
JavaLikeScript User Guide

spyl <javalikescript@free.fr>

Table of Contents

1. Installation	1
1.1. Installed Directory Tree	1
1.2. Update the PATH variable	2
2. Usage	2
2.1. Command line arguments	2
2.2. Self executable	2
3. Concepts	2
3.1. Namespaces	2
3.2. Bootstrap	3
3.3. Classes	3
3.4. Native libraries	3
3.5. Unit testing	4
3.6. Graphical User Interface	4

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.
jls.dll	The jls engine library.
jls.exe	The jls executable.
jlsw.exe	The jls executable without console.
js.dll	The SpiderMonkey library.
jsdoc.zip	The JavaScript documentation archive.
nspr.dll	The Netscape Portable Runtime library.

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.
jls_jpeg.dll	The jls JPEG native library.
jls_net.dll	The jls network native library.

<code>jls_win32.dll</code>	The jls Win32 native library.
<code>rt.zip</code>	The runtime JavaScript library.
<code>test.zip</code>	The test archive. It contains JUnit tests of the runtime 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.

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:

<code>-bs <bootstrap></code>	The file name of the script to use for bootstrap.
<code>-ep <extension path></code>	A directory to search for native library files, script directories and ZIP files.
<code>-lp <library paths></code>	A separated list of directories to search for native library files.
<code>-sp <script paths></code>	A separated list of directories and ZIP archives to search for script files.
<code>-D<name>=<value></code>	Set a system property.

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

2.2. 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.

3. Concepts

3.1. Namespaces

By default two namespaces are created as root objects.

`jls` The `jls` namespace contains the `jls` framework.

`_native` The `_native` namespace contains the native objects, methods and classes created by a native library.

3.2. 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.

3.3. Classes

3.3.1. Loader

The `jls` framework comes with a default loader facility, see class `jls.lang.ClassLoader`. This loader enables to reuse your classes

```
jls.loader.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.

3.3.2. Creation

A class can be created using the prototype create function. See [prototype class inheritance](http://prototypejs.org/learn/class-inheritance)¹.

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

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

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

If the class is going to be used in other class or script you must tell the `jls` loader that this class is about to be provided.

```
jls.loader.provide('myPackage.MyClass');
```

3.4. Native libraries

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

¹ <http://prototypejs.org/learn/class-inheritance>

```
jls.loader.requireLibrary('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()`.

3.5. 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.

```
jls.loader.provide('myPackage.SomeTest');

jls.loader.require('jls.jsunit.TestCase');

myPackage.SomeTest = jls.lang.Class.create(jls.jsunit.TestCase,
{
    testSomething : function() {
        jls.jsunit.Assert.assertEquals(2, 1 + 1);
        jls.jsunit.Assert.assertTrue(true);
    }
});
```

3.6. 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.

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