JavaLikeScript User Guide

spyl <javalikescript@free.fr>

Table of Contents

1. Installation

There is no specific installation, just unzip the distribution in the folder you want.

1.1. Installed Directory Tree

The jls distribution has the directory structure shown below.

`ext`	The extension folder.
`wb`	The web browser folder.
`jls.dll`	The jls engine library (`libjls.so`).
`jls.exe`	The jls executable (`jls`).
`jlsw.exe`	The jls executable without console (`jlsw`).
`js.dll`	The SpiderMonkey library (`libjs.so`).
`jsdoc.zip`	The JavaScript documentation archive for jls standalone.
`nspr.dll`	The Netscape Portable Runtime library (`libnspr.so`).

1.1.1. The extension folder default content

The extension folder default content is the following.

`demo.zip`	The demo archive. It contains scripts for demo the jls framework.

`jls_io.dll`	The jls IO native library (`libjls_io.so`).
`jls_jpeg.dll`	The jls JPEG native library (`libjls_jpeg.so`).
`jls_net.dll`	The jls network native library (`libjls_net.so`).
`jls_win32.dll`	The jls Win32 native library (not available on Linux).
`rt.zip`	The runtime JavaScript library.
`test.zip`	The test archive. It contains JSUnit tests of the runtime library.

1.1.2. The web browser folder default content

The web browser folder default content is the following.

`jls.js`	The runtime JavaScript library in a single file.
`jsdoc.zip`	The JavaScript documentation archive for jls in the web browsers.
`rt.zip`	The runtime JavaScript library.
`test.zip`	The test archive. It contains JSUnit tests of the runtime library.
`php.zip`	A minimalist PHP library.

1.2. Update the PATH variable

You can run jls without setting the PATH variable. If you plane to use jls anywhere you might consider to add its root folder on the system path.

1.3. Linux 64-bit

If you need to run the 32-bit distribution on a Linux 64-bit system then you need the 32-bit libraries.

```
apt-get install ia32-libs
```

2. Usage

2.1. Command line arguments

The command line is the following.

```
jls [options] target [args...]
```

The target could be a script filename or a classname. For both cases the target must be available in the extension or the script path list.

The options include:

`-bs <bootstrap>`	The file name of the script to use for bootstrap.
`-ep <extension path>`	A directory to search for native library files, script directories and ZIP files.

`-lp <library paths>`	A separated list of directories to search for native library files.
`-sp <script paths>`	A separated list of directories and ZIP archives to search for script files.
`-D<name>=<value>`	Set a system property.

The separation character could be ; or : depending on the operating system.

2.2. Garbage collection

By default the standalone jls bootstrap script does not enable garbage collection. If your application generates a lot of object you must manage garbage collection. The default way to enable garbage collection is to set the system properties `jls.gcDelay` to a value in milliseconds.

```
jls -Djls.gcDelay=5000 myApp.js
```

A periodic execution of the garbage collector does not fit to all kind of applications. You must also consider calling the garbage collector directly in your application.

```
System.gc();
```

2.3. Self executable

At startup jls executables look for an initialisation file named with the executable name without the extension (jls.ini for example), if such a file exists then it content is used as the default arguments. The executables jls and jlsw are lightweight executables, just calling the jls native library. Within this feature it is possible to create self executables by copying the default executable using the name of your tool. Depending of your tool you must choose the jls or jlsw executable.

By example if you have created the xyz tool then copy jls to xyz then create the initialisation file xyz.ini with your class or script name in it. To launch your tool you just have to launch the xyz executable.

2.4. Web browser

To use the web browser jls library, you could use the `browser.js` demo script, this script gives access to the runtime and the test archive files.

```
jls browser.js
```

In order to use the jls framework, you must have access to the jls web runtime library (`wb/rt.zip`) and load the bootstrap script like the following.

```
<script src="/rt/bootstrap.js" type="text/javascript"></script>
```

To avoid loading each files separately you could concatenate several files into one and load it using a script tag. The full jls runtime stack is provided in a single file `wb/jls.js` including the bootstrapping.

You can set arguments and properties by using the `jlsOptions` variable before to load the bootstrap script, by example you can set the log level and specify the script path.

```
<script type="text/javascript">
<!--
jlsOptions = {
    properties : {
        'jls.script.path' : '/rt;/test',
        'jls.logger.logLevel' : 'trace'
    }
};
//-->
</script>
<script src="/rt/bootstrap.js" type="text/javascript"></script>
```

3. Concepts

3.1. Bootstrap

At startup the Jls native engine look for a bootstrap script named by default `bootstrap.js`. The default runtime archive contains such a bootstrap script that loads the jls namespace and base lang classes: `jls.lang.Exception`, `jls.lang.ClassLoader`. It also extends JavaScript objects with the prototype framework.

3.2. Classes

3.2.1. Loader

The jls framework comes with an implementation of the Asynchronous Module Definition (AMD)[1] API. A globale require function

```
require('myPackage/MyClass');
```

The jls framework is going to look for a script named `MyClass.js` under the `myPackage` folder in all the script paths. The script will be evaluated and the `myPackage.MyClass` object will be returned.

3.2.2. Creation

A class can be created using the prototype create function. See prototype class inheritance[2].

```
var MyClass = Class.create(
{
    initialize : function(/* ... */) {
        // ...
    }
});
```

The class can be extended with static fields and functions using the extend function.

```
Object.extend(MyClass,
{
    MY_VALUE : 'something'
});
```

If the class is going to be used in other class or script you must tell AMD that this class is about to be defined.

```
define(['jls/lang/Class', ...], function (Class, ...) {
```

[1] https://github.com/amdjs/amdjs-api/wiki/AMD
[2] http://prototypejs.org/learn/class-inheritance

```
    return Class.create({...});
});
```

3.3. Native libraries

Native libraries can be loaded using the default loader. The library name must be provided without the extension (`jls_io` for `jls_io.dll`). The library file must be present in the library or extension paths.

```
Runtime.loadLibrary('libName');
```

By example, the native io library creates the `mkDir()` function in the `io` namespace. This function can be access like the following: `_native.io.mkDir()`.

3.4. Unit testing

The package `jls.jsunit` contains classes for unit testing. A default test runner can be used to launch a test case class.

```
jls jls/jsunit/TestRunner myPackage/SomeTest
```

The `jls.jsunit.TestCase` class can be extend to create test cases. The `jls.jsunit.Assert` class contains functions to check the unit tests.

```
define(['jls/lang/Class', 'jls/jsunit/Assert', 'jls/jsunit/TestCase'],
function (Class, Assert, TestCase) {
  return Class.create(TestCase,
  {
    testSomething : function() {
        jls.jsunit.Assert.assertEquals(2, 1 + 1);
        jls.jsunit.Assert.assertTrue(true);
    }
  });
});
```

3.5. Graphical User Interface

The package `jls.gui` provides an abstract graphical user interface library. Each graphical component is a jls.gui.Element an element is created with two arguments: a parameters object and an optional parent. The parameters object can have the following properties.

`attributes`	The element attributes. The attributes are set/get using the associated setter/getter methods. By example `attributes: {text: 'a label'}` will call `setText('a label')`.
`style`	The element style. The style properties are borrowed from CSS, by example: `style: {width: 120, height: 20}`.
`children`	The element children.