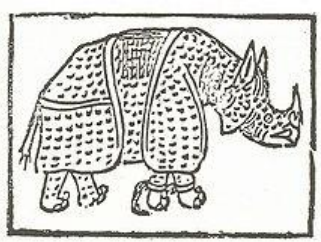
The Rhino Project

A Javascript Engine

INFO3 Logiciel - Mélissa Da Costa & Emilie Marti

Table of Contents

The History of The Rhino Project



Rhino is created by Netscape.

Great performance but heavy compilation time and memory leak issues.

Interpretative mode is released.

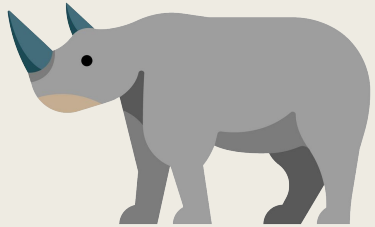
Rhino is released to the Mozilla Foundation.

Rhino becomes open-source.

Nashorn is created by Oracle.

It aims to replace Rhino as a Javascript Interpreter.

Rhino, a Javascript Runtime



- Open-source implementation of Javascript written in Java, that allows writing programs in Javascript that use the power of the Java platform APIs and Virtual Machine as its runtime environment.
- Rhino compile Javascript scripts to Java Bytecode which will run on a JVM.
- An interpretive mode is supported: the generated compiled code is represented as an object that can be garbage collected.
- Allows the use of Java libraries, also called « Scripting Java ».

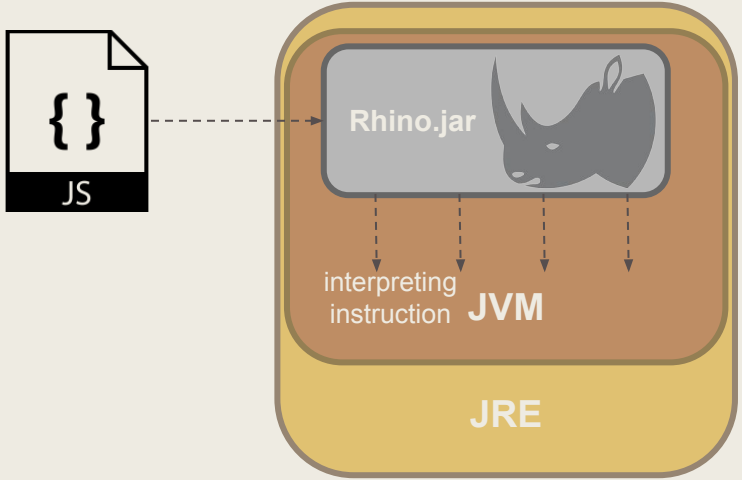
Features of Rhino

01	Javascript 1.7 features	• All features until JavaScript 2006
02	Javascript Shell	• Rhino shell can execute Javascript instructions and load and execute whole scripts.
03	Javascript Interpreter	• Rhino offers an interpretative mode where a Javascript script is interpreted by the Rhino engine.
04	Javascript Compiler	• The Rhino compiler will transform the Javascript script into bytecode directly understood by the JVM.
05	Java Scripting	• Rhino exposes Java API to be called within a Javascript script.

Rhino as a Javascript Interpreter

Rhino can take a Javascript script as a parameter and interpret each instruction at a time, at runtime.

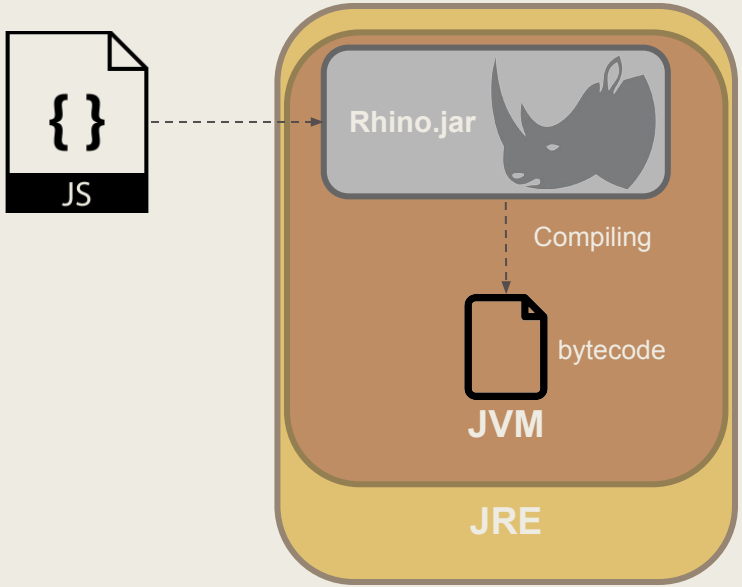
It doesn't generate class files from the script.



Rhino as a Javascript Compiler

Rhino can take a Javascript script and compile it into bytecode.

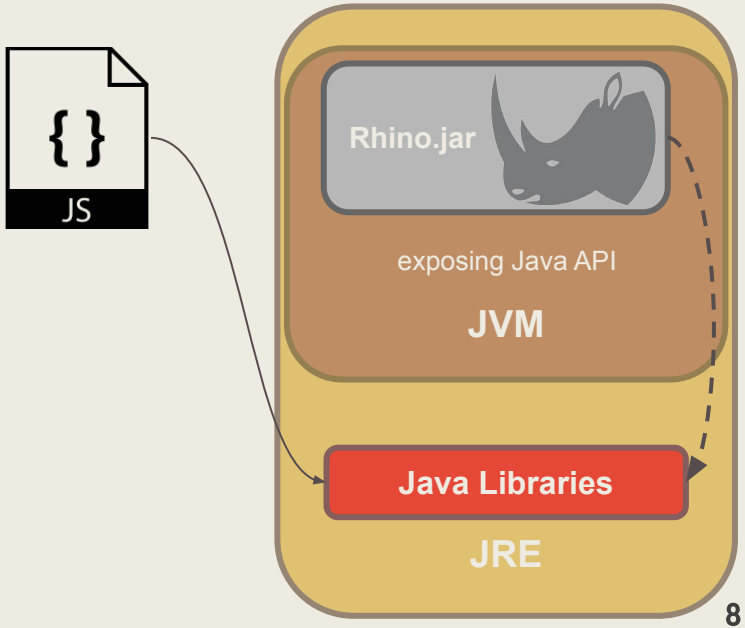
This bytecode will be directly interpreted by the JVM, thus offering better performances than interpretive mode.



Scripting Java with Rhino : Java API

Rhino allows to use the Java libraries through a Javascript script, thus benefiting from the power and efficiency of Java API and simplicity of JavaScript programming.

Java API is exposed by the top-level variable **Packages.java**.



Scripting Java with Rhino : Import Package

To use Java APIs, we use *importPackage(packageName)* to access the class of this package.

```
$  java -jar rhino1.7.13/buildGradle/libs/rhino1.7.13-1.7.13.jar
Rhino 1.7.13 2020 10 24
js> importPackage(java.io)
js> file = new File("foo.txt")
foo.txt
js>
js> importPackage(java.lang)
js> System.out.println("toto")
toto
```

Scripting Java with Rhino : Java Array

Rhino provides no special syntax for creating Java arrays, so we use the method *java.lang.reflect.Array.newIntance(...)*.

➔ String array of size 500:

```
js> stringArray = java.lang.reflect.Array.newInstance(java.lang.String, 500)
[Ljava.lang.String;@1ed4004b
```

In order to create an array of primitive types, we must use the *TYPE*.

➔ int array of size 500:

```
js> intArray = java.lang.reflect.Array.newInstance(java.lang.Integer.TYPE, 500)
[I@6ebc05a6
```

Scripting Java with Rhino : Java Interfaces Thread example

1. Define a JavaScript object (obj) with function properties whose names match the methods required by the Java interface (run);
2. Create an object (r) implementing the Runnable interface by constructing a Runnable;
3. Create a Thread and run it.

```
js> obj = {run:function(){print("thread running\n");}};
[object Object]
js> obj.run()
thread running

js>
js> runnable = new java.lang.Runnable(obj)
adapter1@52af6cff
js> thread = new java.lang.Thread(runnable);
Thread[Thread-0,5,main]
js> thread.start()
js> thread running
```

Rhino optimizations

Rhino, as other Javascript engines, offers some optimizations to improve the performances of a JavaScript application.

Scripts are optimized at compilation time, not at execution time, thus optimizations cannot be performed on interpretative mode.

Some optimization settings are provided by Rhino when executing the program in the form of levels: from -1 to 9.

Optimization level -1

DESCRIPTION

This is the default interpretive mode (cf. slide 8: Rhino as a Javascript Interpreter).

The compilation time is minimized at the expense of runtime performance.

FEATURES

No class files are generated, which may improve memory usage depending on your system.

The interpreter performs tail-call elimination of recursive functions.

Optimization level 0

DESCRIPTION

Rhino compiles the JavaScript script into bytecode in this mode. (cf. slide 8: Rhino as a Javascript compiler)

However, level 0 means that it performs no optimizations.

FEATURES

The bytecode runs faster than Javascript interpreted.

Generated code is less efficient.

Optimization levels 1-9

All local variables and parameters are allocated to JVM registers	• Fastens the access to the variables and parameters.
Simple data and flow analysis	• Determine which Javascript variables can be allocated to JVM registers. • Determine which variables are used only as Numbers.
Function call targets are speculatively pre-cached	• It allows the dispatching to be direct, pending runtime confirmation of the actual target.
Arguments are passed as Objects / Number pairs	• It reduces conversion overhead.
Local common sub-expressions are collapsed	• Only happens for property lookup. • Maybe more expressions in a future release.

Optimization levels 1-9 - Notes

- ➔ The level will determine how aggressive we want the optimizations to be: at level 9 all optimizations will be performed.
- ➔ Which optimizations are performed in each level isn’t specified.
- ➔ Future versions may allocate higher levels to offer better optimizations so it is advised to use level 1 in order to ensure future compatibility.
- ➔ Optimization level can be set within a script but it will be useless as optimizations are performed at compile time and not runtime.

Benchmarking Rhino

WHAT IS TESTED

We test Rhino performances on loops and variable access.

- ➔ Direct access: native type (int)
- ➔ Indirect access: functions that return a native type value (int)

HOW IS IT TESTED

Using Benchmark.js as a benchmarking library.

Loop performance is tested by iterating through the arrays.

Benchmark.js

- Benchmark.js is a benchmarking library running on most Javascript platforms.

- Dependencies : lodash.js and platform.js
  - Using the function load(‘path/to/script.js’) to import librairies on Rhino.

- Using the library:
  - Creation of a suite of Benchmarks
  - Addition of each Benchmarking test to the suite

Programming Javascript on Rhino

SIMULATING AN OBJECT

Rhino doesn't allow Javascript classes so the objects are simulated through functions:

```
function A(v) {
      return {value: () => v};
}
function B(v) {
     return {value: () => v};
}
```

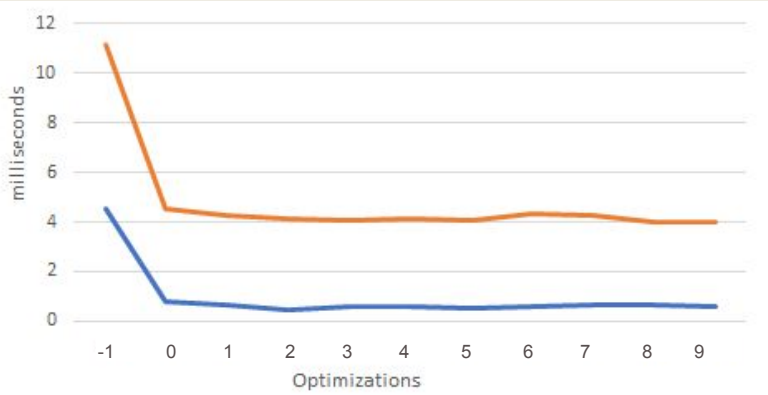
CREATING AN ARRAY

The array can be constructed using the array constructor:

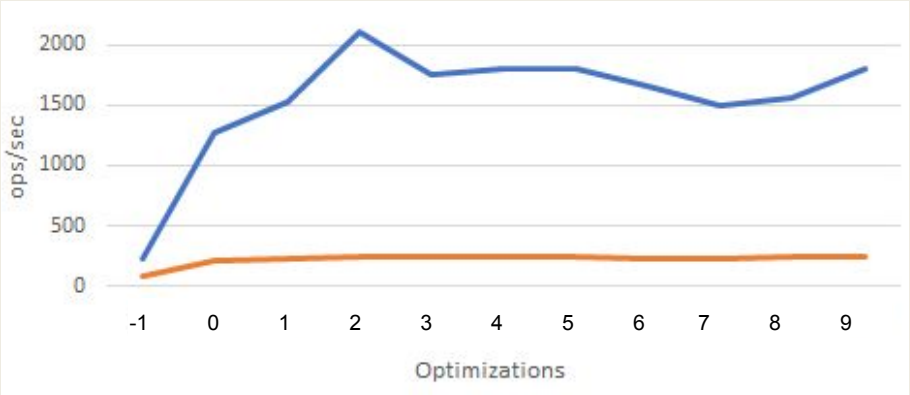
```
OBJ_ARRAY = new Array(ARRAY_SIZE)
   .fill(null).map((_, i) => {
   let random = Math.floor(Math.random() * MAX_VALUE)
   if (i % 2 === 0)
      return new A(random)
   else
      return new B(random)
});
```

Benchmarks with Benchmark.js : Results

EXECUTION TIME



OPERATIONS PER SECOND



Tests performed on integer array : **BLUE**
Tests performed on object array : **ORANGE**

Benchmarks: Analysis

- ➔ As expected, loops on native type arrays are way faster than loops on object arrays.
- ➔ The interpreted mode (level -1) is the slowest mode: This is because it does not have any optimization and the code is interpreted at runtime.
- ➔ There are two slight peaks on the curve of operations/second at optimization levels 2 and 9 with the integer array tests. No significant impact on the execution time though.
- ➔ The optimizations performed on levels 3 to 8 do not seem to have any significant impact on loops as the curve stabilizes on both benchmarks.

/!\ Hard spot: Analyse the Assembly Code

Rhino doesn’t give an explicit option to retrieve the ASM code from the compiled script.

Strategy: Retrieve the .class files created on compiled mode

The generated .class files can’t be found on the project. We ran the script on compiled mode and compared the number of files and directories and the size of the project before and after and no file was added. The .class are most likely created on Runtime memory and deleted at the end of the program.

/!\ Hard spot: Analyse the Assembly Code

Strategy: Use the PrintAssembly option from the java command line

Giving a simple script performing a loop on a 100-sized integer array:

```
java -jar -XX:+UnlockDiagnosticVMOptions -XX:+PrintCompilation -XX:+PrintAssembly rhino-1.7.13.jar Rhino_JS/src/benchmark/int.js
```

➔ 14 000 lines of ASM code that most likely corresponds to the Rhino code.

```
CompileCommand=print,*MyClass.myMethod rhino-1.7.13.jar -opt 1 Rhino_JS/src/benchmark/int.js
```

➔ Since Rhino does not generate .class files on the hard disk, we can't give the class and method names as parameters of the option CompileCommand.

/!\ Hard spot: Analyse the Assembly Code

Strategy: Compare the ASM code using the same Javascript script but two different levels of optimization (1 and 2)

```
java -jar -XX:+UnlockDiagnosticVMOptions -XX:+PrintNMethods rhino-1.7.13.jar -opt 1 Rhino_JS/src/benchmark/int.js
```

➔ 2 000 lines of ASM code not showing any code from the Javascript script but from Rhino runtime.

➔ Comparing the comments of the two ASM codes with levels 1 and 2 respectively, there is no explicit optimization performed on one over another.

Rhino, an old runtime

Rhino is still supported and maintained currently by the Mozilla foundation and can be found on github: https://github.com/mozilla/rhino.

As Rhino is getting old and obsolete, it has been replaced by Nashorn (german for rhinoceros).