Nashorn jsr223 engine notes

Nashorn implements javax.script [http://docs.oracle.com/javase/7/docs/api/javax/script/package-frame.html] API. This page has specific info regarding nashorn script engine as well as nashorn specific scripting extensions under **jdk.nashorn.api.scripting** package.

Basic script engine usage document is [http://download.java.net/jdk8/docs/technotes/guides/scripting/nashorn/]

In this page, where ever "context" is mentioned it denotes **javax.script.ScriptContext** instance being used by script engine to evaluate scripts. "engine" represents a **javax.script.ScriptEngine** instance (nashorn engine instance).

Setting options for Nashorn script engine

Nashorn script engine allows customization options via a System property called "nashorn.args". By default, nashorn script engine sets **-doe** (dump stack trace on error) option. If you want to override to add/modify it, you can specify **-Dnashorn.args=<nashorn options>** in your Java command line. You can check out the list of options available by using the nashorn command line shell tool "jjs" which is available under $JDK_HOME/bin directory.

It is also possible to create a nashorn engine by passing customizing options programmatically:

```
import jdk.nashorn.api.scripting.NashornScriptEngineFactory;

NashornScriptEngineFactory factory = new NashornScriptEngineFactory();
ScriptEngine engine = factory.getScriptEngine(new String[] { "--global-per-engine" });
```

ScriptContext and Bindings

A ScriptContext contains one or more Bindings each associated each jsr223 "scope". By default, there are two scopes, namely ENGINE_SCOPE and GLOBAL_SCOPE. When nashorn engine is created it creates a default context.

```
ScriptContext defaultContext = engine.getContext();
```

The default context's ENGINE_SCOPE is a wrapped instance of ECMAScript "global" object - which is the "this" in top level script expressions. So, you can access ECMAScript top-level objects like "Object", "Math", "RegExp", "undefined" from this scope object. Nashorn Global scope object is represented by an internal implementation class called **jdk.nashorn.internal.objects.Global**. Instance of this class is wrapped as a **jdk.nashorn.api.scripting.ScriptObjectMirror** instance. ScriptObjectMirror class implements **javax.script.Bindings** interface. Please note that the context's GLOBAL_SCOPE Bindings and nashorn global object are different. Nashorn's global object is associated with ENGINE_SCOPE and not with GLOBAL_SCOPE. GLOBAL_SCOPE object of default script context is a javax.script.SimpleBindings instance.The user can fill it with name, value pairs from the java code.

```
Bindings b = engine.getContext().getBindings(ScriptContext.ENGINE_SCOPE);
System.out.println(b.get("Object")); // gets ECMAScript "Object" constructor
System.out.println(b.get("undefined")); // ECMAScript 'undefined' value
```

If you create a new ScriptContext object and use it to evaluate scripts, then ENGINE_SCOPE of that context has to be associated with a nashorn Global object somehow - or else script execution is not possible with that context - this is because evaluated script expects standard ECMAScript global builtins always. You could copy default script context's ENGINE_SCOPE to your new context.

```
ScriptContext myContext = new SimpleScriptContext();
myContext.setBindings(defaultContext.getBindings(ScriptContext.ENGINE_SCOPE);
engine.eval(myScript, myContext);
```

In that case, script references to "Object", "Function" etc. will use definitions in the default context's ENGINE_SCOPE. But if you want you can create a new Bindings backed by a nashorn Global scope.

```
myContext.setBindings(engine.createBindings(), ScriptContext.ENGINE_SCOPE);
engine.eval(myScript, myContext); // script runs with a new ECMAScript global scope
```

The above code creates a fresh nashorn global object and makes a Bindings out of it. When user refers to "Object" constructor, it is a different Object constructor than the one in default context's ENGINE_SCOPE.

But, user can supply any ScriptContext implementation containing any Bindings object as ENGINE_SCOPE, nashorn engine cannot always assume ENGINE_SCOPE Bindings to be backed by a nashorn Global instance. Nashorn engine checks if ENGINE_SCOPE of the ScriptContext is backed by a Nashorn Global object or not. If not, it creates a fresh Bindings backed by a nashorn Global instance and associates the same with the ENGINE_SCOPE that the user provided.

```
ScriptContext myNewContext = new SimpleScriptContext();
// ENGINE_SCOPE is a SimpleBindings instance
engine.eval(myScript, myNewContext);
// nashorn engine associates a fresh nashorn Global with the ENGINE_SCOPE
Object obj = myNewContext.getBindings(ScriptContext.ENGINE_SCOPE);
Object nashornGlobal = ((Bindings)obj).get("nashorn.global");
// "nashorn.global" is the key used to associate
ScriptObjectMirror globalMirror = (ScriptObjectMirror) nashornGlobal;
globalMirror.get("Function"); // get "Function" constructor object from nashorn global object
```

When a script attempts to access a global variable not defined within it, nashorn searches for the variable in Bindings of the current ScriptContext used.

```
ScriptContext defCtx = engine.getContext();
defCtx.getBindings(ScriptContext.GLOBAL_SCOPE).put("foo", "hello");
ScriptContext myCtx = new SimpleScriptContext();
myCtx.setBindings(defCtx.getBindings(ScriptContext.ENGINE_SCOPE), ScriptContext.ENGINE_SCOPE);
Bindings b = new SimpleBindings(); b.put("foo", "world");
myCtx.setBindings(b, ScriptContext.GLOBAL_SCOPE);

engine.eval("print(foo)"); // prints 'hello'
engine.eval("print(foo)", myCtx); // prints "world"
engine.eval("print(foo)", defCtx); // prints "hello"
```

But, when you do

```
engine.eval("foo = 2");
```

nashorn engine will **not** modify GLOBAL_SCOPE Bindings "foo". Instead it will create a new variable in nashorn Global object with the name "foo" shadowing the GLOBAL_SCOPE's "foo". **Implementation note:** every nashorn script object can have a special __noSuchProperty__ function property. This is invoked whenever a property is not found in the object. Nashorn script engine installs __noSuchProperty__ on nashorn's Global object and in that function it searches Bindings of ScriptContext for any missing property. If not found in any of the ScriptContext's Bindings, ReferenceError is thrown.

The following nashorn test has code that demonstrates various 'scope' cases discussed above:

$nashorn_repo/test/src/jdk/nashorn/api/scripting/ScopeTest.java

--global-per-engine option

The above description is about nashorn engine's default mode. It is possible to customize the above by using nashorn customization option --global-per-engine. If this option is specified, nashorn script engine behaves slightly differently. A single nashorn global object backed Bindings object is stored in engine instance. ScriptEngine.createBindings returns an instance of SimpleBindings. i.e., it does not create a fresh nashorn global object backed Bindings. Also, if a ScriptContext is passed for eval that contains a SimpleBindings (or any Bindings not backed by nashorn global object), nashorn engine associates the single nashorn global object created during engine initialization. The rationale is that all script evaluations regardless of ScriptContext passed - share the same nashorn global object and so "Object", "Function" etc. refer to the same objects. But, user can still pass other Bindings instances for GLOBAL_SCOPE (or other scopes) in ScriptContext instance. Nashorn will still search those Bindings for missing global properties.

ScriptObjectMirror and JSObject

Nashorn represents script objects created by script as instances of jdk.nashorn.internal.runtime.ScriptObject or a subclass of it. For example, nashorn global object is an instance of jdk.nashorn.internal.objects.Global class. These are implementation classes and therefore can not be accessed by user code. Under security manager, attempt to access this class or any subclass will result in SecurityException being thrown. **jdk.nashorn.api.scripting.ScriptObjectMirror** class is the API entry point for nashorn "ECMAScript script objects". Whenever "eval" results in a script object value (i.e., not a Java object or any "foreign" object), the script object is returned as ScriptObjectMirror.

```
ScriptObjectMirror sobj = (ScriptObjectMirror)engine.eval("({ foo: 23 }");
System.out.println(sobj.get("foo")); // prints 23
```

ScriptObjectMirror class implements **jdk.nashorn.api.scripting.JSObject** interface and javax.script.Bindings interface. JSObject exposed methods like getMember, getSlot, setMember, setSlot, call etc. to access properties and call functions.

The following tests in nashorn repo have detailed test methods explaining the usage of ScriptObjectMirror:

$nashorn_repo/test/src/jdk/nashorn/api/scripting/ScriptEngineTest.java
$nashorn_repo/test/src/jdk/nashorn/api/scripting/ScriptObjectMirrorTest.java

It is also possible to supply user's implementation of jdk.nashorn.api.scripting.JSObject. Wherever possible, nashorn engine tries to treat instances of ScriptObjectMirror and JSObject as though they are "normal" script objects. Nashorn uses flexible dynalink linker to give this illusion. For example, script use familiar "obj.foo", "obj[3]", "obj.bar()", "delete obj.foo" on JSObjects as well as on ScriptObjectMirror instances (from other nashorn globals or even other nashorn script engines).

The following nashorn test demonstrates user supplied JSObject implementation:

$nashorn_repo/test/src/jdk/nashorn/api/scripting/PluggableJSObjectTest.java

ScriptObjectMirror conversion

If you a java method called from script accepts ScriptObjectMirror as a parameter, nashorn converts script object to ScriptObjectMirror - so that your method can manipulate script objects via method of the class ScriptObjectMirror. ~~Note that this auto-conversion from script object to ScriptObjectMirror won't happen if you use "Object" as the param type of your Java method. Also, if you put script object into a Java Object[] or any other Collection like a List, Map, script object conversion won't happen.~~ If you convert a script array of script objects via **Java.to** API to ScriptObjectMirror[], then the script object to mirror conversion will happen. With jdk8u40 onwards, script objects are converted to ScriptObjectMirror whenever script objects are passed to Java layer - even with Object type params or assigned to a Object[] element. Such wrapped mirror instances are automatically unwrapped when execution crosses to script boundary. i.e., say a Java method returns Object type value which happens to be ScriptObjectMirror object, then script caller will see it a ScriptObject instance (mirror gets unwrapped automatically)

Explicit script object mirror wrapping

If you want to explicitly wrap a script object to a ScriptObjectMirror instance and pass along to any API (which may even accept Object), you can use **jdk.nashorn.api.scripting.ScriptUtils** class's **wrap** method. Similarly you can explicitly unwrap a ScriptObjectMirror to a script object via **unwrap** method. Note that unwrap won't unwrap the object if it is not a wrapper of object created with the current global object. In most instances you won't have to wrap /unwrap manually. **In Java code, always operate on ScriptObjectMirror instances and in script you'll always see normal ScriptObject (unwrapped) instances automatically.**

JavaDoc for Nashorn specific API

jdk.nashorn.api.* is the only Nashorn specific API package (extension to javax.script). Anything else is considered to be Nashorn implementation detail

jdk8: https://docs.oracle.com/javase/8/docs/jdk/api/nashorn/index.html?jdk/nashorn/api/scripting/package-summary.html

jdk9 http://download.java.net/jdk9/docs/jdk/api/nashorn/

Note: if you clone nashorn openjdk repository, you can locally generate this javadoc using "javadocapi" ant target.

Limitations/Known issues

While nashorn attempts to give a seamless illusion of ScriptObjectMirrors and JSObjects, not every operation and script API (JSON, Array, Function's properties/functions) treats ScriptObjectMirror and jdk.nashorn.internal.runtime.ScriptObject uniformly. There are places where ScriptObjects work as expected but if you pass ScriptObjectMirror or your own JSObject implementation, it won't work as expected.

Security Permissions for Nashorn scripts

Nashorn script security permissions