Java Platform, Standard Edition

Nashorn User's Guide

Release 13
F18357-01
September 2019

Java Platform, Standard Edition Nashorn User's Guide, Release 13

F18357-01

Contents

Preface

This document describes the use of the Nashorn engine for Java application development with scripting features provided by the Java Platform, Standard Edition (Java SE).

Audience

This document is intended for Java programmers who want to use the Nashorn engine for scripting features provided by Java SE. It is assumed that you are familiar with Java and a version of the ECMAScript language standard (JavaScript).

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

- Scripting Languages and Java

  in *Java Platform, Standard Edition Java Scripting Programmer's Guide*
- The Java Scripting API specification (the `javax.script` package)

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction to the Nashorn Engine

This section provides introductory information about the Nashorn engine and how it can be used to interpret JavaScript code in a Java application or from the command line.

> **Note:**
>
> The Nashorn engine, the `jjs` tool, and the modules `jdk.scripting.nashorn` and `jdk.scripting.nashorn.shell` are deprecated in JDK 11 in preparation for removal in a future release.

The Nashorn engine is an implementation of the ECMAScript Edition 5.1 Language Specification. It also implements many new features introduced in ECMAScript 6 including template strings; `let`, `const`, and block scope; iterators and `for..of` loops; `Map`, `Set`, `WeakMap`, and `WeakSet` data types; symbols; and binary and octal literals.

It was fully developed in the Java language as part of the Nashorn Project. The code is based on the new features of the Da Vinci Machine, which is the reference implementation of Java Specification Request (JSR) 292: Supporting Dynamically Typed Languages on the Java Platform.

The Nashorn engine is included in the Java SE Development Kit (JDK). You can invoke Nashorn from a Java application using the Java Scripting API to interpret embedded scripts, or you can pass the script to the `jjs` or `jrunscript` tool.

> **Note:**
>
> Nashorn is the only JavaScript engine included in the JDK. However, you can use any script engine compliant with JSR 223: Scripting for the Java Platform or implement your own; see Scripting Languages and Java in *Java Platform, Standard Edition Java Scripting Programmer's Guide*.

Topics

Invoking Nashorn from Java Code

To invoke Nashorn in your Java application, create an instance of the Nashorn engine using the Java Scripting API.

To get an instance of the Nashorn engine:

1. Import the `javax.script` package.

   The Java Scripting API is composed of classes and interfaces in this package.

2. Create a `ScriptEngineManager` object.

   The `ScriptEngineManager` class is the starting point for the Java Scripting API. A `ScriptEngineManager` object is used to instantiate `ScriptEngine` objects and maintain global variable values shared by them.

3. Get a `ScriptEngine` object from the manager using the `getEngineByName()` method.

   This method takes one `String` argument with the name of the script engine. To get an instance of the Nashorn engine, pass in `"nashorn"`. Alternatively, you can use any of the following: `"Nashorn"`, `"javascript"`, `"JavaScript"`, `"js"`, `"JS"`, `"ecmascript"`, `"ECMAScript"`.

After you have the Nashorn engine instance, you can use it to evaluate statements and script files, set variables, and so on. The following example, `EvalScript.java`, provides simple Java application code that evaluates a `print("Hello, World!");` statement using Nashorn.

```
import javax.script.*;

public class EvalScript {
    public static void main(String[] args) throws Exception {
        // create a script engine manager
        ScriptEngineManager factory = new ScriptEngineManager();
        // create a Nashorn script engine
        ScriptEngine engine = factory.getEngineByName("nashorn");
        // evaluate JavaScript statement
        try {
            engine.eval("print('Hello, World!');");
        } catch (final ScriptException se) { se.printStackTrace(); }
    }
}
```

> **Note:**
>
> The `eval()` method throws a `ScriptException` that must be handled properly.

Java Scripting API Examples with Java Classes in *Java Platform, Standard Edition Java Scripting Programmer's Guide* demonstrates how to use scripts in Java code.

Invoking Nashorn from the Command Line

There are two command-line tools that can be used to invoke the Nashorn engine:

- `jrunscript`: This is a generic command that invokes any available script engine compliant with JSR 223. By default, without any options, `jrunscript` invokes the Nashorn engine, because it is the default script engine in the JDK.

- `jjs`: This is the recommended tool, created specifically for Nashorn. To evaluate a script file using Nashorn, pass the name of the script file to the `jjs` tool. To launch an interactive shell that interprets statements passed in using standard input, start the `jjs` tool without specifying any script files.

Nashorn Parser API

The Nashorn parser API enables applications, in particular IDEs and server-side frameworks, to parse and analyze ECMAScript code.

Parse ECMAScript code from a string, URL, or file with methods from the `Parser` class. These methods return an instance of `CompilationUnitTree`, which represents ECMAScript code as an abstract syntax tree. The package `jdk.nashorn.api.tree` contains the Nashorn parser API.

2
The Nashorn Java API

This section describes how to access Java from a script interpreted by the Nashorn engine.

> **Note:**
>
> The Nashorn engine, the `jjs` tool, and the modules `jdk.scripting.nashorn` and `jdk.scripting.nashorn.shell` are deprecated in JDK 11 in preparation for removal in a future release.

This section contains examples of script statements interpreted by the Nashorn engine in interactive language shell mode. This interactive shell is started by running the `jjs` command without any scripts passed to it. This is useful for trying things out, but the main purpose of the Nashorn Java API is to write Java applications as scripts that can be interpreted by the Nashorn engine.

Topics

Accessing Java Classes

There are two approaches to access packages and classes using Nashorn: the traditional approach is to use the `Packages` global object, and the recommended approach is to use the `Java` global object. This section describes both approaches.

The predefined top-level `Packages` object enables you to access Java packages and classes using their fully qualified names, as if they are properties of the `Packages`

object. The following example shows how you can access the `MyPackage` package and its `MyClass` class if `MyPackage.jar` is in your class path:

```
jjs> Packages.MyPackage
[JavaPackage MyPackage]
jjs> Packages.MyPackage.MyClass
[JavaClass MyPackage.MyClass]
```

Accessing standard Java packages and classes is more straightforward than accessing custom packages and classes. For your convenience, there are global objects defined for each of the standard Java packages: `com`, `edu`, `java`, `javax`, and `org`. They have aliases that correspond to properties of the `Packages` object. The following example shows how you can access the `java.lang` package and the `java.lang.System` class:

```
jjs> java.lang
[JavaPackage java.lang]
jjs> typeof java.lang
object
jjs> java.lang.System
[JavaClass java.lang.System]
jjs> typeof java.lang.System
function
```

As you can see from the previous example, Nashorn interprets Java packages as `JavaPackage` objects, and Java classes as `JavaClass` function objects, which can be used as constructors for the classes. Creating Java Objects describes how to instantiate a class.

The traditional approach for accessing Java packages and classes is intuitive and straightforward, but at the same time, it can be inefficient, limited, and error-prone for the following reasons:

- Each property access has a cost, so accessing a package or class in a deep hierarchy can be slow.
- There is no special syntax for creating Java arrays. You must use the `java.lang.reflect.Array` class as a workaround.
- If you misspell a class name, Nashorn assumes that you provided a package name, and interprets it as a `JavaPackage` object instead of a `JavaClass` function object. You might not be aware of this until an error is thrown when you attempt to use it as a class. To avoid this, use the `typeof` operator to conditionally test that the construct you are trying to access is interpreted as a function object. The following example shows how this conditional check works:

  ```
  jjs> typeof java.lang.System == "function"
  true
  jjs> typeof java.lang.Zyztem == "function"
  false
  ```

To avoid the disadvantages of the approach previously described, Nashorn defines the `Java` global object that has several functions for working with Java classes. The `Java.type()` function takes a string with the fully qualified Java class name, and

returns the corresponding `JavaClass` function object. The following example shows how you can access the `java.lang.System` class:

```
jjs> Java.type("java.lang.System")
[JavaClass java.lang.System]
```

Similar to importing classes in Java, it is a good practice to declare variables of `JavaClass` type at the beginning of a script. The following example shows how you can declare the `System` variable and give it a value of the `java.lang.System` class:

```
jjs> var System = Java.type("java.lang.System")
jjs> System
[JavaClass java.lang.System]
```

Creating Java Objects

To instantiate a class, pass the `JavaClass` function object to the `new` operator. Nashorn invokes the corresponding constructor based on the arguments passed to the function.

The following example shows how you can instantiate the `java.util.HashMap` class with the default initial capacity and with the initial capacity set to 100:

```
jjs> var HashMap = Java.type("java.util.HashMap")
jjs> var mapDef = new HashMap()
jjs> var map100 = new HashMap(100)
```

Accessing Class and Instance Members

You can use the standard dot notation to access static fields, methods, and inner classes as follows.

```
jjs> Java.type("java.lang.Math").PI
3.141592653589793
jjs> Java.type("java.lang.System").currentTimeMillis()
1375813353330
jjs> Java.type("java.util.Map").Entry
[JavaClass java.util.Map$Entry]
```

An inner class can also be accessed using internal representation with the dollar sign ($) as the separator, or a dot, which is consistent with Java:

```
jjs> Java.type("java.util.Map$Entry")
[JavaClass java.util.Map$Entry]
jjs> Java.type("java.util.Map.Entry")
[JavaClass java.util.Map$Entry]
```

To invoke an instance method or access an instance field of an object, use the dot operator, similar to how it is done in Java. The following example shows how you can call the `toUpperCase()` method on a `String` object:

```
jjs> var String = Java.type("java.lang.String")
jjs> var str = new String("Hello")
jjs> str
Hello
jjs> var upper = str.toUpperCase()
jjs> upper
HELLO
```

Nashorn also supports member access using the bracket notation, where you specify the name of the member as a string between brackets (`[]`) that immediately follow the class (in case of a static member) or object (in case of an instance member). This method is defined by the ECMAScript as an alternative to the dot notation, and is not intuitive for Java developers. However, it can be used to resolve method overload ambiguity. By default, Nashorn uses the overloaded method that best matches the arguments, and this is not always what you expect. For example, if you want to print a `double` value, you must use the `java.lang.System.out.println(double)` method overload, as shown in the following example:

```
jjs> Java.type("java.lang.System").out.println(10)
10
jjs> Java.type("java.lang.System").out["println(double)"](10)
10.0
```

Using JavaBeans

Nashorn enables you to treat accessor and mutator methods in JavaBeans as equivalent JavaScript properties. The name of the property is the name of the JavaBean method without the `get` or `set` suffix, and starts with a lowecase letter.

For example you can call the `getYear()` and `setYear()` methods in a `java.util.Date` object using the `year` property as follows:

```
jjs> var Date = Java.type("java.util.Date")
jjs> var date = new Date()
jjs> date.year + 1900
2013
jjs> date.year = 2014 - 1900
114
jjs> date.year + 1900
2014
```

Working with Java Arrays

To access a Java array class, pass to the `Java.type()` function the type of objects that comprise the array followed by a pair of brackets (similar to Java syntax).

The following example shows how you can access a Java array of integers and a Java array of `String` objects:

```
jjs> Java.type("int[]")
[JavaClass [I]
jjs> Java.type("java.lang.String[]")
[JavaClass [Ljava.lang.String;]
```

After you have the array type object, you can use it to instantiate an array as you do any other class. You can access array entries by their indexes, and use the dot or bracket notation to access members (similar to Java syntax), as shown in the following example:

```
jjs> var IntArrayType = Java.type("int[]")
jjs> var arr = new IntArrayType(10)
jjs> arr[1] = 123
123
jjs> arr[2] = 321
321
jjs> arr[1] + arr[2]
444
jjs> arr[10]
java.lang.ArrayIndexOutOfBoundsException: Array index out of range: 10
jjs> arr.length
10
```

If you have an existing JavaScript array, you can convert it to a Java array using the `Java.to()` function. The following example shows how you can convert a JavaScript array of strings `"a"`, `"b"`, and `"c"`, to a `java.lang.String[]` array with the same values:

```
jjs> var jsArr = ["a","b","c"]
jjs> var strArrType = Java.type("java.lang.String[]")
jjs> var javaArr = Java.to(jsArr, strArrType)
jjs> javaArr.class
class [Ljava.lang.String;
jjs> javaArr[0]
a
```

You can iterate through a Java array's indexes and values using the `for` and `for each` statements as follows:

```
jjs> for (var i in javaArr) print(i)
0
1
2
```

```
jjs> for each (var i in javaArr) print(i)
a
b
c
```

Working with Java Strings

Nashorn represents strings as `java.lang.String` objects. When you concatenate two strings, you get a `String` instance.

```
jjs> var a = "abc"
jjs> a.class
class java.lang.String
jjs> var b = a + "def"
jjs> b.class
class java.lang.String
jjs> var c = String(b)
jjs> c.class
class java.lang.String
```

Working with Java Numbers

Nashorn interprets numbers as `java.lang.Double`, `java.lang.Long`, or `java.lang.Integer` objects, depending on the computation performed. You can use the `Number()` function to force a number to be a `Double` object, as shown in the following example.

```
jjs> var intNum = 10
jjs> intNum.class
class java.lang.Integer
jjs> var dblNum = Number(intNum)
jjs> dblNum.class
class java.lang.Double
```

Working with Java Lists and Maps

Nashorn interprets Java lists as arrays; iterate over the values of a list with the `for each` statement. To iterate over keys and values in a map, use the `keySet()` and `values()` methods.

You can access list elements by using the index in brackets (`[]`) and iterate over the values of a list using the `for each` statement, as shown in the following example.

```
jjs> var ArrayList = Java.type("java.util.ArrayList")
jjs> var alist = new ArrayList()
jjs> alist.add("a")
true
jjs> alist.add("b")
true
jjs> alist.add("c")
true
```

```
jjs> alist[1]
b
jjs> for each (var i in alist) print(i)
a
b
c
```

> **Note:**
>
> You can only access list and map elements by using the index in brackets; in particular, you cannot access queue and set elements by using this syntax.

The following example shows how you can create a `HashMap` object and iterate over its keys and values with the `keySet()` and `values()` methods.

```
jjs> var HashMap = Java.type("java.util.HashMap")
jjs> var hm = new HashMap()
jjs> hm.put("name", "Bob")
jjs> hm.put("age", 40)
jjs> hm.put("weight", 180)
jjs> for each (var i in hm.keySet()) print(i)
weight
age
name
jjs> for each (var i in hm.values()) print(i)
180
40
Bob
```

Alternatively, you can iterate over the values of a map the same way you iterate over the values of a list, as shown in the following example:

```
jjs> for each (var i in hm) print(i)
180
40
Bob
```

Extending Java Classes

You can extend a class using the `Java.extend()` function that takes a Java type as the first argument and method implementations (in the form of JavaScript functions) as the other arguments.

The following script extends the `java.lang.Runnable` interface and uses it to construct a new `java.lang.Thread` object:

```
var Run = Java.type("java.lang.Runnable");
var MyRun = Java.extend(Run, {
    run: function() {
        print("Run in separate thread");
```

```
    }
});
var Thread = Java.type("java.lang.Thread");
var th = new Thread(new MyRun());
```

Nashorn can automatically extend functional interfaces (see the annotation type `FunctionalInterface`) if you provide the function for implementing the method as the argument to the constructor. The following script extends the `java.lang.Runnable` interface and uses it to construct a new `java.lang.Thread` object, but it uses fewer lines of code than in Example 2-1, because the `Java.extend()` function is called automatically for a functional interface.

```
var Thread = Java.type("java.lang.Thread")
var th = new Thread(function() print("Run in a separate thread"))
```

Extending Java Classes in *Java Platform, Standard Edition Java Scripting Programmer's Guide* describes the capabilities of the `Java.extend()` function.

Example 2-1 Extending a Java Class

```
var Run = Java.type("java.lang.Runnable");
var MyRun = Java.extend(Run, {
    run: function() {
        print("Run in separate thread");
    }
});
var Thread = Java.type("java.lang.Thread");
var th = new Thread(new MyRun());
```

Example 2-2 Extending a Functional Interface

```
var Thread = Java.type("java.lang.Thread")
var th = new Thread(function() print("Run in a separate thread"))
```

Restricting Script Access to Specified Java Classes

The `jdk.nashorn.api.scripting.ClassFilter` interface provides fine-grained control over access to Java classes from JavaScript code by restricting access to specified Java classes from scripts run by a Nashorn script engine.

Applications that embed Nashorn, in particular, server-side JavaScript frameworks, often have to run scripts from untrusted sources and therefore must limit access to Java APIs. These applications can implement the `ClassFilter` interface to restrict Java class access to a subset of Java classes. To do so, client applications must use Nashorn APIs to instantiate the Nashorn script engine.

> **Note:**
>
> While the `ClassFilter` interface can prevent access to Java classes, it is not enough to run untrusted scripts securely. The `ClassFilter` interface is not a replacement for a security manager. Applications should still run with a security manager before evaluating scripts from untrusted sources. Class filtering provides finer control beyond what a security manager provides. For example, an application that embeds Nashorn may prevent the spawning of threads from scripts or other resource-intensive operations that may be allowed by security manager.

Using Class Filters

To use a class filter, implement the `ClassFilter` interface and define the method `boolean exposeToScripts(String)`.

The `String` argument is the name of the Java class or package that the Nashorn script engine encounters when it runs a script. Define the method `exposeToScripts` such that it returns `false` for those classes and packages you want to prevent scripts from accessing. Then, create a Nashron script engine with your class filter with the method `NashornScriptEngineFactory.getScriptEngine(ClassFilter)` or `NashornScriptEngineFactory.getScriptEngine(String[], ClassLoader, ClassFilter)`.

Restricting Access to Java Reflection APIs

If you are using a security manager, then Nashorn allows a script to use the Java Reflection APIs (for example, the `java.lang.reflect` and `java.lang.invoke` packages) only if the script has the `nashorn.javaReflection` run time permission.

If you are using a class filter, then Nashorn prevents access to Java Reflection APIs even when a security manager is not present. Note that you do not need to use a class filter if the Java Reflection APIs are available because a script can use the `Class.forName(String)` to circumvent the class filter.

Example of Using the ClassFilter Interface

This example, `MyClassFilterTest.java`, demonstrates the `ClassFilter` interface. It restricts scripts' access to the class `java.io.File`.

```
import javax.script.ScriptEngine;
import jdk.nashorn.api.scripting.ClassFilter;
import jdk.nashorn.api.scripting.NashornScriptEngineFactory;

public class MyClassFilterTest {

  class MyCF implements ClassFilter {
    @Override
    public boolean exposeToScripts(String s) {
      if (s.compareTo("java.io.File") == 0) return false;
      return true;
    }
  }
```

```
  public void testClassFilter() {

    final String script =
      "print(java.lang.System.getProperty(\"java.home\"));" +
      "print(\"Create file variable\");" +
      "var File = Java.type(\"java.io.File\");";

    NashornScriptEngineFactory factory = new NashornScriptEngineFactory();

    ScriptEngine engine = factory.getScriptEngine(
      new MyClassFilterTest.MyCF());
    try {
      engine.eval(script);
    } catch (Exception e) {
      System.out.println("Exception caught: " + e.toString());
    }
  }

  public static void main(String[] args) {
    MyClassFilterTest myApp = new MyClassFilterTest();
    myApp.testClassFilter();
  }
}
```

This example prints the following:

```
C:\Java\jdk-11
Create file variable
Exception caught: java.lang.RuntimeException: java.lang.ClassNotFoundException:
java.io.File
```

The `MyClassFilterTest.java` example does the following:

1. Implements the `ClassFilter` with the inner class `MyCF` by defining the method `exposeToScripts`. In this example, the method `exposeToScripts` returns false if its `String` argument is `java.io.File`; otherwise it returns `true`.
2. Creates a Nashorn script engine, `engine`, with the method `NashornScriptEngineFactory.getScriptEngine(ClassFilter)`. The example invokes `getScriptEgnine` with an instance of `MyCF`.
3. Runs the following script with the Nashorn script engine `engine`:

   ```
   print(java.lang.System.getProperty(\"java.home\"));
   print(\"Create file variable\");
   var File = Java.type(\"java.io.File\");
   ```

   The class filter in `engine` checks each Java package and class in the script. When the class filter encounters the class `java.io.File`, the class filter returns `false`, and the `Java.type` function throws a `ClassNotFoundException`.

3
Nashorn and JavaFX

This section describes how to create and run JavaFX applications using scripts interpreted by the Nashorn engine. It is assumed that you are familiar with JavaFX.

> **Note:**
>
> The Nashorn engine, the `jjs` tool, and the modules `jdk.scripting.nashorn` and `jdk.scripting.nashorn.shell` are deprecated in JDK 11 in preparation for removal in a future release.

Topics

Interpreting JavaFX Script Application with Nashorn

You can interpret a JavaFX script application with Nashorn using the `jjs` command with the `-fx` option. For example, the following command invokes Nashorn to interpret the `JavaFXscript.js` file:

```
jjs -fx --module-path /SOMEDIR/javafx-sdk-11/lib --add-modules
javafx.controls JavaFXscript.js
```

> **Note:**
>
> - You must explicitly add the JavaFX modules to launch a script as a JavaFX application. If you don't add them, then the `jjs` command prints a message and exits:
>
>   ```
>   jjs -fx JavaFXscript.js
>   ```
>
>   ```
>   JavaFX is not available.
>   ```
> - JavaFX modules include `javafx.base`, `javafx.controls`, `javafx.fxml`, `javafx.graphics`, `javafx.media`, `javafx.swing`, and `javafx.web`. Ensure that you specify all the modules your application requires in the `--add-modules` option.
> - Obtain JavaFX from OpenJFX.

A JavaFX script application is similar to the Java equivalent, but Nashorn enables you to simplify many of the JavaFX constructs. Typically, a JavaFX script application

contains only the `start()` function, which is equivalent to the `start()` method in its Java counterpart. It can also contain the `init()` and `stop()` functions.

Using Nashorn to Simplify JavaFX Script Applications

Nashorn enables you to simplify your JavaFX code by not requiring you to declare certain types and enabling you to access certain methods and properties more easily.

By analyzing the examples in Example of a JavaFX Application (HelloWorld.java) and Example of a JavaFX Script Application (HelloWorld.js), you can see how Nashorn enables you to simplify Java code when you write a JavaFX application as a script:

- There is no need to declare variable types, import packages, use annotations, specify the class name, and implement its `main()` method.
- Only the JavaFX classes that are instantiated must be declared.
- JavaBeans do not require the `get` and `set` prefixes, and are treated as JavaScript properties instead of as Java methods. See Using JavaBeans.
- Implementing the `javafx.event.EventHandler` interface does not require you to specify the implemented method explicitly. Because `handle()` is the only method, Nashorn automatically applies the provided function to the method. See Extending Java Classes.

The JavaFX primary stage is available to Nashorn as a global property `$STAGE`. This global property enables you to treat the whole script as one `start()` function (you can still add the `init()` and `stop()` functions). Example of a Simpler Version of JavaFX Script Application (HelloWorldSimple.js) contains the source code for a simplified version of the JavaFX script application from Example of a JavaFX Script Application (HelloWorld.js).

Example of a JavaFX Application (HelloWorld.java)

The following example, `HelloWorld.java`, contains the source code for a simple JavaFX application that displays a button, which when clicked prints `"Hello World!"` to standard output.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
```

```
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

Example of a JavaFX Script Application (HelloWorld.js)

The following example, `HelloWorld.js`, contains the source code for a JavaFX script application that displays a button, which when clicked prints `"Hello World!"` to standard output.

```
var Button = javafx.scene.control.Button;
var StackPane = javafx.scene.layout.StackPane;
var Scene = javafx.scene.Scene;

function start(primaryStage) {
    primaryStage.title = "Hello World!";
    var button = new Button();
    button.text = "Say 'Hello World'";
    button.onAction = function() print("Hello World!");
    var root = new StackPane();
    root.children.add(button);
    primaryStage.scene = new Scene(root, 300, 250);
    primaryStage.show();
}
```

Example of a Simpler Version of JavaFX Script Application (HelloWorldSimple.js)

The following example, `HelloWorldSimple.js`, contains the source code for a simplified version of the JavaFX script application using the global property `$STAGE`.

```
var Button = javafx.scene.control.Button;
var StackPane = javafx.scene.layout.StackPane;
var Scene = javafx.scene.Scene;

$STAGE.title = "Hello World!";
var button = new Button();
button.text = "Say 'Hello World'";
button.onAction = function() print("Hello World!");
var root = new StackPane();
root.children.add(button);
$STAGE.scene = new Scene(root, 300, 250);
$STAGE.show();
```

Nashorn Script Objects

In most cases, you should only add the classes that you instantiate or use to access static fields. However, for prototyping purposes, Nashorn predefines a set of scripts

that can be loaded to import groups of JavaFX packages and classes. You can load a script using the `load()` function that takes a string with the name of the script.

The following table lists the predefined script objects that are available for inclusion:

This script ...	Imports ...
fx:base.js	javafx.stage.Stage javafx.scene.Scene javafx.scene.Group javafx/beans javafx/collections javafx/events javafx/util
fx:graphics.js	javafx/animation javafx/application javafx/concurrent javafx/css javafx/geometry javafx/print javafx/scene javafx/stage
fx:controls.js	javafx/scene/chart javafx/scene/control
fx:fxml.js	javafx/fxml
fx:web.js	javafx/scene/web
fx:media.js	javafx/scene/media
fx:swing.js	javafx/embed/swing
fx:swt.js	javafx/embed/swt

Example of a JavaFX Script Application with Loaded Scripts contains the source code of the simplified JavaFX script application from Example of a Simpler Version of JavaFX Script Application (HelloWorldSimple.js) with `load()` functions used to import the necessary packages and classes.

Example of a JavaFX Script Application with Loaded Scripts

The following example contains the source code for a simplified version of the JavaFX script application using the `load()` functions to import packages and classes:

```
load("fx:base.js");
load("fx:controls.js");
load("fx:graphics.js");

$STAGE.title = "Hello World!";
var button = new Button();
button.text = "Say 'Hello World'";
button.onAction = function() print("Hello World!");
var root = new StackPane();
root.children.add(button);
$STAGE.scene = new Scene(root, 300, 250);
$STAGE.show();
```

For more examples of JavaFX script applications, see JavaFX Script Application Examples.

4
Nashorn and Shell Scripting

This section describes the extensions of the Nashorn engine that enable you to use UNIX shell scripting features for JavaScript scripts.

> **Note:**
>
> The Nashorn engine, the `jjs` tool, and the modules `jdk.scripting.nashorn` and `jdk.scripting.nashorn.shell` are deprecated in JDK 11 in preparation for removal in a future release.

You can enable shell scripting extensions in Nashorn using the `jjs` command with the `-scripting` option. For example, the following command invokes Nashorn in interactive mode with shell scripting extensions enabled:

```
jjs -scripting
```

In addition to the standard JavaScript comments (`//` and `/* */`), Nashorn supports shell-style comments using the number sign (#). If the number sign is the first character in a script, then the Nashorn shell scripting extensions are automatically enabled when you interpret the script, even if you use the `jjs` tool without the `-scripting` option. This is useful when specifying the shebang (`#!`) at the beginning of a script to run it as a shell executable. See Shebang.

Topics

Shebang

You can use the shebang (`#!`) at the beginning of a script file to enable the script file to run as a shell executable. If you specify the path to the `jjs` tool in the shebang, then when you execute the script, the shell runs the `jjs` tool instead, and passes the script file to it.

The `jjs` tool is located in the `JAVA_HOME/bin` directory, where `JAVA_HOME` is the installation directory of the JDK. When you install the JDK, the `JAVA_HOME` environment variable is usually set up automatically. If it was not set up automatically, or if you have several versions of the JDK installed, set the `JAVA_HOME` environment variable to the correct path of the JDK installation directory manually.

You can specify the direct path to the `jjs` tool in the shebang, but it is a good practice to create a symbolic link in the `/usr/bin` directory as follows:

```
>> cd /usr/bin
>> ln -s $JAVA_HOME/bin/jjs jjs
>>
```

> **Note:**
>
> You might have to run the `ln` command with root privileges using `sudo`.

After you set up the symbolic link, you can create Nashorn scripts that can be run as executables. Also, it is possible to add command-line options directly to the shebang statement. The following example, `scriptArgs.js`, shows an executable script that prints the version of the Nashorn engine and then the arguments passed to the script.

```
#!/usr/bin/jjs -fv
print("Arguments: " + arguments);
```

If you run the `scriptArgs.js` file as a shell executable, it is interpreted by the Nashorn engine as follows:

```
>> ./scriptArgs.js -- arg1 arg2 arg3
nashorn full version 1.8.0
Arguments: arg1,arg2,arg3
>>
```

Alternatively, if the path to the `jjs` tool is in the `PATH` environment variable, you can point the shebang to `jjs` as follows:

```
#!/usr/bin/env jjs
```

String Interpolation

String interpolation is used in UNIX shells to construct strings that contain values of variables or expressions. With shell scripting features enabled, Nashorn enables you to embed variables and expressions into string literals in the same way.

For example, you can assign the result of the `Date()` constructor to the `date` variable, and then pass this variable to the string using the dollar sign ($) and braces ({}) as follows:

```
jjs> var date = Date()
jjs> "Date and time: ${date}"
Date and time: Mon Aug 19 2013 19:43:08 GMT+0400 (MSK)
```

The preceding example displays the date and time when the `date` variable was assigned the value returned by the `Date()` constructor. If you want to display the

current date and time when the expression is evaluated, you can pass the `Date()` constructor directly as follows:

```
jjs> "Current date and time: ${Date()}"
Current date and time: Mon Aug 19 2013 19:49:53 GMT+0400 (MSK)
```

String interpolation works only for strings within double quotation marks. Strings within single quotation marks are not interpolated:

```
jjs> 'The variable is ${date}'
The variable is ${date}
```

Here Document

A here document (*heredoc*) specifies strings in UNIX shells that preserve line breaks and indentations. With shell scripting features enabled, you can use Nashorn to evaluate scripts with heredocs.

The following example, `scriptHereArgs.js`, shows an executable script that prints the first passed-in argument on the first line, then the second argument indented on the second line, and the third argument on the forth line (after one blank line).

```
#!/usr/bin/jjs
print(<<EOD);
${arguments[0]} is normal
    ${arguments[1]} is indented

${arguments[2]} is separated by a blank line
EOD
```

If you run the `scriptHereArgs.js` file as a shell executable, it is interpreted by the Nashorn engine as follows:

```
>> ./scriptHereArgs.js -- Line1 Line2 Line3
Line1 is normal
    Line2 is indented

Line3 is separated by a blank line
```

Global Objects

With shell scripting features enabled, Nashorn defines several global objects.

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

```
>> jjs -scripting -- arg1 arg2 arg3
jjs> $ARG
arg1,arg2,arg3
```

```
jjs> $ARG[1]
arg2
```

$ENV
This global object maps all the current environment variables, for example:

```
jjs> $ENV.USER
johndoe
jjs> $ENV.PWD
/foo/bar
```

$EXEC()
This global function launches processes to run commands, for 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>
```

The `$EXEC()` function can also take a second argument, which is a string to be used as standard input (`stdin`) for the process:

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

> **Note:**
>
> 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` ``.

$OPTIONS
This object exposes command line options passed to nashorn "command line". For example:

```
jjs> print("-scripting=" + $OPTIONS_scripting);
jjs> print("--compile-only=" + $OPTIONS_compile_only);
jjs> print("-timezone="+ $OPTIONS_timezone.ID);
```

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

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

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

Additional Nashorn Built-in Functions

Nashorn defines several built-in functions: `echo`, `readLine` and `readFully` functions are defined only for `-scripting` mode. Other extensions like `quit`, `exit`, `load`, `loadWithNewGlobal`, `Object.bindProperties` are always available.

quit()
exit()
These functions are synonymous, causing the current script process to exit to the system. You can pass an integer value as the argument that represents the exit code to be returned to the system. By default, without an argument, the exit code is set to 0, meaning that the process terminated correctly.

print()
echo()
These functions are synonymous, causing the values passed in as arguments to be converted to strings, printed to `stdout` separated by spaces, and followed by a new line. The implementation involves calls to `java.lang.System.out.print(string)` followed by `java.lang.System.out.println()`.

```
>> jjs -scripting -- arg1
jjs> var a = "Hello"
jjs> print(123, $ARG[0], a, "World")
123 arg1 Hello World
jjs>
```

readLine()
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:

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

readFully()
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.

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

jjs>
```

The `readFully()` function reads data with the method `byte[] Files.readAllBytes(Path)`, then performs byte order mark (BOM) detection on the returned byte array to determine whether the data read is Unicode-encoded.

`load()`
This function loads and evaluates a script from a path, URL, or script object.

```
jjs> load("/foo/bar/script.js")
jjs> load("http://example.com/script.js")
jjs> load({name:"script.js", script:"var x = 1 + 1; x;"})
```

`loadWithNewGlobal()`
This function is similar to the `load()` function, but the script is evaluated with a new global object. This is the primary method for creating a fresh context for evaluating scripts. Additional arguments (after the script) passed to `loadWithNewGlobal()` are stored in the `arguments` global variable of the new context.

`Object.bindProperties(`*`target`*`, `*`source`*`)`
This function is used to bind the *`source`* object's properties to the object *`target`*. The function enables sharing of global properties. For example, in a Document Object Model (DOM) simulation, you can share properties between the global object and the document. In a multithreading application, you can share functions across global objects of threads.
The following example shows how you can bind the `obj` object's properties to the global object:

```
jjs> var obj = {x:34,y:100}
jjs> obj.x
34
jjs> obj.y
100
jjs> x
<shell>:1 ReferenceError: "x" is not defined
jjs> Object.bindProperties(this,obj)
[object global]
jjs> x
34
jjs> y = Math.PI
3.141592653589793
jjs> obj.y
3.141592653589793
jjs>
```

Note that the function `Object.bindProperties(target, source)` only binds properties of *source* that do not exist in *target*. In the following example, the function `bindProperties` tries to bind the properties of the `obj` object to the global object. However, the `x` property already exists in the global object. Therefore, the function `bindProperties` does not bind `obj.x` to global `x`. Thus, changing the value of `obj.x` does not change the value of global `x`.

```
jjs> x = 2
2
```

```
jjs> var obj = {x:3}
jjs> Object.bindProperties(this,obj)
[object global]
jjs> x
2
jjs> x = 4
4
jjs> obj.x
3
jjs> x
4
```

A

JavaFX Script Application Examples

This section provides examples of JavaFX script applications.

> **Note:**
>
> The Nashorn engine, the `jjs` tool, and the modules `jdk.scripting.nashorn` and `jdk.scripting.nashorn.shell` are deprecated in JDK 11 in preparation for removal in a future release.

Nashorn and JavaFX describes how to write JavaFX applications as scripts interpreted by the Nashorn engine.

Topics

Example JavaFX 3-D

This example demonstrates how a complex task (in this case, rendering 3-D graphics with JavaFX) can be done with relatively little code using the Nashorn script engine capabilities. When you run this example, a window with three green shapes is shown: a box, a sphere, and a cylinder.

```
load("fx:base.js");
load("fx:controls.js");
load("fx:graphics.js");

var material = new PhongMaterial();
material.diffuseColor = Color.LIGHTGREEN;
material.specularColor = Color.rgb(30, 30, 30);

var meshView = Java.to([
    new Box(200, 200, 200),
    new Sphere(100),
    new Cylinder(100, 200)
], "javafx.scene.shape.Shape3D[]");

for (var i = 0; i != 3; i++) {
    meshView[i].material = material;
    meshView[i].translateX = (i + 1) * 220;
    meshView[i].translateY = 200;
    meshView[i].translateZ = 20;
    meshView[i].drawMode = DrawMode.FILL;
    meshView[i].cullFace = CullFace.BACK;
```

```
};

var pointLight = new PointLight(Color.WHITE);
pointLight.translateX = 800;
pointLight.translateY = -200;
pointLight.translateZ = -1000;

var root = new Group(meshView);
root.children.add(pointLight);

var scene = new Scene(root, 800, 400, true);
scene.fill = Color.rgb(127, 127, 127);
scene.camera = new PerspectiveCamera(false);
$STAGE.scene = scene;
$STAGE.show();
```

Example JavaFX Animation

This example demonstrates how to create a JavaFX animation using the Nashorn script engine capabilities. When you run this example, a window with colorful circles moving around is shown.

```
load("fx:base.js");
load("fx:controls.js");
load("fx:graphics.js");

var WIDTH = 500;
var HEIGHT = 600;
var animation;

function setup(primaryStage) {
 var root = new Group();
 primaryStage.resizable = false;
 var scene = new Scene(root, WIDTH, HEIGHT);
 scene.title = "Colourful Circles";
 primaryStage.scene = scene;

 // create first list of circles
 var layer1 = new Group();
 for(var i = 0; i < 15; i++) {
  var circle = new Circle(200, Color.web("white", 0.05));
  circle.strokeType = StrokeType.OUTSIDE;
  circle.stroke = Color.web("white", 0.2);
  circle.strokeWidth = 4;
  layer1.children.add(circle);
 }

 // create second list of circles
 var layer2 = new Group();
 for(var i = 0; i < 20; i++) {
  var circle = new Circle(70, Color.web("white", 0.05));
  circle.strokeType = StrokeType.OUTSIDE;
  circle.stroke = Color.web("white", 0.1);
  circle.strokeWidth = 2;
  layer2.children.add(circle);
 }

 // create third list of circles
```

```
var layer3 = new Group();
for(var i = 0; i < 10; i++) {
 var circle = new Circle(150, Color.web("white", 0.05));
 circle.strokeType = StrokeType.OUTSIDE;
 circle.stroke = Color.web("white", 0.16);
 circle.strokeWidth = 4;
 layer3.children.add(circle);
}

// set a blur effect on each layer
layer1.effect = new BoxBlur(30, 30, 3);
layer2.effect = new BoxBlur(2, 2, 2);
layer3.effect = new BoxBlur(10, 10, 3);

// create a rectangle size of window with colored gradient
var colors = new Rectangle(WIDTH, HEIGHT,
  new LinearGradient(0, 1, 1, 0, true, CycleMethod.NO_CYCLE,
     new Stop(0,   Color.web("#f8bd55")),
     new Stop(0.14, Color.web("#c0fe56")),
     new Stop(0.28, Color.web("#5dfbc1")),
     new Stop(0.43, Color.web("#64c2f8")),
     new Stop(0.57, Color.web("#be4af7")),
     new Stop(0.71, Color.web("#ed5fc2")),
     new Stop(0.85, Color.web("#ef504c")),
     new Stop(1,   Color.web("#f2660f"))));
colors.blendMode = BlendMode.OVERLAY;

// create main content
var group = new Group(new Rectangle(WIDTH, HEIGHT, Color.BLACK),
       layer1,
       layer2,
       layer3,
       colors);
var clip = new Rectangle(WIDTH, HEIGHT);
clip.smooth = false;
group.clip = clip;
root.children.add(group);

// create list of all circles
var allCircles = new java.util.ArrayList();
allCircles.addAll(layer1.children);
allCircles.addAll(layer2.children);
allCircles.addAll(layer3.children);

// Create a animation to randomly move every circle in allCircles
animation = new Timeline();
for each (var circle in allCircles) {
 animation.getKeyFrames().addAll(
    new KeyFrame(Duration.ZERO, // set start position at 0s
        new KeyValue(circle.translateXProperty(),
Math.random() * WIDTH),
        new KeyValue(circle.translateYProperty(),
Math.random() * HEIGHT)),
     new KeyFrame(new Duration(20000), // set end position at 20s
        new KeyValue(circle.translateXProperty(),
Math.random() * WIDTH),
        new KeyValue(circle.translateYProperty(),
Math.random() * HEIGHT))
     );
 }
 animation.autoReverse = true;
```

```
 animation.cycleCount = Animation.INDEFINITE;
}

function stop() {
 animation.stop();
}

function play() {
 animation.play();
}

function start(primaryStage) {
 setup(primaryStage);
 primaryStage.show();
 play();
}
```