W

Java 8 Nashorn Tutorial

April 05, 2014

Learn all about the Nashorn Javascript Engine with easily understood code examples. The Nashorn Javascript Engine is part of Java SE 8 and competes with other standalone engines like Google V8 (the engine that powers Google Chrome and Node.js). Nashorn extends Javas capabilities by running dynamic javascript code natively on the JVM.

In the next ~15 minutes you learn how to evaluate javascript on the JVM dynamically during runtime. The most recent Nashorn language features are demonstrated with small code examples. You learn how to call javascript functions from java code and vice versa. At the end you're ready to integrate dynamic scripts in your daily java business.

UPDATE *- I'm currently working on a JavaScript implementation of the Java 8 Streams API for the browser. If I've drawn your interest check out Stream.js on GitHub. Your Feedback is highly appreciated.*

Using Nashorn

The Nashorn javascript engine can either be used programmatically from java programs or by utilizing the command line tool `jjs`, which is located in `$JAVA_HOME/bin`. If you plan to work with `jjs` you might want to put a symbolic link for simple access:

```
$ cd /usr/bin
$ ln -s $JAVA_HOME/bin/jjs jjs
$ jjs
jjs> print('Hello World');
```

This tutorial focuses on using nashorn from java code, so let's skip `jjs` for now. A simple HelloWorld in java code looks like this:

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval("print('Hello World!');");
```

In order to evaluate javascript code from java, you first create a nashorn script engine by utilizing the `javax.script` package already known from Rhino (Javas legacy js engine from Mozilla).

Javascript code can either be evaluated directly by passing javascript code as a string as shown above. Or you can pass a file reader pointing to your .js script file:

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval(new FileReader("script.js"));
```

Nashorn javascript is based on ECMAScript 5.1 but future versions of nashorn will include support for ECMAScript 6:

> *The current strategy for Nashorn is to follow the ECMAScript specification. When we release with JDK 8 we will be aligned with ECMAScript 5.1. The follow up major release of Nashorn will align with ECMAScript Edition 6.*

Nashorn defines a lot of language and API extensions to the ECMAScript standard. But first let's take a look at how the communication between java and javascript code works.

Invoking Javascript Functions from Java

Nashorn supports the invocation of javascript functions defined in your script files directly from java code. You can pass java objects as function arguments and return data back from the function to the calling java method.

The following javascript functions will later be called from the java side:

```
var fun1 = function(name) {
    print('Hi there from Javascript, ' + name);
    return "greetings from javascript";
};

var fun2 = function (object) {
    print("JS Class Definition: " + Object.prototype.toString.call(object));
};
```

In order to call a function you first have to cast the script engine to `Invocable`. The Invocable interface is implemented by the `NashornScriptEngine` implementation and defines a method `invokeFunction` to call a javascript function for a given name.

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval(new FileReader("script.js"));

Invocable invocable = (Invocable) engine;

Object result = invocable.invokeFunction("fun1", "Peter Parker");
System.out.println(result);
System.out.println(result.getClass());

// Hi there from Javascript, Peter Parker
// greetings from javascript
// class java.lang.String
```

Executing the code results in three lines written to the console. Calling the function `print` pipes the result to `System.out`, so we see the javascript message first.

Now let's call the second function by passing arbitrary java objects:

```
invocable.invokeFunction("fun2", new Date());
// [object java.util.Date]

invocable.invokeFunction("fun2", LocalDateTime.now());
// [object java.time.LocalDateTime]
```

```
invocable.invokeFunction("fun2", new Person());
// [object com.winterbe.java8.Person]
```

Java objects can be passed without loosing any type information on the javascript side. Since the script runs natively on the JVM we can utilize the full power of the Java API or external libraries on nashorn.

Invoking Java Methods from Javascript

Invoking java methods from javascript is quite easy. We first define a static java method:

```
static String fun1(String name) {
    System.out.format("Hi there from Java, %s", name);
    return "greetings from java";
}
```

Java classes can be referenced from javascript via the `Java.type` API extension. It's similar to importing classes in java code. As soon as the java type is defined we naturally call the static method `fun1()` and print the result to `sout`. Since the method is static, we don't have to create an instance first.

```
var MyJavaClass = Java.type('my.package.MyJavaClass');

var result = MyJavaClass.fun1('John Doe');
print(result);

// Hi there from Java, John Doe
// greetings from java
```

How does Nashorn handle type conversion when calling java methods with native javascript types? Let's find out with a simple example.

The following java method simply prints the actual class type of the method parameter:

```
static void fun2(Object object) {
    System.out.println(object.getClass());
}
```

To understand how type conversations are handled under the hood, we call this method with different javascript types:

```
MyJavaClass.fun2(123);
// class java.lang.Integer

MyJavaClass.fun2(49.99);
// class java.lang.Double

MyJavaClass.fun2(true);
// class java.lang.Boolean

MyJavaClass.fun2("hi there")
// class java.lang.String

MyJavaClass.fun2(new Number(23));
// class jdk.nashorn.internal.objects.NativeNumber

MyJavaClass.fun2(new Date());
// class jdk.nashorn.internal.objects.NativeDate

MyJavaClass.fun2(new RegExp());
// class jdk.nashorn.internal.objects.NativeRegExp

MyJavaClass.fun2({foo: 'bar'});
// class jdk.nashorn.internal.scripts.JO4
```

Primitive javascript types are converted to the appropriate java wrapper class. Instead native javascript objects are represented by internal adapter classes. Please keep in mind that classes from `jdk.nashorn.internal` are subject to change, so you shouldn't program against those classes in client-code:

> *Anything marked internal will likely change out from underneath you.*

ScriptObjectMirror

When passing native javascript objects to java you can utilize the class `ScriptObjectMirror` which is actually a java representation of the underlying javascript object. ScriptObjectMirror implements the map interface and resides inside the package `jdk.nashorn.api`. Classes from this package are intended to be used in client-code.

The next sample changes the parameter type from `Object` to `ScriptObjectMirror` so we can extract some infos from the passed javascript object:

```
static void fun3(ScriptObjectMirror mirror) {
    System.out.println(mirror.getClassName() + ": " +
        Arrays.toString(mirror.getOwnKeys(true)));
}
```

When passing an object hash to this method, the properties are accessible on the java side:

```
MyJavaClass.fun3({
    foo: 'bar',
    bar: 'foo'
});

// Object: [foo, bar]
```

We can also call member functions on javascript object from java. Let's first define a javascript type Person with properties `firstName` and `lastName` and method `getFullName`.

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.getFullName = function() {
        return this.firstName + " " + this.lastName;
    }
}
```

The javascript method `getFullName` can be called on the ScriptObjectMirror via `callMember()`.

```
static void fun4(ScriptObjectMirror person) {
    System.out.println("Full Name is: " + person.callMember("getFullName"));
}
```

When passing a new person to the java method, we see the desired result on the console:

```
var person1 = new Person("Peter", "Parker");
MyJavaClass.fun4(person1);

// Full Name is: Peter Parker
```

Language Extensions

Nashorn defines various language and API extensions to the ECMAScript standard. Let's head right into the most recent features:

Typed Arrays

Native javascript arrays are untyped. Nashorn enables you to use typed java arrays in javascript:

```
var IntArray = Java.type("int[]");

var array = new IntArray(5);
array[0] = 5;
array[1] = 4;
array[2] = 3;
array[3] = 2;
array[4] = 1;

try {
    array[5] = 23;
} catch (e) {
    print(e.message);  // Array index out of range: 5
}

array[0] = "17";
print(array[0]);  // 17

array[0] = "wrong type";
print(array[0]);  // 0

array[0] = "17.3";
print(array[0]);  // 17
```

The `int[]` array behaves like a real java int array. But additionally Nashorn performs implicit type conversions under the hood when we're trying to add non-integer values to the array. Strings will be auto-converted to int which is quite handy.

Collections and For Each

Instead of messing around with arrays we can use any java collection. First define the java type via `Java.type`, then create new instances on demand.

```
var ArrayList = Java.type('java.util.ArrayList');
var list = new ArrayList();
list.add('a');
list.add('b');
```

```
list.add('c');

for each (var el in list) print(el);  // a, b, c
```

In order to iterate over collections and arrays Nashorn introduces the `for each` statement. It works just like the foreach loop in java.

Here's another collection foreach example, utilizing `HashMap`:

```
var map = new java.util.HashMap();
map.put('foo', 'val1');
map.put('bar', 'val2');

for each (var e in map.keySet()) print(e);  // foo, bar

for each (var e in map.values()) print(e);  // val1, val2
```

Lambda expressions and Streams

Everyone loves lambdas and streams - so does Nashorn! Although ECMAScript 5.1 lacks the compact arrow syntax from the Java 8 lambda expressions, we can use function literals where ever lambda expressions are accepted.

```
var list2 = new java.util.ArrayList();
list2.add("ddd2");
list2.add("aaa2");
list2.add("bbb1");
list2.add("aaa1");
list2.add("bbb3");
list2.add("ccc");
list2.add("bbb2");
list2.add("ddd1");

list2
    .stream()
    .filter(function(el) {
        return el.startsWith("aaa");
    })
    .sorted()
    .forEach(function(el) {
        print(el);
    });
    // aaa1, aaa2
```

Extending classes

Java types can simply be extended with the `Java.extend` extension. As you can see in the next example, you can even create multi-threaded code in your scripts:

```
var Runnable = Java.type('java.lang.Runnable');
var Printer = Java.extend(Runnable, {
    run: function() {
        print('printed from a separate thread');
    }
});

var Thread = Java.type('java.lang.Thread');
new Thread(new Printer()).start();

new Thread(function() {
    print('printed from another thread');
}).start();

// printed from a separate thread
// printed from another thread
```

Parameter overloading

Methods and functions can either be called with the point notation or with the square braces notation.

```
var System = Java.type('java.lang.System');
System.out.println(10);                 // 10
System.out["println"](11.0);            // 11.0
System.out["println(double)"](12);      // 12.0
```

Passing the optional parameter type `println(double)` when calling a method with overloaded parameters determines the exact method to be called.

Java Beans

Instead of explicitly working with getters and setters you can just use simple property names both for getting or setting values from a java bean.

```
var Date = Java.type('java.util.Date');
var date = new Date();
date.year += 1900;
print(date.year);  // 2014
```

Function Literals

For simple one line functions we can skip the curly braces:

```
function sqr(x) x * x;
print(sqr(3));    // 9
```

Binding properties

Properties from two different objects can be bound together:

```
var o1 = {};
var o2 = { foo: 'bar'};

Object.bindProperties(o1, o2);

print(o1.foo);    // bar
o1.foo = 'BAM';
print(o2.foo);    // BAM
```

Trimming strings

I like my strings trimmed.

```
print("   hehe".trimLeft());            // hehe
print("hehe    ".trimRight() + "he");   // hehehe
```

Whereis

In case you forget where you are:

```
print(__FILE__, __LINE__, __DIR__);
```

Import Scopes

Sometimes it's useful to import many java packages at once. We can use the class `JavaImporter` to be used in conjunction with the `with` statement. All class files from the imported packages are accessible within the local scope of the `with` statement:

```
var imports = new JavaImporter(java.io, java.lang);
with (imports) {
    var file = new File(__FILE__);
    System.out.println(file.getAbsolutePath());
    // /path/to/my/script.js
}
```

Convert arrays

Some packages like `java.util` can be accessed directly without utilizing `Java.type` or `JavaImporter`:

```
var list = new java.util.ArrayList();
list.add("s1");
list.add("s2");
list.add("s3");
```

This code converts the java list to a native javascript array:

```
var jsArray = Java.from(list);
print(jsArray);                                  // s1,s2,s3
print(Object.prototype.toString.call(jsArray));  // [object Array]
```

And the other way around:

```
var javaArray = Java.to([3, 5, 7, 11], "int[]");
```

Calling Super

Accessing overridden members in javascript is traditionally awkward because javas `super` keyword doesn't exist in ECMAScript. Luckily nashorn goes to the rescue.

First we define a super type in java code:

```
class SuperRunner implements Runnable {
    @Override
    public void run() {
        System.out.println("super run");
    }
}
```

Next we override `SuperRunner` from javascript. Pay attention to the extended nashorn syntax when creating a new `Runner` instance: The syntax of overriding members is borrowed from javas anonymous objects.

```
var SuperRunner = Java.type('com.winterbe.java8.SuperRunner');
var Runner = Java.extend(SuperRunner);

var runner = new Runner() {
    run: function() {
        Java.super(runner).run();
        print('on my run');
    }
}
runner.run();

// super run
// on my run
```

We call the overridden method `SuperRunner.run()` by utilizing the `Java.super` extension.

Loading scripts

Evaluating additional script files from javascript is quite easy. We can load both local or remote scripts with the `load` function.

I'm using Underscore.js a lot for my web front-ends, so let's reuse Underscore in Nashorn:

```
load('http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.6.0/underscore-min.js');

var odds = _.filter([1, 2, 3, 4, 5, 6], function (num) {
    return num % 2 == 1;
});

print(odds);  // 1, 3, 5
```

The external script will be evaluated in the same javascript context, so we can access the underscore variable directly. Keep in mind that loading scripts can potentially break your own code when variable names are overlapping each other.

This problem can be bypassed by loading script files into a new global context:

```
loadWithNewGlobal('script.js');
```

Command-line scripts

If you're interested in writing command-line (shell) scripts with Java, give Nake a try. Nake is a simplified Make for Java 8 Nashorn. You define tasks in a project-specific `Nakefile`, then run those tasks by typing `nake -- myTask` into the command line. Tasks are written in javascript and run in Nashorns scripting mode, so you can utilize the full power of your terminal as well as the JDK8 API and any java library.

For Java Developers writing command-line scripts is easy as never before...

That's it

I hope this guide was helpful to you and you enjoyed our journey to the Nashorn Javascript Engine. For further information about Nashorn read here, here and here. A guide to coding shell scripts with Nashorn can be found here.

I recently published a follow up article about how to use Backbone.js models with the Nashorn Javascript Engine. If you want to learn more about Java 8 feel free to read my Java 8 Tutorial and my Java 8 Stream Tutorial.

The runnable source code from this Nashorn tutorial is hosted on GitHub. Feel free to fork the repository or send me your feedback via Twitter.

Keep on coding!

Benjamin is Software Engineer, Full Stack Developer at Pondus, an excited runner and table foosball player. Get in touch on Twitter and GitHub.

Read More

Recent | All Posts | Java | Tutorials

Integrating React.js into Existing jQuery Web Applications

Java 8 Concurrency Tutorial: Atomic Variables and ConcurrentMap

Java 8 Concurrency Tutorial: Synchronization and Locks

Java 8 Concurrency Tutorial: Threads and Executors