Nashorn extensions

Nashorn Syntax Extensions

Nashorn implements ECMAScript 5.1 specification —***http://www.ecma-international.org/ecma-262/5.1/***

HTML non-normative version of the spec is here: http://es5.github.io/

Nashorn implements number of syntactic and API extensions as well. This page describes syntax and API extensions of nashorn implementation. A number of syntax and API extensions are enabled only when scripting mode is enabled. Scripting mode is turned on by nashorn option "-scripting". Whenever a syntax or an API extension is available only in -scripting mode, it is noted so in this document.

Conditional catch clauses

This Mozilla JavaScript 1.4 extension is also supported by Nashorn. A single try.. statement can have more than one catch clause each with it's own catch-condition.

Conditional catch clause example

```
try {
    func()
} catch (e if e instanceof TypeError) {
     // handle TypeError here
} catch (e) {
    // handle any other..
}
```

Function expression closures

This Mozilla JavaScript 1.8 extension is also supported by Nashorn. This syntax allows braces and return keyword to be dropped when defining simple one-liner functions. See also https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/1.8

Example:

Closure expression example

```
function sqr(x) x*x

// is equivalent to
// function sqr(x) { return x*x }
```

For each expressions

This is a Mozilla JavaScript 1.6 extension supported by Nashorn. ECMAScript for..in iterates over property names or array indices of an object. for..each..in loop iterates over property values of an object rather than property names/indices. See also https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for_each...in

for each loop

```
// for each (variable in object) { statement }

// print each array element
var arr = [ "hello", "world" ];
for each (a in arr) {
 print(a)
}
```

for..each works on Java arrays as well as any Java Iterable objects.

for each Java arrays example

```
var JArray = Java.type("int[]");
// create 10 element int array
var arr = new JArray(10);

// assign squares
for (i in arr) {
   arr[i] = i*i;
}

// print values of each array element
for each (i in arr) {
   print(i);
}
```

for..each Java map

```
var System  = Java.type("java.lang.System")

// print all Java System property name, value pairs
for each (p in System.properties.entrySet()) {
    print(p.key, "=", p.value)
}

// print all environment variables with values
for each (e in System.env.entrySet()) {
  print(e.key, "=", e.value)
}
```

New expression with last argument after ")"

This is another Rhino extension supported by Nashorn. In a new Constructor(x, y) expression, the last argument can be specified after ")" if it happens to be an object literal. Example:

new anonymous class-like syntax

```
// This syntax is primarily used to support anonymous class-like syntax for
// Java interface implementation as shown below.

var r = new java.lang.Runnable() {
    run: function() { print("run"); }
}
```

Anonymous function statements

Top-level function statements can be anonymous. Example:

anonymous function statements

```
function () {
    print("hello")
}
```

How do you call it then? Say if you had evaluated the above code by "eval" or script engine's "eval" call from Java, you can get the return value to be the function object - which can be invoked later.

Multi-line string literals (-scripting mode only)

Nashorn supports multi-line string literals with Unix shell's here-doc syntax. Example:

multi-line string literals

```
var str = <<EOF

This is a string that contains multiple lines
hello
world
That's all!

EOF

print(str)
```

String interpolation (-scripting mode only)

Expressions can be specified inside string literals with the syntax ${expression}. The string value is computed by substituting value of expressions for ${expression} in the string.

String expression interpolation

```
var x = "World"
var str = "Hello, ${x}"

print(str) // prints "Hello, World" because ${x} is substituted with value of variable
"x"
```

Back-quote exec expressions (-scripting mode only)

Nashorn supports Unix shell like back quote strings. Back quoted strings are evaluated by executing the programs mentioned in the string and returning value produced by the 'exec'-ed program.

back quote exec strings

```
// exec "ls -l" to get file listing as a string
var files = `ls -l`
var lines = files.split("\n");

// print only the directories
for (var l in lines) {
    var line = lines[l];
    if (line.startsWith("d")) // directory
        print(line)
}
```

Shell script style hash comments (-scripting mode only)

In -scripting mode, nashorn accept shell script style # (hash) single line comments.

 # comment example

```
# style line comment -scripting mode
# prints hello

print("hello")
```

Nashorn supports shebang scripting. You need to create a link to jjs from your platform executable directory like /usr/bin as shown below.

Shebang scripts

```
$ cd /usr/bin
$ sudo ln -s $JAVA_HOME/bin/jjs jjs
$ chmod 755 test.js

where test.js is as follows:

#!/usr/bin/jjs
print("hello")

$ ./test.js
```

For shebang nashorn scripts, -scripting mode is automatically enabled.

Switching off syntax extensions

Nashorn option --no-syntax-extensions (or it's short form -nse) can be used to switch off syntax extensions of nashorn. With syntax extensions switched off, only ECMAScript standard syntax is supported. Note that even when -scripting mode is on, -nse switches off scripting syntax extensions as well. Also note that API extensions are still enabled even -nse is specified.

Nashorn script API extensions

__proto__ property

Like other ECMAScript implementations (like Rhino, v8) Nashorn supports mutable __proto__ magic property to read and write prototype of a given object. See also https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto

__proto__ is deprecated. Please avoid using __proto__. Use Object.getPrototypeOf and Object.setPrototypeOf instead.

Object.setPrototypeOf(obj, newProto)

ECMAScript specifies Object.getPrototypeOf(obj) http://es5.github.io/#x15.2.3.2 function to get prototype of the given object. This Object.setPrototypeOf is a nashorn specific extension that allows prototype of an object to be set as newProto. Object.setPrototypeOf is one of proposed extensions in ECMAScript 6.

See also https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf

__noSuchProperty__

Any object can have this "property missing" hook method. When a property accessed is not found on the object, this __noSuchProperty__ hook will be called.

noSuchProperty

```
var obj = {
  __noSuchProperty__: function(n) {
    print("No such property: " + n);
  }
}

obj.foo;     // prints No such property: foo

obj["bar"];  // prints No such property: bar
```

__noSuchMethod__

Any object can have this "method missing" hook method. When a method is called is not found on the object, this __noSuchMethod__ hook will be called.

noSuchMethod

```
var obj = {
  __noSuchMethod__: function() {
    for (i in arguments) {
       print(arguments[i])
    }
}

obj.foo(2, 'hello'); // prints "foo", 2, "hello"
```

Typed Arrays

Nashorn implements typed arrays as specified at https://www.khronos.org/registry/typedarray/specs/latest/ For a tutorial presentation, please check out https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays (which includes non-standard StringView which is not implemented by nashorn). Among the typed array constructors, DataView (https://www.khronos.org/registry/typedarray/specs/latest/#8) is not implemented in jdk8's nashorn. DataView is included in jdk8u-dev and so is expected to make it into jdk8u20 release.

Object.bindProperties

Object.bindProperties lets you bind properties of one object to another object. Example:

Object.bindProperties

```
var obj = {}

var obj2 = { foo: 344 }

// bind properties of 'obj2' to 'obj'
Object.bindProperties(obj, obj2);

print(obj.foo);    // obj2.foo
obj.foo = "hello"  // obj2.foo assigned
print(obj2.foo);   // prints "hello"

// bind to global 'this' is also possible
Object.bindProperties(this, obj2);

print(foo);        // prints obj2.foo
foo = 42;          // assigns to obj2.foo
print(obj2.foo);   // prints 42
```

The bound object need not be a script object. It could be a Java object as well - usual Java bean style getter/setter properties will be bound. Also a property only with getter will be bound as read-only property.

```
The source object bound can be a ScriptObject or a ScriptOjectMirror.
null or undefined source object results in TypeError being thrown.

Limitations of property binding:

* Only enumerable, immediate (not proto inherited) properties of the source object are bound.
* If the target object already contains a property called "foo", the source's "foo" is skipped (not bound).
* Properties added to the source object after binding to the target are not bound.
* Property configuration changes on the source object (or on the target) is not propagated.
* Delete of property on the target (or the source) is not propagated -
only the property value is set to 'undefined' if the property happens to be a data property.

It is recommended that the bound properties be treated as non-configurable
properties to avoid surprises.
```

Extensions of Error objects, Error.prototype and Error constructor

Nashorn extends ECMAScript Error (or subtype) objects with the following properties.

- **lineNumber** - source code line number from which error object was thrown
- **columnNumber** - source code column number from which error object was thrown
- **fileName** - source script file name
- **stack** - script stack trace as a string

In addition to this Error.prototype has the following extension functions as well.

- **Error.prototype.printStackTrace()** - prints full stack trace (including all Java frames) from where error was thrown from
- **Error.prototype.getStackTrace()** - returns an array of java.lang.StackTraceElement instance for ECMAScript frames only.

Error constructor includes dumpStack() function property

- **Error.dumpStack()** - prints stack trace of the current thread - just like java.lang.Thread.dumpStack()

Error extensions

```
function func() {
    throw new Error()
}

function f() {
    func()
}

try {
    f()
} catch (e) {
    print(e.stack)
    print(e.lineNumber)
    print(e.columnNumber)
    print(e.fileName)
}
```

String.prototype extensions

String.prototype.trimLeft - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/TrimLeft

String.prototype.trimRight - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/TrimRight

Function.prototype.toSource

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/toSource

Extension properties, functions in global object

__FILE__ , __LINE__, __DIR__

Current executing script file, line, directory of the script can be accessed by these global (read-only) properties. This is useful in debug/tracing scripts. __DIR__ can be used like Unix 'dirname' command. Script can load script-file relative path resources using __DIR__.

_FILE_* and *_LINE_

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

print function

print function prints it's arguments (after converting those to Strings) on standard output

print

```
print("hello", "world")
```

load function

Loads and evaluates script from a file or a URL or a script object. Examples:

load function

```
// can load script from files, URLs

load("foo.js"); // loads script from file "foo.js" from current directory
load("http://www.example.com/t.js"); // loads script file from given URL

// loads script from an object's properties.

// Object should have "script" and "name" properties.
//     "script" property contains string code of the script.
//     "name" property specifies name to be used while reporting errors from script
// This is almost like the standard "eval" except that it associates a name with
// the script string for debugging purpose.

load({ script: "print('hello')", name: "myscript.js"})

// load can also load from pseudo URLs like "nashorn:", "fx:", "classpath:"

// "nashorn:" pseudo URL scheme
// for nashorn's built-in scripts.

// load nashorn's parser support script - defines 'parse'
// function in global scope

load("nashorn:parser.js");

// load Mozilla compatibility script - which defines global functions
// like importPackage, importClass for rhino compatibility.

load("nashorn:mozilla_compat.js");

// "fx:" pseudo URL scheme for JavaFX support scripts
// "classpath:" pseudo URL scheme to load scripts from jjs classpath jars and
directories

load("classpath:foo.js"); // load the first foo.js found in jjs -classpath dir
```

loadWithNewGlobal function

loadWithNewGlobal is almost like the function "load" in that it loads script from a file or a URL or a script object. But unlike 'load', loadWithNewGlobal creates a fresh ECMAScript global scope object and loads the script into it. The loaded script's global definitions go into the fresh global scope. Also, the loaded script's modifications of builtin objects (like String.prototype.indexOf) is not reflected in the caller's global scope – these are affected in the newly created global scope.

loadWithNewGlobal

```
loadWithNewGlobal({
    script: "foo = 333; print(foo)",
    name: "test"
});

// prints undefined as "foo" is defined in the new global and not here
print(typeof foo);
```

exit and quit functions

These synonymous functions exit the current process with specified (optional) exit code.

exit

```
exit(1); // exit with given exit code
exit();  // exit 0

quit(2); // exit with given exit code
quit();  // exit 0
```

Packages, java, javax, javafx, com, edu, org

Packages and related objects are there to support java package, class access from script. The properties of the `Packages` variable are all the top-level Java packages, such as `java` and `javax etc.`

Packages

```
var Vector = Packages.java.util.Vector;

// but short-cuts defined for important package prefixes like
//    Packages.java, Packages.javax, Packages.com
//    Packages.edu, Packages.javafx, Packages.org

var JFrame = javax.swing.JFrame;  // javax == Packages.javax
var List = java.util.List;        // java == Packages.java
```

jjs session of Packages, classes

```
jjs> Packages.java
[JavaPackage java]
jjs> java
[JavaPackage java]
jjs> java.util.Vector
[JavaClass java.util.Vector]
jjs> javax
[JavaPackage javax]
jjs> javax.swing.JFrame
[JavaClass javax.swing.JFrame]
```

JavaImporter constructor

Many times you may want to import many Java packages – but without having to "pollute" global scope. JavaImporter helps in such situations.

JavaImporter

```
// JavaImporter constructor accepts one or more Java Package objects
var imports = new JavaImporter(java.util, java.io);

// a JavaImporter can be used as a "with" expression object
with(imports) {
    // classes from java.util and java.io packages can
    // can be accessed by unqualified names

    var map = new HashMap(); // refers to java.util.HashMap
    map.put("js", "javascript");
    map.put("java", "java");
    map.put("cpp", "c++");
    print(map);

    var f = new File("."); // refers to java.io.File
    print(f.getAbsolutePath());
}
```

JavaImporter in 'with' statement example

```
// read text content from the given URL

function readText(url) {
    // Using JavaImporter to resolve classes
    // from specified java packages within the
    // 'with' statement below

    with (new JavaImporter(java.io, java.net)) {
        // more or less regular java code except for static types
        var is = new URL(url).openStream();
        try {
            var reader = new BufferedReader(
                new InputStreamReader(is));
            var buf = '', line = null;
            while ((line = reader.readLine()) != null) {
                buf += line;
            }
        } finally {
            reader.close();
        }
        return buf;
    }
}

print(readText("http://google.com?q=jdk8"))
```

Java object

"Java" global property is a script object that defines useful functions for script-to-Java interface.

Java.type function

Given a name of a Java type, returns an object representing that type in Nashorn. The Java class of the objects used to represent Java types in Nashorn is not `Class` but rather StaticClass. They are the objects that you can use with the `new` operator to create new instances of the class as well as to access static members of the class. In Nashorn, `Class` objects are just regular Java objects that aren't treated specially. Instead of them, StaticClass instances - which we sometimes refer to as "Java type objects" are used as constructors with the `new` operator, and they expose static fields, properties, and methods. While this might seem confusing at first, it actually closely matches the Java language: you use a different expression (e.g. `java.io.File`) as an argument in "new" and to address statics, and it is distinct from the `Class` object (e.g. `java.io.File.class`). To get StaticClass object corresponding to a Java Class object, you can use "static" property. Below we cover in details the properties of the type objects.

Java.type

```
var arrayListType = Java.type("java.util.ArrayList")
 var intType = Java.type("int")
 var stringArrayType = Java.type("java.lang.String[]")
 var int2DArrayType = Java.type("int[][]")

 // Note that the name of the type is always a string for a fully
 // qualified name. You can use any of these types to create
 // new instances, e.g.:

 var anArrayList = new Java.type("java.util.ArrayList")

 // or

 var ArrayList = Java.type("java.util.ArrayList")
 var anArrayList = new ArrayList
 var anArrayListWithSize = new ArrayList(16)


 var BoolArray = Java.type("boolean[]");
 var arr = new BoolArray(10);
 arr[0] = true;

 // In the special case of inner classes, you can either use the JVM fully
 // qualified name, meaning using $ sign in the class name, or you can
 // use the dot:

 var ftype = Java.type("java.awt.geom.Arc2D$Float")

 // and this works too:

 var ftype = Java.type("java.awt.geom.Arc2D.Float")

 // Java Class object to type:
 // similar to Java.type("java.util.Vector")

 var Class = Java.type("java.lang.Class")
 var VectorClass = Class.forName("java.util.Vector")
 var Vector = VectorClass.static;
```

If the type is abstract, you can instantiate an anonymous subclass of it using an argument list that is applicable to any of its public or protected constructors, but inserting a JavaScript object with functions properties that provide JavaScript implementations of the abstract methods. If method names are overloaded, the JavaScript function will provide implementation for all overloads. E.g.:

new on abstract class

```
var TimerTask =  Java.type("java.util.TimerTask")
var task = new TimerTask({ run: function() { print("Hello World!") } })
```

Nashorn supports a syntactic extension where a "new" expression followed by an argument is identical to invoking the constructor and passing the argument to it, so you can write the above example also as:

anonymous class like expression

```
var task = new TimerTask {
    run: function() {
       print("Hello World!")
    }
}
```

which is very similar to Java anonymous inner class definition. On the other hand, if the type is an abstract type with a single abstract method (commonly referred to as a "SAM type") or all abstract methods it has share the same overloaded name), then instead of an object, you can just pass a function, so the above example can become even more simplified to:

SAM

```
var task = new TimerTask(function() { print("Hello World!") })
```

The use of functions can be taken even further; if you are invoking a Java method that takes a SAM type, you can just pass in a function object, and Nashorn will know what you meant:

SAM function conversion

```
var Timer = Java.type("java.util.Timer")
 var timer = new Timer()
 timer.schedule(function() { print("Hello World!") }, 1000)
 java.lang.System.in.read()
```

Here, `Timer.schedule()` expects a `TimerTask` as its argument, so Nashorn creates an instance of a `TimerTask` subclass and uses the passed function to implement its only abstract method, `run()`. In this usage though, you can't use non-default constructors; the type must be either an interface, or must have a protected or public no-arg constructor.

Java.extend function

Java.extend function returns a type object for a subclass of the specified Java class (or implementation of the specified interface) that acts as a script-to-Java adapter for it. Note that you can also implement interfaces and subclass abstract classes using `new` operator on a type object for an interface or abstract class. However, to extend a non-abstract class, you will have to use this method. Example:

Java.extend examples

```
var ArrayList = Java.type("java.util.ArrayList")
var ArrayListExtender = Java.extend(ArrayList)
var printSizeInvokedArrayList = new ArrayListExtender() {
    size: function() { print("size invoked!"); }
}

var printAddInvokedArrayList = new ArrayListExtender() {
    add: function(x, y) {
      if(typeof(y) === "undefined") {
          print("add(e) invoked!");
      } else {
          print("add(i, e) invoked!");
      }
    }
}
```

Java.extend - larger example

```
#// Usage: jjs javafoovars.js -- <directory>

// This example demonstrates Java subclassing by Java.extend
// and javac Compiler and Tree API. This example counts number
// of variables called "foo" in the given java source files!
if (arguments.length == 0) {
    print("Usage: jjs javafoovars.js -- <directory>");
    exit(1);
}

// Java types used
var File = Java.type("java.io.File");
var Files = Java.type("java.nio.file.Files");
var FileVisitOption = Java.type("java.nio.file.FileVisitOption");
var StringArray = Java.type("java.lang.String[]");
var ToolProvider = Java.type("javax.tools.ToolProvider");
var Tree = Java.type("com.sun.source.tree.Tree");
var TreeScanner = Java.type("com.sun.source.util.TreeScanner");
var VariableTree = Java.type("com.sun.source.tree.VariableTree");

// count "foo"-s in the given .java files
function countFoo() {
    // get the system compiler tool
    var compiler = ToolProvider.systemJavaCompiler;
    // get standard file manager
    var fileMgr = compiler.getStandardFileManager(null, null, null);
    // Using Java.to convert script array (arguments) to a Java String[]
    var compUnits = fileMgr.getJavaFileObjects(
        Java.to(arguments, StringArray));
    // create a new compilation task
    var task = compiler.getTask(null, fileMgr, null, null, null, compUnits);
    // subclass SimpleTreeVisitor - to count variables called "foo"
    var FooCounterVisitor = Java.extend(TreeScanner);
    var fooCount = 0;
```

```
    var visitor = new FooCounterVisitor() {
        visitVariable: function (node, p) {
            if (node.name.toString() == "foo") {
                fooCount++;
            }
        }
    }

    for each (var cu in task.parse()) {
        cu.accept(visitor, null);
    }
    return fooCount;
}

// for each ".java" file in directory (recursively) count "foo".
function main(dir) {
    var totalCount = 0;
    Files.walk(dir.toPath(), FileVisitOption.FOLLOW_LINKS).
      forEach(function(p) {
        var name = p.toFile().absolutePath;
        if (name.endsWith(".java")) {
            var count = 0;
            try {
                count = countFoo(p.toFile().getAbsolutePath());
            } catch (e) {
                print(e);
            }
            if (count != 0) {
                print(name + ": " + count);
            }
            totalCount += count;
        }
      });
    print("Total foo count: " + totalCount);
```

```
}

main(new File(arguments[0]));
```

Java.from function

Given a Java array or Collection, this function returns a JavaScript array with a shallow copy of its contents. Note that in most cases, you can use Java arrays and lists natively in Nashorn; in cases where for some reason you need to have an actual JavaScript native array (e.g. to work with the array comprehensions functions), you will want to use this method. Example:

Java.from examples

```
var File = Java.type("java.io.File")
var listHomeDir = new File("~").listFiles()

// Java array to JavaScript array conversion by Java.from
var jsListHome = Java.from(listHomeDir)

var jpegModifiedDates = jsListHome
    .filter(function(val) { return val.getName().endsWith(".jpg") })
    .map(function(val) { return val.lastModified() })
```

Java.to function

Given a script object and a Java type, converts the script object into the desired Java type. Currently it performs shallow creation of Java arrays, as well as wrapping of objects in Lists and Dequeues. Example:

Java.to example

```
var anArray = [1, "13", false]

// Java.to used to convert script array to a Java int array
var javaIntArray = Java.to(anArray, "int[]")

print(javaIntArray[0])  // prints 1
print(javaIntArray[1])  // prints 13, as string "13" was converted to number 13
                        // as per ECMAScript ToNumber conversion
print(javaIntArray[2])  // prints 0, as boolean false was converted to number 0
                        // as per ECMAScript ToNumber conversion
```

Java.super function

When given an object created using `Java.extend()` or equivalent mechanism (that is, any JavaScript-to-Java adapter), Java.super returns an object that can be used to invoke superclass methods on that object. E.g.:

Java.super example

```
var CharArray = Java.type("char[]")
var jString = Java.type("java.lang.String")
var Character = Java.type("java.lang.Character")

function capitalize(s) {
    if(s instanceof CharArray) {
        return new jString(s).toUpperCase()
    }
    if(s instanceof jString) {
        return s.toUpperCase()
    }
    return Character.toUpperCase(s) // must be int
}

var sw = new (Java.type("java.io.StringWriter"))

var FilterWriterAdapter = Java.extend(Java.type("java.io.FilterWriter"))

var cw = new FilterWriterAdapter(sw) {
    write: function(s, off, len) {
        s = capitalize(s)
        // Must handle overloads by arity
        if(off === undefined) {
            cw_super.write(s, 0, s.length())
        } else if (typeof s === "string") {
            cw_super.write(s, off, len)
        }
    }
}
var cw_super = Java.super(cw)

cw.write("abcd")
cw.write("e".charAt(0))
cw.write("fgh".toCharArray())
cw.write("**ijk**", 2, 3)
cw.write("***lmno**".toCharArray(), 3, 4)
cw.flush()
print(sw)
```

Java subclassing - Java.extend and Java.super - a larger example:

Java.extend and Java.super - larger example

```
#// Usage: jjs -fx filebrowser.js -- <start_dir>

// Uses -fx and javafx TreeView to visualize directories
if (!$OPTIONS._fx) {
    print("Usage: jjs -fx filebrowser.js -- <start_dir>");
    exit(1);
}

// Java classes used
var File = Java.type("java.io.File");
```

```
var Files = Java.type("java.nio.file.Files");

// check directory argument, if passed
var dir = arguments.length > 0? new File(arguments[0]) : new File(".");
if (! dir.isDirectory()) {
    print(dir + " is not a directory!");
    exit(2);
}

// JavaFX classes used
var FXCollections = Java.type("javafx.collections.FXCollections");
var Scene     = Java.type("javafx.scene.Scene");
var TreeItem  = Java.type("javafx.scene.control.TreeItem");
var TreeView  = Java.type("javafx.scene.control.TreeView");

// create a subclass of JavaFX TreeItem class
var LazyTreeItem = Java.extend(TreeItem);

// lazily filling children of a directory LazyTreeItem
function buildChildren(dir) {
    var children = FXCollections.observableArrayList();
    var stream = Files.list(dir.toPath());
    stream.forEach(function(path) {
        var file = path.toFile();
        var item = file.isDirectory()?
            makeLazyTreeItem(file) : new TreeItem(file.name);
        children.add(item);
    });
    stream.close();
    return children;
}

// create an instance LazyTreeItem with override methods
function makeLazyTreeItem(dir) {
    var item = new LazyTreeItem(dir.name) {
        expanded: false,
        isLeaf: function() false,
        getChildren: function() {
            if (! this.expanded) {
                // call super class (TreeItem) method
                Java.super(item).getChildren().setAll(buildChildren(dir));
                this.expanded = true;
            }
            // call super class (TreeItem) method
            return Java.super(item).getChildren();
        }
    }
    return item;
}

// JavaFX start method
function start(stage) {
    stage.title = dir.absolutePath;
    var rootItem = makeLazyTreeItem(dir);
    rootItem.expanded = true;
    var tree = new TreeView(rootItem);
```

```
    stage.scene = new Scene(tree, 300, 450);
    stage.show();
}
```

Java.synchronized function

Returns synchronized wrapper version of the given ECMAScript function.

Java.synchronized example

```
var Thread = Java.type("java.lang.Thread");
var sum = 0;
var lock = {};

function run() {
    Thread.sleep(Math.floor(Math.random()*700) + 300);

    // create synchronized wrapper of given function
    // and use the second param as the synchronization lock
    Java.synchronized(function() sum++, lock)();
}

var threads = [];
for (var i = 0; i < 4; i++) {
    var t = new Thread(run);
    threads.push(t);
    t.start();
}

for (var i in threads) {
   threads[i].join();
}

// always prints 4
print(sum);
```

Java.asJSONCompatible function

This function accepts a script object and returns an object that is compatible with Java JSON libraries expectations; namely, that if it itself, or any object transitively reachable through it is a JavaScript array,then such objects will be exposed as JSObject that also implements the List interface for exposing the array elements.

An explicit API is required as otherwise Nashorn exposes all objects externally as JSObjects that also implement the Map interface instead. By using this method, arrays will be exposed as Lists and all other objects as Maps.

This API is since jdk 8u60+ and jdk 9. A simple example that uses Java.asJSONCompatible function:

Java.asJSONCompatible example

```
import javax.script.*;
import java.util.*;
public class JSONTest {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager m = new ScriptEngineManager();
        ScriptEngine e = m.getEngineByName("nashorn");
        Object obj = e.eval("Java.asJSONCompatible({ x: 343, y: 'hello', z: [2,4, 5]
})");
        Map<String, Object> map = (Map<String, Object>)obj;
        System.out.println(map.get("x"));
        List<Object> array = (List<Object>)map.get("z");
        for (Object elem : array) {
            System.out.println(elem);
        }
    }
}
```

Special treatment of objects of specific Java classes

Java Map keys as properties

Nashorn allows java.util.Map instances to be treated like a script object - in that it allows indexed, property access to Map's keys.

java.util.Map keys as properties example

```
var HashMap = Java.type("java.util.HashMap")
var map = new HashMap()
// map key-value access by java get/put method calls
map.put('js', 'nashorn')
print(map.get('js'))
// access keys of map as properties
print(map['js'])
print(map.js)
// also assign new key-value pair
// as 'property-value'
map['language'] = 'java'
print(map.get("language"))
print(map.language)
print(map['language'])
map.answer = 42
print(map.get("answer"))
print(map.answer)
print(map['answer'])
```

Java List element access/update via array index access/update syntax

Elements of java.util.List objects can be access as though those are array elements - array like indexed element access and update is supported. "length" property can be used on List objects to get size of the list.

Java List elements via array index access

```
// Java List elements accessed/modified via
// array element access/update syntax
var ArrayList = Java.type("java.util.ArrayList")
var list = new ArrayList()
// add elements to list by List's add method calls
list.add("js")
list.add("ecmascript")
list.add("nashorn")
// get by List's get(int) method
print(list[0])
print(list[1])
print(list[2])
// access list elements by indexed access as well
print(list[0])
print(list[1])
print(list[2])
// assign to list elements by index as well
list[0] = list[0].toUpperCase()
list[1] = list[1].toUpperCase()
list[2] = list[2].toUpperCase()
print(list.get(0))
print(list.get(1))
print(list.get(2))
print(list[0])
print(list[1])
print(list[2])
print(list.length); // prints list.size()
```

Lambdas, SAM types and Script functions

Every function is a lambda and a SAM type object

Where ever a JDK8 lambda or SAM (single-abstract-method) type is required, an ECMAScript function can be passed as argument. Nashorn will auto-convert a script function to a lambda object or any SAM interface implementing object.

lambda, SAM example

```
var IntStream = Java.type("java.util.stream.IntStream");

// IntStream's forEach requires a lambda implementing
// java.util.function.IntConsumer. You can pass a script function!

IntStream.range(0, 100).forEach(
    function(x) {
        print(x*x)
    });

var Thread = Java.type("java.lang.Thread");

// java.lang.Thread constructor requires a java.lang.Runnable
// You can pass script function wherever a SAM is expected
new Thread(function() {
    print("I am a thread!");
}).start();
```

Every lambda is a script function

Any Java object that is an instance of lambda type can be treated like a script function.

lambda as function example

```
var JFunction = Java.type('java.util.function.Function')

var obj = new JFunction() {
   apply: function(x) { print(x*x) }
}

// every lambda object is a 'function'

print(typeof obj); // prints "function"

// 'calls' lambda as though it is a function
obj(91);
```

Explicit method selection

When you call java method from script, Nashorn automatically selects the most specific overload variant of that method (based on the runtime type of the actual arguments passed – with appropriate type conversions as needed). But sometimes you may want to manually select a specific overload variant. If so, you can specify method signature along with the method name.

Explicit method signature

```
var out = java.lang.System.out;
out["println(int)"](Math.PI); // selects PrintStream.println(int).
// prints 3 as Math.PI is converted to int
```

Explicit constructor selection (jdk9, jdk8u65)

With jdk9 and jdk8u65, explicit constructor overload selection is also supported. See Index selection of overloaded java new constructors and Explicit constructor overload selection should work with StaticClass as well

Explicit constructor selection (jdk9, jdk8u65)

```
// With jdk9, you can select a specific constructor as well.

var C = java.awt["Color(int,int,int)"];
print(new C(255, 0, 0);

var F = Java.type("java.io.File")["(String)"];
print(new F("foo.txt"));
```

--no-java option

--no-java options can be used to switch-off Java specific extensions like "Java", "Packages" object etc.

Script Proxy implementation

Sometimes you may want to create a script friendly proxy object adapting another object. You can implement such a proxy in Java code or JavaScript code.

Plugging-in your own JSObject

Nashorn exposes script objects as an object of jdk.nashorn.api.scripting.JSObject to java code. It is also possible to provide your own jdk.nashorn.api.scripting.JSObject implementation and nashorn will treat such objects specially (by calling magic methods for property getter, setter, delete etc.)

JSObject implementation example

```
import jdk.nashorn.api.scripting.AbstractJSObject;
import java.nio.DoubleBuffer;
/**
 * Simple class demonstrating pluggable script object
 * implementation. By implementing jdk.nashorn.api.scripting.JSObject
 * (or extending AbstractJSObject which implements it), you
 * can supply a friendly script object. Nashorn will call
 * 'magic' methods on such a class on 'obj.foo, obj.foo = 33,
 * obj.bar()' etc. from script.
 *
 * In this example, Java nio DoubleBuffer object is wrapped
 * as a friendly script object that provides indexed acces
```

```
 * to buffer content and also support array-like "length"
 * readonly property to retrieve buffer's capacity. This class
 * also demonstrates a function valued property called "buf".
 * On 'buf' method, we return the underlying nio buffer object
 * that is being wrapped.
 */
public class BufferArray extends AbstractJSObject {
    // underlying nio buffer
    private final DoubleBuffer buf;
    public BufferArray(int size) {
        buf = DoubleBuffer.allocate(size);
    }
    public BufferArray(DoubleBuffer buf) {
        this.buf = buf;
    }
    // called to check if indexed property exists
    @Override
    public boolean hasSlot(int index) {
        return index > 0 && index < buf.capacity();
    }
    // get the value from that index
    @Override
    public Object getSlot(int index) {
       return buf.get(index);
    }
    // set the value at that index
    @Override
    public void setSlot(int index, Object value) {
       buf.put(index, ((Number)value).doubleValue());
    }
    // do you have a property of that given name?
    @Override
    public boolean hasMember(String name) {
       return "length".equals(name) || "buf".equals(name);
    }
    // get the value of that named property
    @Override
    public Object getMember(String name) {
       switch (name) {
          case "length":
              return buf.capacity();
          case "buf":
              // return a 'function' value for this property
              return new AbstractJSObject() {
                  @Override
                  public Object call(Object thiz, Object... args) {
                      return BufferArray.this.buf;
                  }
                  // yes, I'm a function !
                  @Override
                  public boolean isFunction() {
                      return true;
                  }
              };
       }
```

```
        return null;
    }
}
```

JS example using the above JSObject implementation is as follows:

JSObject usage example

```
// Usage: jjs -scripting -cp . jsobject.js
// This sample demonstrats how to expose a
// script friendly object from your java code
// by implementing jdk.nashorn.api.scripting.JSObject
// compile the java program
`javac BufferArray.java`
// print error, if any and exit
if ($ERR != '') {
    print($ERR)
    exit($EXIT)
}
// create BufferArray
var BufferArray = Java.type("BufferArray")
var bb = new BufferArray(10)
// 'magic' methods called to retrieve set/get
// properties on BufferArray instance
var len = bb.length
print("bb.length = " + len)
for (var i = 0; i < len; i++) {
    bb[i] = i*i
}
for (var i = 0; i < len; i++) {
    print(bb[i])
}
// get underlying buffer by calling a method
// on BufferArray magic object
// 'buf' is a function member
print(typeof bb.buf)
var buf = bb.buf()
// use retrieved underlying nio buffer
var cap = buf.capacity()
print("buf.capacity() = " + cap)
for (var i = 0; i < cap; i++) {
   print(buf.get(i))
}
```

JSAdapter constructor

JSAdapter supports java.lang.reflect.Proxy-like proxy mechanism for script objects. i.e., it allows script proxy objects be implemented in script itself.

JSAdapter example

```
// A JSAdapter object with proxy-like special hooks
var obj = new JSAdapter() {
    __get__: function(name) {
        print("getter called for '" + name + "'"); return name;
    },

    __put__: function(name, value) {
        print("setter called for '" + name + "' with " + value);
    },

    __call__: function(name, arg1, arg2) {
        print("method '" + name + "' called with " + arg1 + ", " + arg2);
    },

    __new__: function(arg1, arg2) {
        print("new with " + arg1 + ", " + arg2);
    },

    __getIds__: function() {
        print("__getIds__ called");
        return [ "foo", "bar" ];
    },

    __getValues__: function() {
        print("__getValues__ called");
        return [ "fooval", "barval" ];
    },

    __has__: function(name) {ECMAscript typed arrays
        print("__has__ called with '" + name + "'");
        return name == "js";
    },

    __delete__: function(name) {
        print("__delete__ called with '" + name + "'");
        return true;
    }
};

// calls __get__
print(obj.foo);

// calls __put__
obj.foo = 33;

// calls __call__
obj.func("hello", "world");

// calls __new__
new obj("hey!", "it works!");

// calls __getIds__ to get array of properties
for (i in obj) {
    print(i);
}

// calls __getValues__ to get array of property values
for each (i in obj) {
    print(i);
```

```
}
```

Scripting mode extension objects and functions

$ARG (-scripting mode only)

This global object can be used to access the arguments passed to the script, similar to how the `arguments` object is used, for example:

$ARG example

```
$ jjs -scripting -- arg1 arg2 arg3
jjs> $ARG
arg1,arg2,arg3
jjs> $ARG[1]
arg2
```

echo (-scripting mode only)

This is a synonym to "print" function - the values passed in as arguments to be converted to strings, printed to `stdout` separated by spaces, and followed by a new line.

echo example

```
$ jjs -scripting
jjs> echo("hello world");
hello world
```

$ENV (-scripting mode only)

$ENV object exposes OS process environment variables to script.

$ENV example

```
// print $JAVA_HOME and $PATH from the OS shell

print($ENV["JAVA_HOME"])
print($ENV["PATH"])
print($ENV.JAVA_HOME)
print($ENV.PATH)
```

$EXEC (-scripting mode only)

This global function launches processes to run commands, for example:

$EXEC example

```
jjs> $EXEC("ls -l")
total 0
drwxr-xr-x+ 1 johndoe staff 4096 Aug 18 11:03 dir
-rwxrw-r--  1 johndoe staff  168 Aug 19 17:44 file.txt

jjs> $EXEC("cat", "Send this to stdout")
Send this to stdout
```

If the command does not require any input, you can launch a process using the backtick string notation. For example, instead of `$EXEC("ls -l")`, you can use `` `ls -l` ``.

$OUT (-scripting mode only)

This global object is used to store the latest standard output (`stdout`) of the process spawned by $EXEC. For example, the result of `$EXEC()` is saved to `$OUT`.

$EXEC example with $OUT usage

```
var Arrays = Java.type("java.util.Arrays")

// use curl to download JSON weather data from the net
var str = `curl
http://api.openweathermap.org/data/2.5/forecast/daily?q=Chennai&amp;mode=json&amp;unit
s=metric&amp;cnt=7`

// parse JSON
var weather = JSON.parse($OUT)

// pull out humidity as array
var humidity = weather.list.map(function(curVal) {
    return curVal.humidity
})

// Stream API to print stat
print("Humidity")
print(Arrays["stream(int[])"](humidity).summaryStatistics())

// pull maximum day time temperature
var temp = weather.list.map(function(curVal) {
    return curVal.temp.max
})

// Stream API to print stat
print("Max Temperature")
print(Arrays["stream(double[])"](temp).summaryStatistics())
```

$ERR (-scripting mode only)

This global object is used to store the latest standard error (`stderr`) of the process spawned by $EXEC.

$EXIT (-scripting mode only)

This global object is used to store the exit code of the process spawned by $EXEC. If the exit code is not zero, then the process failed

$OPTIONS (-scripting mode only)

This property exposes command line options passed to nashorn "command line".

$OPTIONS usage

```
print("-scripting = " + $OPTIONS._scripting);
print("--compile-only = " + $OPTIONS._compile_only);
print("-timezone = " + $OPTIONS._timezone.ID);
```

readFully (-scripting mode only)

This function reads the entire contents of a file passed in as a string argument and sends it to `stdout`, or you can assign the result to a variable.

readFully example

```
jjs> readFully("text.txt")
This is the contents of the text.txt file located in the current working directory.
```

readLine (-scripting mode only)

This function reads one line of input from `stdin` and sends it to `stdout`, or you can assign the result to a variable. You can also pass a string to the `readLine()` function to get a prompt line as in the following example:

readLine example

```
jjs> var name = readLine("What is your name? ")
What is your name? Bob
jjs> print("Hello, ${name}!")
Hello, Bob!
jjs>
```

Nashorn call site tracing and profiling

Nashorn supports callsite tracing and profile jjs (jjs tool) options via command line options -tcs and -pcs. You can pass these to jjs tool or set these options via "nashorn.args" System property. But these options produce trace, profile output for all scripts that are run. To avoid having to look at too much trace output (or too big NashornProfile.txt file), nashorn allows per script or per function tracing, profiling via user directives.

You can include nashorn specific user directives (directive prologues) at the start of a source script or at the start of a script function as shown below:

tracing, profiling directives

```
function func() {
    "nashorn callsite trace enterexit"; // equivalent to -tcs=enterexit
    "nashorn callsite trace miss";      // equivalent to -tcs=miss
    "nashorn callsite trace objects";   // equivalent to -tcs=objects

    print("hello");
}

func();
```

Profile directive

```
function func() {
    "nashorn callsite profile";  // equivalent to -pcs
    for(var i = 0; i < 10; i++) print("hello " + i);
}

func();
```

Just like -pcs option, profile directive will produce NashornProfile.txt file in the current dir of the application. But

profiling is enabled per script or per function only rather than globally. These nashorn directives are enabled only in nashorn debug mode. So, these are effective only when jjs is run -J-Dnashorn.debug option set or script engine is initialized after "nashorn.debug" system property is set to true. Also, this feature is available only on jdk 8u40 + and jdk 9 only.