JavaScript für Java-Developer – brauchen wir das wirklich?

Robert Rohm, Aeonium Software Systems

JavaScript ist „The World's Most Misunderstood Programming Language" [1] – und ein fester Bestandteil der Java-Laufzeit-Umgebung. Die JVM bringt die JavaScript-Engine Nashorn mit. In JavaFX besitzt die WebView-Komponente als Teil des eingebetteten WebKit-Browsers ebenfalls einen JavaScript-Interpreter, der auch aus Java heraus genutzt werden kann. Nur: Wofür soll das gut sein?

Mögliche Antworten auf diese Frage lassen sich auf zwei Wegen finden. Zum einen hilft das „Big Picture", wenn man sich vergegenwärtigt, wo JavaScript im Umfeld von Java-Anwendungen auftaucht *(siehe Abbildung 1)*. Wichtig ist, dass nicht nur der JavaScript-Interpreter, sondern auch die JavaScript-APIs mit ihm zusammen die JS-Laufzeitumgebung ausmachen. Im Web-Browser sind dies das DOM-API und die HTML5-JS-APIs. Standalone-Umgebungen wie „node.js" haben ihr eigenes API – und in der JVM kann Nashorn auf die Java-APIs zugreifen und die Webkit-Engine auf das dargestellte HTML. Andererseits gibt es gerade im Zusammenspiel mit der JVM und der Verwendung von Nashorn einiges zu beachten. Deshalb werden zunächst grundlegende Themen rund um die Nashorn-Engine näher beleuchtet.

JavaScript in der JVM

Nashorn, die JavaScript-Engine der JVM, kann auf zwei Wegen genutzt werden: entweder mit dem Konsolen-Tool „jjs" *[2]* oder über das Java-Scripting-API *[3]* in der Java-Anwendung. Mit „jjs" wird die Nashorn-Engine an der Kommandozeile aufgerufen; man kann entweder eine Skript-Datei angeben oder Nashorn interaktiv verwenden.

Als JavaScript-Ausführungsumgebung hat Nashorn seine eigenen Laufzeit-Bibliotheken, das sind neben dem Scripting-API alle Bibliotheken des JRE. Wer bereits mit JavaScript vertraut ist, kann also jede Java-Klasse in JavaScript nutzen. *Listing 1* zeigt dies am Beispiel einer Dateiverarbeitung mit dem NIO2- und Collections-API. Es geht von dem fiktiven Szenario aus, dass innerhalb großer Text-Dateien

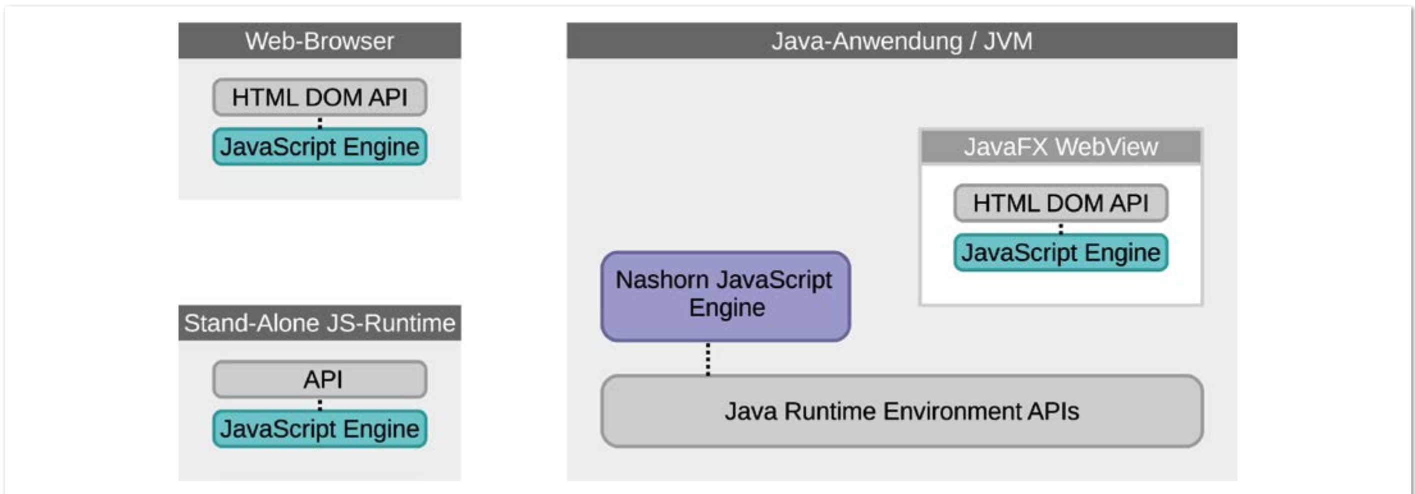

Abbildung 1: JavaScript im Umfeld von Java-Anwendungen

die maximale Zeilenlänge ermittelt werden soll – stellvertretend für anspruchsvollere Anforderungen. Man sieht neben dem Aufruf der Java-Methoden in JavaScript auch, wie Lambda-Ausdrücke in JavaScript abgebildet werden: mit Funktionsausdrücken beziehungsweise Closures.

Erfahrenere JavaScript-User werden einwenden, dass JavaScript ein eigenes Array-API *[4]* besitzt, das für diese Zwecke völlig ausreichend ist. Auch das ist kein Problem: Nashorn stellt in seiner Ausführungsumgebung das „Java"-Objekt bereit, das unter anderem Konvertierungsfunktionen besitzt, mit denen Java-Arrays in JavaScript-Arrays umgewandelt werden können – und umgekehrt. In *Listing 2* wird das String-Array mit „Java.from()" in ein JavaScript-Array umgewandelt, auf dem dann auch mit dem JavaScript-Array-API gearbeitet werden kann.

```
var path = java.nio.file.Paths.get("/data/daten.csv");
var lines = java.nio.file.Files.readAllLines(path);

var maxLength = lines.stream().mapToInt(function(t)
{return t.length()}).max().getAsInt();
print("Max. Zeilenlänge: " + maxLength);
```

Listing 1: Java-Klassen in JavaScript verwenden

```
var path = java.nio.file.Paths.get("/data/daten.csv");
var lines = java.nio.file.Files.readAllLines(path);

// Umwandlung in ein JavaScript-Array
var linesArray = Java.from(lines);

// Verarbeitung mit der JavaScript-Array-API
var maxLength = linesArray.reduce(
  function(accumulator, value){
     return Math.max(accumulator || 0, value.length);
  }
);
print("Max. Zeilenlänge: " + maxLength);
```

Listing 2: Java-Arrays in JavaScript-Arrays konvertieren und verarbeiten

JavaFX auf Nashorn

Mit Nashorn lassen sich nicht nur sogenannte „Headless-Anwendungen" ohne GUI erstellen; auch Anwendungen mit Swing- oder JavaFX-Benutzeroberfläche können dank Nashorn in JavaScript geskriptet werden. Im Fall von JavaFX ist jedoch zu berücksichtigen, dass das JavaFX-Framework seinen eigenen Initialisierungszyklus besitzt. Die Start-Klasse einer JavaFX-Anwendung muss von „javafx.application.Application" ableiten und hat zwingend mehr oder weniger diese Grundstruktur *(siehe Listing 3)*.

Die Ableitung neuer Typen von Java-Klassen ist in Nashorn zwar möglich *[5]*, das würde jedoch nicht das Problem der Initialisierung lösen. Nashorn geht einen anderen Weg: Die Nashorn-Engine muss mit „jjs -fx" gestartet und der JavaScript-Code als Datei übergeben werden. Interaktives JavaFX-Scripting von der Konsole weg ist leider nicht möglich. Dieses Problem lässt sich umgehen, wenn der Script-Code dynamisch in einer bereits laufenden JavaJX-Anwendung ausgewertet wird.

Listing 4 zeigt, wie die Initialisierung einer JavaFX-Anwendung in JavaScript aussehen kann. Das Listing greift das Beispiel-Szenario wieder auf und visualisiert die Auswertung in einem JavaFX-Pie-Chart. Die Funktion „start(primaryStage)" ist übrigens verpflichtend. Sie ist auch in JavaScript mit Nashorn der Einsprungpunkt für JavaFX-Anwendungen.

Die Umsetzung in JavaScript bringt gegenüber der Implementierung in Java einige Vereinfachungen; so dürfen Variablen, die in anonymen Funktionen oder Closures verwendet werden, durchaus erst unterhalb der Funktion deklariert sein. Auch die Notwendigkeit, dass solche Variablen „final" oder zumindest „effectively final" sein müssen, gibt es in JavaScript nicht.

Besseres Import-Management

Im *Listing 4* fällt die umständliche Deklaration der Variablen für die Java-Typen ins Auge. Für ein bequemeres und besseres Management der zu importierenden Klassen bietet Nashorn zwei Mechanismen: Einfacher geht es mit den Funktionen „importPackage" und „importClass" aus dem Nashorn-API *(siehe Listing 5)*.

Die Anweisung „importPackage(java.lang)" birgt ein Problem: Sie importiert Klassen als Konstruktor-Objekte in den globalen Scope der JavaScript-Engine. Darunter sind Standard-Typen wie „java.lang.Object", „java.lang.Boolean" etc. In JavaScript existieren eingebaute

Standard-Objekte mit dem Namen „Object", „Boolean", „Number", „Array" etc. Um Namenskollisionen zu vermeiden, wird statt der Import-Funktionen die Verwendung des „JavaImporter" empfohlen. Mit ihm kann die Verwendung von Java-Typen auf einen definierten engeren Scope beschränkt werden. *Listing 6* zeigt, worauf beim Einsatz eines Importer in komplexeren Beispielen zu achten ist. So könnte die JavaFX-Anwendung mit „JavaImporter" aussehen. Man sieht: Die „with(importer)"-Anweisung ist gegebenenfalls in inneren Funktionen zu wiederholen. Es reicht nicht, einen Scope mit Importer in der äußeren Funktion zu definieren.

```
import javafx.application.Application;
// …
public class MeineJavaFXApp extends Application {

  @Override
  public void start(Stage primaryStage) {
     // Aufbau des Fenster-Inhalts …
     VBox root = new VBox();
     root.getChildren().addAll( /* Fenster-Inhalt hier einfügen … */ );
     primaryStage.setScene(new Scene(root, 300, 250));
     primaryStage.show();
  }
  public static void main(String[] args) {
     Application.launch(args);
  }
}
```

Listing 3: Grundstruktur einer JavaFX-Anwendung

```
var Button = javafx.scene.control.Button;
var VBox = javafx.scene.layout.VBox;
var Scene = javafx.scene.Scene;
var PieChart = javafx.scene.chart.PieChart;
var PieChartData = javafx.scene.chart.PieChart.Data;
var FXCollections = javafx.collections.FXCollections;
var ObservableList = javafx.collections.ObservableList;

function start(primaryStage) {
  primaryStage.title = "Hello PieChart!";
  var button = new Button();
  button.text = "Datei laden ...";
  button.onAction = function () {
  var pieChartData = FXCollections.observableArrayList();
     var path = java.nio.file.Paths.get("/data/daten.csv");
     var lines = java.nio.file.Files.readAllLines(path);
     var n = 1;
     lines.stream().forEach(function(line){
        pieChartData.add(new PieChartData("Zeile " + n + ":\n" + line.length(), line.length()));
        n++;
     });
     chart.setData(pieChartData);
  };

  var root = new VBox();
  var chart = new PieChart();
  root.children.addAll(button, chart);
  primaryStage.scene = new Scene(root, 300, 250);
  primaryStage.show();
}
```

Listing 4: Mit Nashorn geskriptete Datenauswertung und -Visualisierung

```
// Dieses Skript muss zuerst geladen werden. Es stellt die Import-Funktionen bereit:
load("nashorn:mozilla_compat.js“);
// Import für Klassen und Packages
importClass(java.nio.file.Files);
importClass(java.nio.file.Paths);
importPackage(java.lang); // ACHTUNG: Hier gibt es Namenskollisionen!

var path = Paths.get("/data/daten.csv");
// ...
```

Listing 5: Vereinfachter Import für Klassen und Packages

```
var importer = JavaImporter(
            java.nio.file,
            javafx.scene.control,
            javafx.scene.layout,
            javafx.scene.chart.PieChart,
            javafx.scene.chart.PieChart.Data,
            javafx.scene.Scene,
            javafx.collections
         );

function start(primaryStage) {
  with (importer) { // Wichtig!

      primaryStage.title = "Hello PieChart!";
      var button = new Button();
      button.text = "Datei laden ...";
      button.onAction = function () {
         with (importer) { // Wichtig!
            var pieChartData = FXCollections.observableArrayList();
            var path = Paths.get(„/data/daten.csv“);
            var lines = Files.readAllLines(path);
            var n = 1;
            lines.stream().forEach(function (line) {
               pieChartData.add(new PieChart.Data("Zeile " + n + ":\n" + line.length(), line.length()));
               n++;
            });
            chart.setData(pieChartData);
         }
      };

      var root = new VBox();
      var chart = new PieChart();
      root.children.addAll(button, chart);
      primaryStage.scene = new Scene(root, 300, 250);
      primaryStage.show();
  }
}
```

Listing 6: Import mit „JavaImporter", begrenzt auf engere Scopes

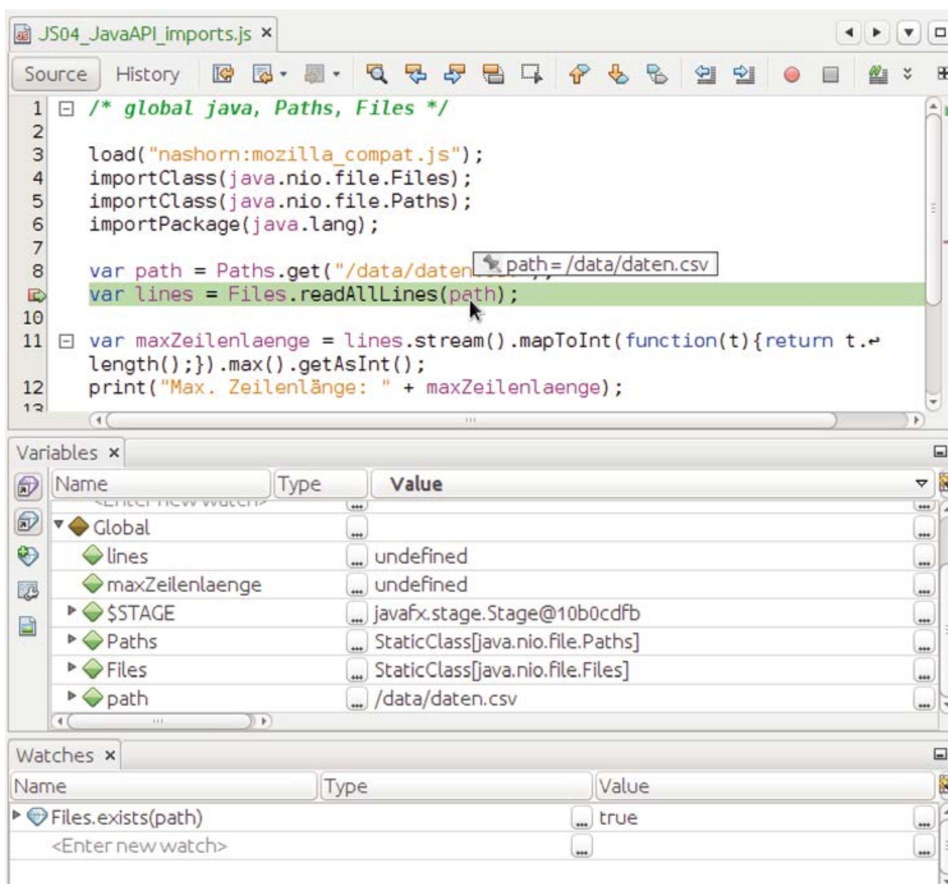

Abbildung 2: Debugging von JavaScript-Code mit NetBeans

Debugging mit Nashorn? Kein Problem

Die Fehlersuche mit klassischem Debugging funktioniert mit geeigneten IDEs wie NetBeans oder IntelliJ IDEA im Wesentlichen so wie in Java-Anwendungen. So kann bei Ausführung des JavaScript-Codes aus der IDE heraus auch wie gewohnt mit Break-Points und Debugging-Tools gearbeitet werden. *Abbildung 2* zeigt dies am Beispiel von NetBeans – hier hat man am Haltepunkt unter anderem Zugriff auf lokale Variablen, den globalen Scope, den Call Stack und auf überwachbare Ausdrücke.

Für tiefere Einblicke in die Arbeit von Nashorn ist es übrigens gut zu wissen, dass der Nashorn-Launcher „jjs" mit der Option „-help" zwar die wichtigsten (offiziellen) Kommandozeilen-Optionen beschreibt; der undokumentierten Aufruf „jjs -xhelp" bringt dagegen die volle Palette zum Vorschein. So findet man hier nicht nur einige Optionen für die Erzeugung von Debug-Informationen, son-

dern auch für die Ausgabe des erzeugten Java-Bytecodes an der Konsole *(siehe Listing 7)*.

Das Java-Scripting-API

Bis hierher war nur die Ausführung von JavaScript in der Nashorn-Umgebung das Thema. Mit dem Java-Scripting-API kann die Nashorn-Engine dagegen auch aus Java-Anwendungen heraus genutzt werden. Dies erlaubt nicht nur die Integration von JavaScript-Code in Java-Anwendungen, sondern eröffnet auch zusätzliche Möglichkeiten. *Listing 8* zeigt zunächst den ersten Schritt, das Laden des Codes. Das Beispiel lässt sich gut mit dem Code von *Listing 2* in einer Datei „JS_Beispielcode.js“ ausführen. Gibt man in dem Beispiel die verfügbaren JavaScript-Engines der JVM aus, fällt übrigens auf, dass der Name „Oracle Nashorn“ lautet. Auch der Name „nashorn“ kann für den „getEngineByName()“-Aufruf verwendet werden.

Für das Zusammenspiel von Java und JavaScript bietet das Java-Scripting-API drei wichtige Möglichkeiten: Zum einen können Java-Objekte und -Variablen in den Scope der ScriptEngine eingefügt werden. Zum anderen gibt der „engine.eval()“-Aufruf gegebenenfalls Objekte und Variablen aus dem Scope der ScriptEngine heraus. Außerdem lassen sich aus Java heraus Funktionen im Scope der ScriptEngine aufrufen. Dies erlaubt eine enge Verzahnung von Java- und JavaScript-Logik. Das Beispiel in *Listing 9* zeigt diese Möglichkeiten anhand einer fiktiven Business-Logik-Klasse „JavaLogic“. Diese kann mit der Methode „engine.put()“ in den JavaScript-Scope eingefügt und an einen Variablennamen gebunden werden.

Die „engine.get(…)“-Methode liest übrigens auch Variablen aus dem Engine-Scope heraus. Zudem ist in der Beispiel-Anwendung JavaScript-Code definiert, der Methoden auf der „JavaLogic“-Instanz aufruft. Um eine JavaScript-Funktion aus Java heraus aufrufen zu können, muss die Engine zunächst in eine aufrufbare „Invocable“-Engine gecastet werden. Hat der Rückgabe-Wert einen primitiven Typ, wird er in einen entsprechenden Java-Typ umgewandelt. Das gilt etwa für Strings. Das Beispiel zeigt jedoch, wie mit komplexen Rückgabe-Werten gearbeitet werden kann. JavaScript-Objekte lassen sich als Instanz von „jdk.nashorn.api.scripting.JSObject“ ansprechen. Über dieses Interface ist auch der Zugriff auf die Eigenschaften und Methoden des Objekts möglich.

In der „JavaFX-WebView“ gelten andere Rahmenbedingungen: Intern wird die WebKit Browser Engine *[6]* verwendet, die unter anderem auch in Apples Browser Safari zum Einsatz kommt. Das bedeutet für JavaFX gutes HTML-Rendering – aber auch einen eigenen, zweiten JavaScript-Interpreter. Aus verschiedenen Gründen konnte WebKits-JavaScript-Interpreter JavaScriptCore nicht ohne Weiteres gegen Nashorn ersetzt werden *[7]*. Man hätte damit rechnen müssen, gegebenenfalls einen eigenen Fork von WebKit pflegen zu müssen. Zudem implementiert die Klasse „javafx.scene.web.WebEngine“ nicht dasselbe Interface „ScriptEngine“.

Die WebView-Komponente eignet sich jedoch hervorragend zum Einbetten von HTML- und JavaScript-Inhalten. So können auch komplexere JavaScript-Anwendungen wie der Sourcecode-Editor „Ace“ *[8]* oder Google Maps in Java-Anwendungen eingebettet werden.

Fazit

Mit den hier umrissenen Möglichkeiten kann JavaScript in der JVM zum einen sehr leicht sämtliche JRE-Bibliotheken für Scripting-

BROCKHAUS AG

SOFTWAREENTWICKLER - JAVA (M/W)

FESTANSTELLUNG, IN VOLLZEIT

IT-Beratung und Softwareentwicklung sind unsere Leidenschaft. Mit gut 30 Jahren Erfahrung sind wir als mittelständisches Unternehmen erfolgreich unterwegs und suchen zur Verstärkung unseres Teams einen JAVA-Entwickler.

IHRE HERAUSFORDERUNG

- Zu Ihren Hauptaufgaben gehört die Konzeption und Entwicklung von kundenspezifischen Softwarelösungen mit JAVA in einer JEE-Architektur, unter Verwendung von Technologien wie Spring, Hibernte, und objektorientierter Methoden (OOA, OOD, UML, OOP).
- Sie unterstützen das Team bei der Entwicklung von komplexen Schnittstellen, bei der Integration neuer Komponenten in bestehende Architekturen sowie bei der Weiterentwicklung der bestehenden Software.
- Die Integration neuer Komponenten in bestehende Architekturen sowie in unterschiedlichsten IT-Umgebungen
- Sie arbeiten in einem agilen Team und verwenden Entwicklungswerkzeuge und Laufzeitumgebungen, wie z. B. Eclipse, Maven, Apache Tomcat.

WANTED: INNOVATIVER TEAMPLAYER

BROCKHAUS AG | Pierbusch 17 | 44536 Lünen | +49 231 98 75 - 702
bewerbung@brockhaus-ag.de | www.brockhaus-ag.de/career

```
  --debug-lines (Generate line number table in .class files.)
     param: [true|false] default: true

  --debug-locals (Generate local variable table in .class files.)
     param: [true|false] default: false

  --debug-scopes (Put all variables in scopes to make them debuggable.)
     param: [true|false] default: false

  -doe, -dump-on-error (Dump a stack trace on errors.)
     param: [true|false] default: false
(…)
  -pc, --print-code (Print generated bytecode. If a directory is specified, nothing will
           be dumped to stderr. Also, in that case, .dot files will be generated
           for all functions or for the function with the specified name only.)
     param: [dir:<output-dir>,function:<name>]
(…)
  --print-mem-usage (Print memory usage of IR after each compile stage.)
     param: [true|false] default: false
```

Listing 7: Der Aufruf „jjs -xhelp" zeigt eine ganze Reihe interessanter Optionen

```
// …
public class JavaJSAnwendung1 {

  public static void main(String[] args) throws ScriptException, FileNotFoundException {

     // Ausgangspunkt: Der ScriptEngineManager
     ScriptEngineManager scriptEngineManager = new ScriptEngineManager();

     // Zur Information: Ausgabe der Verfügbaren JS-Engines
     scriptEngineManager.getEngineFactories()
        .stream()
        .forEach((t) -> System.out.println(t.getEngineName()));

     ScriptEngine engine = scriptEngineManager.getEngineByName("JavaScript");
     // Direkte Übergabe von JavaScript-Code als String:
     engine.eval("print(\"Let‘s go!\");");
     // Oder besser: Einlesen des JavaScript-Codes aus einer Datei:
     engine.eval(new FileReader("/pfad/zu/JS_Beispielcode.js"));
  }
}
```

Listing 8: Ausführung von JavaScript mit Nashorn in einer Java-Anwendung

```
public class JavaJSAnwendung2 {

  /**
     * Java-Klasse, z.B. Business-Logik
     */
  public static class JavaLogic {

     public String doBusiness(String input) {
        return input + "done.";
     }
  }

  public static void main(String[] args) throws ScriptException, NoSuchMethodException {

     ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
     ScriptEngine engine = scriptEngineManager.getEngineByName("JavaScript");

     // Einfügen einer Instanz in den JavaScript-Scope
     engine.put("javalogic", new JavaLogic());
     // JavaScript-Logik, ruft Methode auf Java-Objekt auf und gibt ein Ergebnis-Objekt zurück
     engine.eval("function doWork(input) { "
           + " return {"
           + "          value: javalogic.doBusiness(input),"
           + "          status: \"OK\" "
           + "        };"
           + "}; ");

     // Aufruf einer JS-Funktion aus Java heraus:
     Invocable invocableEngine = (Invocable) engine;

     // Auswertung der Rückgabe als jdk.nashorn.api.scripting.JSObject:
     JSObject result = (JSObject) invocableEngine.invokeFunction("doWork", "lot of work ");
     System.out.println(result.getMember("value"));
  }
}
```

Listing 9: Gegenseitiger Aufruf von Java und JavaScript

Aufgabe nutzbar machen. Zum anderen bietet JavaScript als dynamische Sprache auch die Chance, dynamisches Verhalten innerhalb einer Java-Anwendung zu realisieren – auch wenn man neben den Chancen die Risiken sorgfältig diskutieren muss.

Neben der direkten Nutzung von JavaScript in der JVM mit Nashorn gibt es übrigens noch eine ganze Reihe weiterer interessanter Möglichkeiten, wie das Zusammenspiel von Java und JavaScript ein schlüssiges Gesamtbild ergibt – oder aber für bestimmte Architektur-Stile. Man denke nur an das WebSockets-API für die Kommunikation von JEE-Anwendungen und JavaScript-Clients, an AJAX- und REST-Kommunikation oder an Thin-Server-Architecture. Es ergibt also durchaus Sinn, sich als Java-Entwickler auch mit JavaScript zu beschäftigen.

Quellen

[1] http://crockford.com/javascript/javascript.html
[2] Oracle JDK 9 Documentation: *https://docs.oracle.com/javase/9/tools/jjs.htm*
[3] Oracle Java Platform, Standard Edition Java Scripting Programmer's Guide: *https://docs.oracle.com/javase/9/scripting/java-scripting-api.htm*
[4] Mozilla Developer Network, JavaScript Reference: *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array*
[5] Oracle Java Platform, Standard Edition Java Scripting Programmer's Guide: *https://docs.oracle.com/javase/9/scripting/using-java-scripts.htm*
[6] *https://webkit.org/*
[7] E-Mail von Richard Bair vom 9. Mai2013: *http://mail.openjdk.java.net/pipermail/openjfx-dev/2013-May/007597.html*
[8] *https://ace.c9.io*

Robert Rohm
info@aeonium-systems.de

Robert Rohm arbeitet als selbständiger Entwickler, Berater und Trainer in Nürnberg. Mit einem starken Fokus auf der Architektur und Full-Stack-Entwicklung von mehrschichtigen Anwendungen steht er mit einem Bein im Frontend und mit dem anderen Bein im Backend. Als Software-Ingenieur (M. Eng.) und Lehrer ist ihm die Wissensarbeit und Wissensvermittlung nicht nur ein berufliches Anliegen, sondern auch eine persönliche Leidenschaft.

JASMIN
IST
EXPERTIN
FÜR
SKETCHING
PRODUKTDESIGN UND
DIGITALE
TRANSFORMATION

BE YOURSELF.
MAKE A DIFFERENCE.
Jetzt bewerben auf **accenture.com/MakeADifference**
#MakeADifference