

# Rhino Migration Guide

## Rhino Migration Guide

jdk8 replaces Rhino based jsr-223 script engine with nashorn based jsr-223 script engine. If you are using Rhino based jsr-223 script engine in jdk 6 or jdk 7, you'll have some migration work when moving to jdk 8. This document is a migration guide towards that task.

### Nashorn extensions

Nashorn implements ECMAScript 5.1 specification with a number of syntax and API extensions as documented in [Nashorn extensions](#) Few of those are Rhino specific extensions as well. You may want to go through that document to check if a Rhino specific extension is already supported by nashorn.

### Accessing Java packages and classes from script

Nashorn supports top-level "**Packages**" object and "**java**", "**javax**" etc. as supported by Rhino. You can use Packages object to access Java packages and classes. But, Nashorn's recommended way to access Java classes is to use **Java.type**.

#### Packages vs Java.type

```
var Vector = java.util.Vector;
var JFrame = Packages.javax.swing.JFrame;

// or preferably

var Vector = Java.type("java.util.Vector")
var JFrame = Java.type("javax.swing.JFrame")
```

Java.type is recommended because

- 1) It avoid multiple step object.property resolution as done by Packages method. Class resolution is done in one step - from String name to class
- 2) Java.type throws ClassNotFoundException rather than silently treating an unresolved name to be package name!

java.util.vector results in a package object named "java.util.vector" whereas Java.type("java.util.vector") results in ClassNotFoundException.

### Creating Java arrays from script

In Rhino, you create a Java array using Java reflection from script. In Nashorn, you can resolve to a Java array class using the same Java.type API. And array creation is done using new operator

#### Creating Java Array

```
// Rhino way!
var Array = java.lang.reflect.Array
var intClass = java.lang.Integer.TYPE
var array = Array.newInstance(intClass, 8)

// Nashorn way!
var IntArray = Java.type("int[]")
var array = new IntArray(8)
```

Java array elements are accessed/modified using [] operator in both rhino as well as nashorn. Also special "length" property is supported both in rhino and nashorn.

### Class object and .class property

If a java API accepts a java.lang.Class object, in rhino you can pass script representation of class "as is". In Nashorn, you've to use ".class" property (similar to Java).

## Class object

```
// Rhino way!
var Array = java.lang.reflect.Array
var strArray = Array.newInstance(java.lang.String, 10)

// Nashorn way!
var Array = Java.type("java.lang.reflect.Array")
var JString = Java.type("java.lang.String")

// note ".class" property access to get java.lang.Class object
var strArray = Array.newInstance(JString.class, 10)
```

In the above example, better way to create Java string array from Nashorn would be to get `String[]` type from Nashorn using `Java.type`. The example is written this way only to demonstrate `".class"` property.

## \_\_proto\_\_ magic property

Rhino's magic writable property `__proto__` to read/write prototype of an object is also supported by nashorn for compatibility. But nashorn recommended way to read/write is `Object.getPrototypeOf` (<http://es5.github.io/#x15.2.3.2>) and `Object.setPrototypeOf` ([https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Object/setPrototypeOf](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf)) APIs. `__proto__`, while supported by nashorn, is deprecated.

## JavalImporter and with

Nashorn supports `JavalImporter` constructor of Rhino. It is possible to locally import multiple java packages and use it within a `'with'` statement.

## Java exceptions

Rhino wraps Java exceptions as a script object. If you want underlying Java exception, you've to use `"javaException"` property to access it. Nashorn does not wrap Java exceptions. Java exception objects as thrown "as is". So, in catch blocks you can access Java exceptions "as is".

## Java exceptions

```
// rhino
try {
  java.lang.System.loadLibrary(null)
} catch (e) {
  // false!
  print(e instanceof java.lang.NullPointerException)
  // true
  print(e.javaException instanceof java.lang.NullPointerException)
}

// in Nashorn, e instanceof java.lang.NullPointerException is true
// as there is no script wrapping of exceptions.
```

Also, no Java object is wrapped as "script object" in Nashorn (unlike Java).

## Implementing Java interface

Both Rhino and Nashorn support java anonymous class-like syntax to implement java interfaces in script.

## Java interface

```
// Works both in rhino and nashorn.
var runnable = new java.lang Runnable() {
  run: function() {
    java.lang.System.out.println("I am run!");
  }
};
```

The example @ <https://github.com/mozilla/rhino/blob/master/examples/enum.js> works on Nashorn as well.

## Extending Java class

To extend a concrete Java class or to implement multiple interfaces, you have to use `Java.extend` in Nashorn - unlike "JavaAdapter" in Rhino. `Java.extend` is explained in [Nashorn extensions](#) document.

## JavaScript vs Java Strings

Nashorn does not use wrapper objects to provide JavaScript access to Java objects like Rhino did. Since Nashorn uses `java.lang.String` to represent JavaScript strings internally it is not able to distinguish between native [JavaScript Strings](#) and host Java String objects, and both JavaScript and Java String methods can be invoked on any String object.

There happens to be a conflict in the case of the `replace` method which is defined in both languages with different semantics. In this case, the JavaScript method has precedence over the Java method. One could use [explicit method selection](#) to invoke the Java method but it is usually simpler to just use the JavaScript method.

## Compatibility script

There are few Rhino/Mozilla extensions that are supported only if you load the compatibility script provided by nashorn. The compatibility script is loaded using `load("nashorn:mozilla_compat.js")`

### Mozilla Compatibility Script

```
// load compatibility script

load("nashorn:mozilla_compat.js");
```

The compatibility script implements the following Rhino extensions:

- **importClass** global function to import a specific Java class. Recommended alternative is to use `Java.type` and assign the result to global variable
- **importPackage** global function to import a specific Java package. Recommended alternative is to use `JavaImporter` and `with` statement.
- **JavaAdapter** global function to subclass java class or implement java interfaces (this is a wrapper over `Java.extend` API of nashorn). Recommended alternative is to use `Java.extend` API directly.
- **\_\_defineGetter\_\_**, **\_\_defineSetter\_\_**, **\_\_lookupGetter\_\_** `Object.prototype` functions explained at <http://ejohn.org/blog/javascript-getters-and-setters/> Note that these are deprecated by Mozilla. Recommended alternative is to use ECMAScript compliant **Object.defineProperty** <http://es5.github.io/#x15.2.3.6>, **Object.defineProperties** <http://es5.github.io/#x15.2.3.7> API
- **toSource** method on number of builtins - for example, function object to get source code of the function
- A number of HTML generation String methods like "anchor", "sup" etc.

## Compatibility script examples

```
load("nashorn:mozilla_compat.js")

var obj = {}
obj.__defineGetter__(
  "name", function(){
    return "sundar"
  }
)
print(obj.name)
obj.__defineSetter__(
  "x", function(xVal) { print("x set to " + xVal); this._x = xVal }
)
obj.x = 434;

// import specific class
importClass(java.util.Vector)

var v = new Vector(3)
print(v)

// import package
importPackage(java.io)

print(new File(".").getAbsolutePath())

// Using JavaAdapter to extend a Java class
var myVector = new JavaAdapter(java.util.Vector) {
  size: function() {
    print("size called!");
    return 0;
  }
};

myVector.size();

// toSource function to get source code
print(print.toSource())
```

## loading compatibility script so that script runs on rhino and nashorn

```
// if you want the script run both on rhino and nashorn
try {
  load("nashorn:mozilla_compat.js");
} catch (e) {}

// Or you can check for importClass function and then load if missing ...
if (typeof importClass != "function") {
  load("nashorn:mozilla_compat.js");
}
```