© Philippe PRADOS

Rhino

Philippe PRADOS

pp@philippe.prados.name

L'organisation Mozilla propose une librairie java appelée « Rhino » en licence libre, permettant d'intégrer le langage javascript à votre application java. Elle est téléchargeable ici : www.mozilla.net/rhino

Le langage Javascript a été initialement proposé par Netscape, puis, à partir de la version 1.3, il a été déposé dans le domaine public sous la référence ECMA-262 (www.ecma.ch). La dernière spécification est la version 1.5. Rhino implémente cette version ainsi que toutes les précédentes.

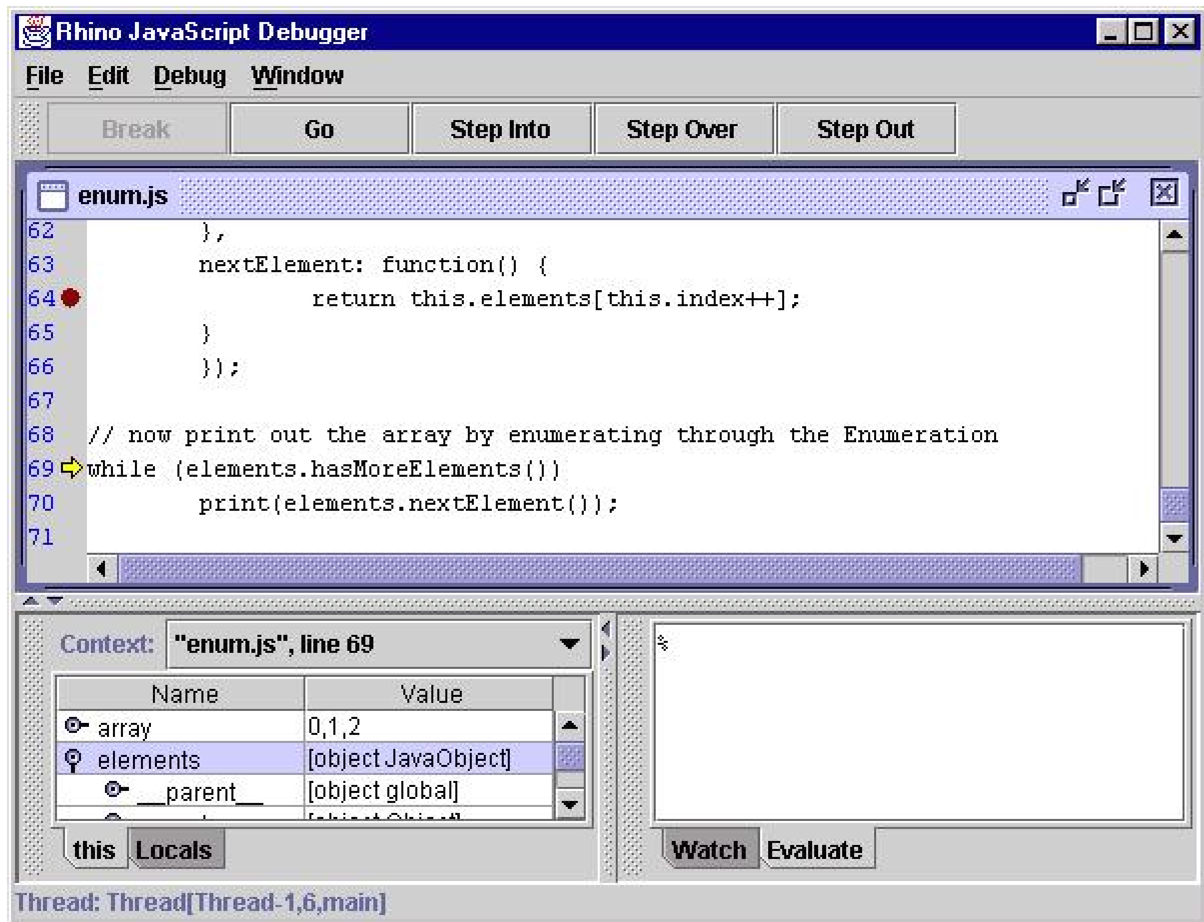
Rhino intègre la technologie « LiveConnect » de Netscape, permettant d'effectuer un lien direct et implicite entre les objets javascript et les objets java. Un script peut manipuler un objet java comme s'il avait été rédigé en script.

Cette librairie permet d'interpréter ou de compiler un programme javascript. La compilation peut s'effectuer à la volé lors de l'utilisation du script ou avant l'exécution, lors de la phase de production de votre programme. Vous pouvez alors intégrer les scripts compilés à tous les programmes javas.

Différents niveaux d'optimisations sont disponibles (de 1 à 9). Il est alors possible d'arbitrer entre le temps de compilation et le temps d'exécution. Un script rarement utilisé peut tolérer une optimisation faible, alors qu'un script utilisé abondamment devra bénéficier de toutes les optimisations possibles. Lors d'une compilation avant la phase d'exécution, il est recommandé d'utiliser le maximum d'optimisation possible ou d'arbitrer cette-fois suivant la taille du code généré. Réduire la qualité de l'optimisation n'améliore par forcément la taille du code produit. Il est nécessaire de faire différents essais afin de réduire la taille de l'archive à télécharger pour exécuter une applet.

Pour que la compilation fonctionne dans une applet, il est nécessaire de signer celle-ci afin d'obtenir l'autorisation d'ajouter un Classloader à la machine virtuelle. En effet, les classes compilées à la volé doivent pouvoir être intégrées à la JVM. A défaut, il est possible d'alléger fortement l'archive `js.jar` en supprimant les packages s'occupant de la compilation. L'archive s'adapte automatiquement à cette situation.

Depuis la version 1.5 release 2, un debugger codé en java permet d'analyser un programme javascript.



Pour exécuter un script avec Rhino, il y a plusieurs approches. La première consiste à lancer un shell javascript. Il y a deux méthodes pour cela :

```
java org.mozilla.javascript.tools.shell.Main
```

ou

```
java -jar js.jar
```

Un prompt vous demande alors d’entrer vos commandes javascript.

```
js> importPackage(java.awt);
js> frame = new Frame("JavaScript")
java.awt.Frame[frame0,0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout,r
esizable,title=JavaScript]
js> frame.show()
js> frame.setSize(new Dimension(200,100))
js> button = new Button("OK")
java.awt.Button[button0,0,0,0x0,invalid,label=OK]
js> frame.add(button)
java.awt.Button[button0,0,0,0x0,invalid,label=OK]
js> frame.show()
js> quit()
```

Cette session affiche la boite de dialogue suivante.

Si vous désirez lancer un script présent dans un fichier, indiquez celui-ci dans la ligne de commande.

```
java org.mozilla.javascript.tools.shell.Main myScript.js
```

Vous devez bien entendu posséder l'archive `js.jar` dans votre CLASSPATH.

LiveConnect propose aux scripts de bénéficier de tous les objets javas et en particulier de tous les beans. Les propriétés des beans sont directement accessibles sans avoir à utiliser les méthodes get ou set.

```
public class Me
{
    public int getAge() { return age; }
    public void setAge(int anAge) { age = anAge; }
    public String getSex() { return "male"; }
    private int age;
};
```

La session suivante manipule une instance Me.

```
js> me = new Packages.Me();
Me@93
js> me.getSex()
male
js> me.sex
male
js> me.age = 33;
33
js> me.age
33
js> me.getAge()
33
js>
```

Quelques API permettent d'améliorer LiveConnect. Des adaptateurs permettent d'implémenter des interfaces java ou de dériver des classes java à l'aide de code JavaScript. Par exemple, le code suivant implémente l'interface `ActionListener`.

```
$ java org.mozilla.javascript.tools.shell.Main
js> importPackage(java.awt);
js> frame = new Frame("JavaScript")
java.awt.Frame[frame0,0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout,r
esizable,title=JavaScript]
js> button = new Button("OK")
java.awt.Button[button0,0,0,0x0,invalid,label=OK]
js> frame.setSize(new Dimension(200,100))
js> frame.add(button)
java.awt.Button[button0,0,0,0x0,invalid,label=OK]
js> frame.show()
js> function printDate() { print(new Date()) }
js> o = { actionPerformed: printDate }
[object Object]
js> buttonListener = java.awt.event.ActionListener(o)
adapter0@6acc0f66
```

```
js> button.addActionListener(buttonListener)
js> quit()
```

Lors de la ligne `buttonListener = java.awt.event.ActionListener(o)`, Rhino crée une nouvelle classe qui implémente l'interface `ActionListerner` et propage les événements à l'instance javascript. Ainsi, lorsque vous appuyez sur le bouton de la boite de dialogue, la fonction `printDate()` est invoquée.

Il est possible d'instancier explicitement un adapter java comme ceci :

```
buttonListener = new JavaAdapter(java.awt.event.ActionListener,o);
```

La syntaxe générale est la suivante :

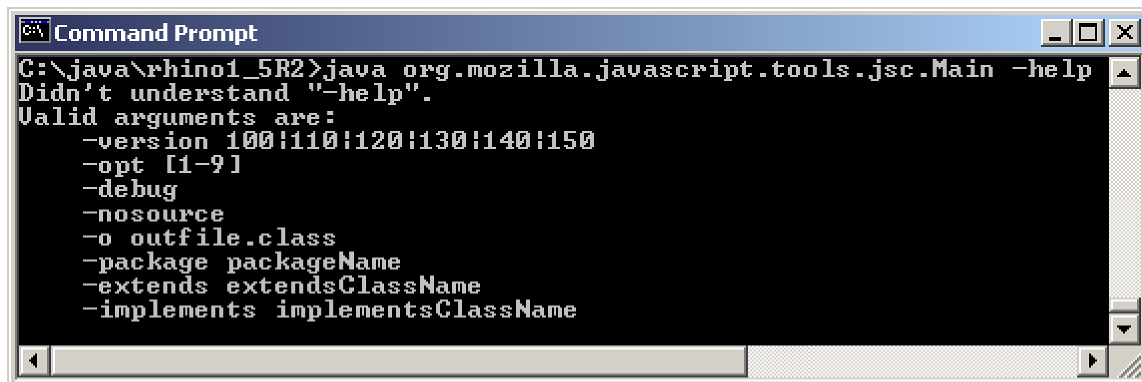
```
new JavaAdapter(java-class, [java-class, ...] javascript-object)
```

Ainsi, un objet javascript peut hériter d'une classe java ou implémenter une interface. L'instance `JavaAdapter` s'occuper d'effectuer la connexion entre les deux mondes.

Pour compiler un programme JavaScript en classe java, utilisez

```
java org.mozilla.javascript.tools.jsc.Main [options] file1.js [file2.js...]
```

Les options permettent d'indiquer le package des classes à générer, la super-classe à dériver, et les interfaces à implémenter.



Intégrer Rhino dans un programme java

Pour intégrer un script javascript dans un programme java, il faut rédiger quelques lignes.

```
Context cx = Context.enter();
Scriptable scope = cx.initStandardObjects(null);
Object result = cx.evaluateString(scope, monScript, "<cmd>", 1, null);
System.out.println(cx.toString(result));
Context.exit();
```

La première ligne ouvre un contexte pour la tâche courante. Rhino accepte les traitements simultanés. Les variables globales au moteur sont portées par un objet `Context`, associé à chaque thread.

La deuxième ligne initialise le premier objet javascript, avec toutes les méthodes et tous les objets standards du langage (Math, etc.)

La troisième ligne demande l'évaluation d'un script à partir du scope initial. Les paramètres permettent d'indiquer le nom du fichier d'où est extrait le script, et la ligne où il commence. Dans cet exemple, n'ayant pas de fichier, nous avons utilisé le nom « `<cmd>` » et la ligne 1.

Pour compiler un script, le code est légèrement différent.

```
Context cx = Context.enter();
Scriptable scope = cx.initStandardObjects(null);
Script compiled = cx.compileFunction(scope,monScript,"<cmd>",1,null)
System.out.println(compiled.exec(cx,compiled));
Context.exit();
```

Le script compilé peut être gardé au chaud, afin d'accélérer les appels suivants.

Vous pouvez rédiger des classes Java directement accessible par les scripts. Cela permet d'offrir un accès simplifié à vos objets. C'est ce que font les navigateurs en proposant les objets `navigator`, `window`, etc.

Par exemple, la classe java suivante offre une nouvelle fonction javascript `f()`.

```
public static class MyFunction extends ScriptableObject
  {
    public MyFunctions()
    {
    }
    public String getClassName()
    {
      return "MyFunction";
    }
    public void jsFunction_f(String s) throws ServletException, IOException
    {
      System.err.println("coucou "+s);
    }
  }
```

Pour bénéficier de cette fonction, il y a deux approches : construire une instance `MyFunction` ou démarrer le script à partir d'une instance `MyFunction`.

La méthode `getClassName()` permet de déclarer la classe Javascript proposée. Le moteur doit alors enregistrer cette classe avant de l'utiliser.

```
js> defineClass("MyFunction")
js> o = new MyFunction()
[object MyFunction]
js> o.f("moi");
coucou moi
```

Une approche plus sympathique consiste à démarrer le script à partir d'une instance `MyFunction`.

```
Context cx = Context.enter();
Scriptable scope = cx.initStandardObjects(null);
ScriptableObject.defineClass(scope,MyFunction.class);
Scriptable myFunction=cx.newObject(scope,"MyFunction",new Object[]{});
Object result = cx.evaluateString(myFunction, monScript, "<cmd>", 1, null);
System.out.println(cx.toString(result));
Context.exit();
```

Le script démarre dans le contexte d'une instance `myFunction`. Il peut alors directement invoquer la méthode `f()`.

```
f("moi");
```

Avec cette librairie, vous pouvez offrir à vos utilisateurs avertis, la possibilité de rédiger facilement des extensions à vos applications, en utilisant un langage standard très répandu. Une journée est suffisante pour maîtriser et intégrer cela.