfälle zu schreiben, nur um schnell etwas auszuprobieren.

Die ersten Gehversuche

Legen wir los, indem wir Java 9 installieren und dann im Terminal mit „jshell" die JShell starten. Es begrüßt uns ein blinkender Cursor, der auf Eingabe wartet. Um uns erstmal zurechtzufinden, tippen wir „/help" ein und bekommen eine Übersicht über alle Befehle. Ganz wichtig, um nicht in einer Falle wie in „emacs" zu landen: Mit „/exit" beenden wir die JShell wieder. Ansonsten gibt es einige Befehle, um verschiedene Einstellungen zu ändern oder unsere bisherigen Eingaben zu sehen. Praktisch ist die Anweisung „/history", die analog zum Linux-Befehl unsere komplette Eingabehistorie zeigt. Mit der Pfeiltaste nach oben können wir unsere bisherigen Eingaben wiederholen. Die restlichen wichtigen Befehle werden im Laufe des Artikels erwähnt. Geben wir jetzt erst mal ganz simpel „3+5;" ein, ein einfacher, aber gültiger Ausdruck in Java. *Listing 1* zeigt die Antwort.

Aufgrund unserer Mathematik-Kenntnisse wissen wir sofort, dass „8" das Ergebnis der Rechnung sein muss, aber was hat es mit „$1" auf sich? Hier handelt es sich um eine anonyme Variable, die JShell für uns angelegt hat. Jedes Mal, wenn wir etwas eingeben, das einen Wert zurückliefert, speichert die JShell das für uns unter „$x" ab, wobei „x" bei jedem Mal inkrementiert wird.

Die Variable lässt sich nun auch weiterverwenden *(siehe Listing 2)*. Beim letzten Listing sehen wir ein weiteres Killer-Feature der JShell: Man kann Semikolons weglassen! Wir wollen die JShell aber nicht nur als Taschenrechner benutzen, daher führen wir auch Methoden aus, etwa „substring" auf einem String *(siehe Listing 3)*.

Gibt man nur „.sub" ein und drückt dann Tabulator, hilft einem die JShell mit Vorschlägen *(siehe Listing 4)*. Tippt man nun ein weiteres „s" ein, wird einem mit einem Druck auf Tabulator der Name vervollständigt, ein erneutes Drücken schlägt einem die möglichen Signaturen vor *(siehe Listing 5)*.

Ein nochmaliges Tippen präsentiert die Dokumentation zu der Funktion. Wir können also nun mit der JShell die eingebauten Java-Funktionen erkunden. Schön wäre es auch, fremden Code auszuprobieren, etwa aus einer JAR.

I've got a JAR of code!

Als Beispiel nehmen wir die Commons-Lang-Library. Um sie zu benutzen, passen wir „class path" an *(siehe Listing 6)*. Die JShell spielt jetzt noch einmal alle Eingaben mit dem neuen „class path" durch. Nun können wir loslegen. Um uns Tipparbeit zu sparen, importieren wir erstmal „StringUtils". Schade – die Autovervollständigung funktioniert nur für eingebaute Packages, erst für den Klassennamen hilft uns die JShell: „jshell> import org.apache.commons.lang3.StringUtils". Wenn wir später einen Überblick über die Imports haben wollen, gibt uns „/imports" eine Liste über alle importierten Klassen aus.

Geben wir nun „StringUtils." ein, hilft uns die JShell mit Autovervollständigungen. Wir können die verschiedenen Methoden ausprobieren und bekommen die Signaturen vorgeschlagen. Leider bekommen wir von den importierten JARs keine Dokumentation angezeigt, selbst wenn wir die JAR mit den JavaDocs zusätzlich in den „class path" übernehmen.

Ähnlich funktioniert es, wenn wir unser eigenes Projekt in der JShell ausprobieren wollen. Als „class path" übernehmen wir dann einfach unser „build"-Verzeichnis. Um mehrere Verzeichnisse als „class path" zu übernehmen, trennen wir diese mit einem Doppelpunkt. Wollen wir Änderungen an unseren Klassen auch in der JShell sichtbar machen, müssen das Projekt neu gebaut und die Änderungen in die JShell übernommen werden, das geht am schnellsten mit dem Befehl „/reload". Die JShell startet übrigens mit dem Verzeichnis als „class path", in dem es gestartet wurde.

Ein Schritt Richtung RDD

Mit den bisher gesehenen Mitteln können wir nun vorhandenen Code ausprobieren. Der nächste Schritt wäre, direkt in der JShell zu entwickeln, um so ein schnelleres Feedback zu bekommen. Analog zum „Test-Driven-Development" hat sich hierfür der Name „REPL-Driven-Development" etabliert. Als Erstes wollen

```
jshell> 3+5;
$1 ==> 8
```

Listing 1

```
jshell> $1 * 2
$2 ==> 16
```

Listing 2

```
jshell> "Hallo Java aktuell".substring(6,10)
$4 ==> "Java"
```

Listing 3

```
jshell> "Hallo Java aktuell".sub
subSequence(   substring(
```

Listing 4

```
Signatures:
String String.substring(int beginIndex)
String String.substring(int beginIndex, int endIndex)
```

Listing 5

```
jshell>/env -class-path /home/thorsten/Downloads/com-
mons-lang3-3.6/commons-lang3-3.6.jar
|  Setting new options and restoring state.
```

Listing 6

```
jshell> public static String helloWorld() {
   ...> return "Hello World"
   ...> }
|  Error:
|  ';' expected
|  return "Hello World"
|
```

Listing 7

```
jshell> public static String helloWorld() {
   ...> return "Hello World";
   ...> }
|  Warning:
|  Modifier 'static'  not permitted in top-level
   declarations, ignored
|  public static String helloWorld() {
|  ^-----------^
|  created method helloWorld()
```

Listing 8

wir eine simple Methode schreiben, die uns „Hello World" zurückgibt. Also tippen wir „jshell> public static String helloWorld() { ...>" ein. Drücken wir auf Enter und bevor unsere Methode fertig ist, fordert ein freundlicher Pfeil uns auf, die Methode doch zu Ende zu schreiben *(siehe Listing 7)*. Schade, sobald wir Methoden schreiben, müssen wir die Semikolons doch setzen, also ein zweiter Anlauf *(siehe Listing 8)*.

Immerhin wurde unsere Methode erstellt, aber anscheinend kann man keine statischen Methoden erstellen? Da wir uns in der REPL befinden, können wir das direkt ausprobieren *(siehe Listing 9)*. Die Methode kann ohne Erstellen eines Objekts ausgeführt werden, also ist sie statisch, nur dass das Keyword „static" nicht erlaubt ist. Dementsprechend würde auch ein „this" in der Methoden-Deklaration zu einem Fehler führen. Wir können allerdings Variablen in der REPL definieren und sie in den Methoden verwenden *(siehe Listing 10)*.

Es lassen sich auch die anonymen Variablen verwenden, wobei sich hier natürlich die Frage der Sinnhaftigkeit stellt. Will man Änderungen an der Methode vornehmen, kann man sie einfach überschreiben *(siehe Listing 11)*.

Es ist mühsam, die Methode jedes Mal komplett einzugeben, da wäre es schön, wenn es einen einfacheren Weg gäbe, den Code zu verändern. Der Befehl „/edit" öffnet den eingebauten Editor, wir ändern die Methode und speichern sie mit einem Klick auf „Exit". Falls einem der eingebaute Editor nicht gefällt, lässt sich mit dem „/set"-Befehl auch ein beliebiger anderer Editor setzen. Stattdessen kann man aber auch jeden anderen Bash-Befehl schreiben *(siehe Listing 12)*.

Ist der Editor noch nicht gestartet, blockiert die JShell, bis der Editor beendet ist, im anderen Fall kann man direkt in der JShell weiterarbeiten. Dann werden allerdings die Änderungen nicht in die JShell übernommen – schade.

Diese REPL hat Klasse

Java wäre nicht Java, wenn man nicht auch ein paar schöne Klassen schreiben kann. Und natürlich geht das auch in der JShell *(siehe Listing 13)*. Wollen wir nun etwas an der Klasse ändern, gehen wir genauso vor wie bei einer Methode. Entweder wir geben die Klasse komplett neu ein oder wir starten den eingestellten Editor.

Wenn wir uns jetzt eine Übersicht darüber verschaffen wollen, was wir bisher an Code in die JShell eingegeben haben, helfen uns einige eingebaute Befehle. Mit „/list" sehen wir ähnlich wie bei „/history" unsere bisherigen Eingaben, allerdings wird nur der eingegebene Code angezeigt, Befehle werden herausgefiltert. Der Befehl „/methods" zeigt alle definierten Methoden an, „/vars" alle Variablen und „/types" alle Typen, also etwa Klassen, Interfaces oder Enums, „/imports" zeigt alle importierten Klassen an. Mit „/save" und „/open" können wir unsere Code-Snippets speichern und später wieder laden.

Da wir diese aber jedes Mal neu starten müssen, werden unsere Feedbackzyklen wieder so lang, dass sich der Vorteil von RDD ins nichts auflöst. Was also tun? Wir brauchen passende Tools und Integration in unsere Entwicklungsumgebung, damit wir die JShell effizient in unseren Entwicklungsprozess einbauen können. Zum Glück liefert die JShell alles mit, um die Integration zu ermöglichen.

```
jshell> helloWorld()
$3 ==> "Hello World"
```

Listing 9

```
jshell> int a = 42;
a ==> 42

jshell> public int getA() {
   ...> return a;
   ...> }
|  modified method getA()

jshell> getA()
$10 ==> 42

jshell> a = 23
a ==> 23
jshell> getA()
$12 ==> 23
```

Listing 10

```
jshell> public int getA() {
   ...> return a * 2;
   ...> }
|  modified method getA()

jshell> getA()
$14 ==> 46
```

Listing 11

```
jshell> /set editor /opt/idea/bin/idea.sh
|  Editor set to: /opt/idea/bin/idea.sh
```

Listing 12

Tool-Integration

Die JShell bietet ein API an, mit dem sie sich von außen komplett steuern lässt. Die Dokumentation dazu findet man unter *„http://cr.openjdk.java.net/~rfield/arch/doc/jdk/jshell/JShell.html"*. Somit kann jeder ein Programm oder ein Plug-in schreiben, das eine oder mehrere Instanzen der JShell aufruft. Eine Integration direkt in die IDE bietet sich hier an, daher ein kurzer Blick auf den Stand der Entwicklung bei den drei großen IDEs:

- **Eclipse**
  Eine direkte Integration in Eclipse wird es vermutlich nicht geben, allerdings wäre ein Plug-in für Eclipse möglich. Bisher gibt es allerdings noch keins, daher liebe Eclipse-User: Geht voran und integriert die JShell in Eclipse.
- **IntelliJ**
  Laut dem Issue-Tracker von JetBrains ist zumindest die grundlegende Integration abgeschlossen. Mit welcher Version das Feature veröffentlicht wird, ist noch nicht bekannt, es wird aber nicht vor der Version 2017.3 sein.

- **NetBeans**
  In den Nightly Builds kann man schon einen Ausblick auf die JShell-Integration in NetBeans werfen. Die JShell ist nun in die IDE integriert, der Code kann einfach bearbeitet werden – und wenn uns der Code aus der REPL gefällt, kann er per Knopfdruck in eine neue Klasse exportiert werden. Schade: Es wird wohl eine veraltete Version der JShell integriert, die Befehle weichen von der aktuellen JShell ab. Insgesamt ist NetBeans aber am weitesten mit der JShell Integration, auch wenn noch Raum für mehr Features ist (ausgewählten Code in der REPL laufen lassen, ausgewählten Code extrahieren, einzelne Methoden in vorhandene Klassen extrahieren etc.).

Bis die Integration der JShell in der Lieblings-IDE angekommen ist, kann man immerhin ein Terminal-Fenster in seiner IDE öffnen und dort die JShell laden. Man hat zwar keine Kommunikation zwischen IDE und JShell, aber immerhin alles in einem Fenster.

Fazit

Mit der JShell wandert endlich eine vollwertige, offizielle REPL in das Repertoire von Java-Entwicklern. Auch wenn es noch kleinere Schwächen gibt, etwa fehlende Package-Autovervollständigung für Nicht-Standardklassen, bietet einem die JShell alle wichtigen Features, die wir bei einer REPL sehen wollen. Ob sich die REPL im Entwickleralltag durchsetzen wird, hängt nun von der Tool-Unterstützung, insbesondere durch die IDEs, ab. Die nächsten Monate werden zeigen, wie die Entwickler der IDEs die Unterstützung der JShell mit neuen Features vorantreiben.

Hinweis: Listing 13 finden Sie unter *https://www.doag.org/de/home/news/java-aktuell-ausgabe-52017-jetzt-online/detail/*

Thorsten Ludwig
tludwig@inovex.de

Thorsten Ludwig arbeitet als Software-Entwickler bei der inovex GmbH. Er ist spezialisiert auf die Entwicklung komplexer Software-Systeme auf der JVM und freut sich über jedes neue Feature, das Java ein klein wenig funktionaler macht.