JSR-223

Scripting for the Java™ Platform

Final Draft Specification
version 1.0

Specification Lead
Mike Grogan
Sun Microsystems, Inc.
mike.grogan@sun.com

technical comments to
jsr-223-comments@jcp.org

SUN IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS LICENSE AGREEMENT ("AGREEMENT"). PLEASE READ THE TERMS AND CONDITIONS OF THIS LICENSE CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY ITS TERMS, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE AND THE DOWNLOADING PROCESS WILL NOT CONTINUE.

Specification: JSR-223, Scripting for the Java Platform Specification ("Specification")

Status: Public Review

Release: July 31, 2006

NOTICE: The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. ("Sun") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, Sun hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Sun's intellectual property rights to review the Specification only for the purposes of evaluation. This license includes the right to discuss the Specification (including the right to provide limited excerpts of text to the extent relevant to the point[s] under discussion) with other licensees (under this or a substantially similar version of this Agreement) of the Specification. Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Sun intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (i) two (2) years from the date of Release listed above; (ii) the date on which the final version of the Specification is publicly released; or (iii) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

TRADEMARKS: No right, title, or interest in or to any trademarks, service

marks, or trade names of Sun, Sun's licensors, Specification Lead or the Specification Lead's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, J2SE, J2EE, J2ME, Java Compatible, the Java Compatible Logo, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES: THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY SUN. SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY: TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Sun (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND: If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Specification and accompanying documentation shall be only as set forth in this license; this

is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT: You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS: Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

Neither party may assign or otherwise transfer any of its rights or obligations under this Agreement, without the prior written consent of the other party, except that Sun may assign this Agreement to an affiliated company.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

(Sun.pSpec.license.11.14.2003)

Table of Contents

l11

SCR.0.1 Changes since Early Draft Review version 1.0

- Change `Title to` **`Scripting for the Java™ Platform`**
- Require Discovery Mechanism Packaging (SCR.4.2.4.1)
- Add setWriter, getReader and getErrorWriter methods to ScriptContext (SCR.4.3.1)
- Add getContext/setContext methods to ScriptEngine. Specify that a default ScriptContext be associated with a ScriptEngine. The default ScriptContext is used for eval methods where a ScriptContext is not passed as an argument. The getNamespace/setNamespace methods of ScriptEngine are mapped to getNamespace/setNamespace of the ScriptContext. All methods of Java Script Engine interfaces requiring a ScriptContext use the default one unless another is specified. (SCR.4.3.4.1.1)
- Add ScriptEngineInfo methods, getMethodCallSyntax, getOutputStatement and getProgram that generate script code that can be executed by the corresponding script interpreters. (SCR.4.3.5)
- Make implementation of javax.script.http.* optional. (SCR.2.3 SCR.5)

SCR.0.2 Changes Since Public Review Draft version 1.0

- Explicitly permit use of Java 5.0 features (SCR.0.3)
- Switched the order of arguments from Invocable.call(String, Object, Object[]) to Invocable.call(Object,String, Object[]) (4.3.4.3)
- Added Invocable.getInterface(Object, Class) (SCR.4.3.4.3)
- Corrected erroneous second arguments to ScriptEngineManager.registerEngineByExtension and ScriptEngineManager.registerEngineByMimeType in Javadocs (SCR.6.12.3)
- Removed non-normative Appendix describing Java/PHP language

bindings.

- Clarified in Javadocs that GlobalScope namespaces are associated with ScriptEngineManagers (SCR.6.1.2, SCR.6.8.1)
- Made all fields of ScriptEngineManager private and removed references to them in the Javadocs (6.12.3)
- Made all fields of ScriptException private (6.13.1)
- Reserved the use of Namespace keys beginning with “javax.script” for future versions of this specification. (SCR.4.3.4.1.1)
- Clarified that the ScriptEngineManager Discovery Mechanism, lookups in getEngineByMimeType, getEngineByName and getEngineByExtension are case-sensitive and that extensions do not include initial dots. (SCR.4.3.9)

SCR.0.3 Changes Since Public Review Draft version 1.1

- Eliminate the ScriptEngineInfo interface. Coalesce its methods into its ScriptEngineFactory subinterface. (SCR.4.1.1, SCR.4.2.4.2, SCR.4.3, SCR.4.3.4.1.4, SCR.4.3.5, SCR.4.3.6.2)
- Change the signature of the ScriptEngineManager.getEngineFactories method to **List<ScriptEngineFactory> getEngineFactories()** (SCR.4.2.4.1, SCR.4.3.8.1)
- Change every occurrence of “Namespace” in interface, class and method names to “Bindings.” (SCR.4.2.6, SCR.4.2.7, SCR.4.3.1, SCR.4.3.2, SCR.4.3.3, SCR.4.3.4.1.1, SCR.4.3.4.1.2, SCR.4.3.4.2, SCR.4.3.6.2, SCR.4.3.7, SCR.4.3.8.2, SCR.5, SCR.5.3)
- Add the **List<Integer> getScopes()** method to ScriptContext. (SCR.4.3.1)
- Rename GenericScriptContext to SimpleScriptContext. (SCR.4.3.2, SCR.4.3.6)
- Change the base interface of Bindings (formerly Namespace) from Map to Map<String>. (SCR.4.3.4)
- Rename both “call” methods of the Invocable interface to “invoke” (SCR.4.3.4.3)

- Change the type of the last argument to both invoke (formerly call) methods of the Invocable interface from Object[] to Object... (SCR.4.3.5.3)

- Specify that the invoke (formerly call) methods of the Invocable interface throw NoSuchMethodException if it can be determined that no procedure with the specified name exists in the script compiled in the ScriptEngine (SCR.4.3.4.3)

- Change the signatures of the getInterface methods of the InvocableInterface to <T> **T getInterface(Class<T>)** and **<T> T getInterface(Object, Class<T>)** (SCR.4.3.4.3)

- Change the return values of all ScriptEngineManagerMethods that formerly returned String[] to immutable List<String>. (SCR.4.3.6.1, SCR.4.3.6.2)

- Rename GenericScriptEngine to AbstractScriptEngine (SCR.4.3.7)

- Rename Invocable.invoke(Object, String, Object...) and Invocable.invoke(String, Object...) to invokeMethod and invokeFunction respectively.

- Remove javax.script.http package from the specification.

SCR.0.4 Acknowledgements

The specification contains the (continuing) contributions of the JSR 223 Expert Group Members: Dejan Bosanac, Per Bothner, Don Coleman, Felix Meschberger (Day Software, Inc.), Rony G. Flatscher (Wirtschaftsuniversität Wien), Victor Orlikowsky (IBM), Patrick D. Niemeyer, Kabe Robichaux, Elaine Chien and Gael Stevens (Oracle), James Strachan, Alan Williamson, Zev Suraski (Zend Technologies, Ltd.), Awdhesh Kumar (Pramai Technologies), Jacob Levy (Sun Microsystems, Inc.), Rajendra Singh (SAS Institute, Inc.), Edwin Smith (Macromedia , Inc.), Alan Williamson.

A. Sundararajan and Basant Kukreja of Sun Microsystems, Inc. contributed to the Reference Implementation and provided feedback for the

specification.

The specification builds on prior work in the the following open-source projects:

- Bean Scripting Framework (http://jakarta.apache.org/bsf/index.html)
- BeanShell (http://www.beanshell.org)
- PHP (http://www.php.net)

SCR.0.5 Related Specifications and Versions

This document contains references to the following specifications.

- Java Servlet Specification Version 2.4 (http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html)

The specification uses Java 5.0 language features. There is no requirement that implementations be compatible with Java language versions prior to 5.0.

SCR.1 Introduction

This document is the Specification for:

JSR-223 Scripting for the Java™ Platform
Version 1.2.

It contains the standard for the Java Scripting API.

SCR.1.1 Use Cases and History

There are currently several areas where Java and scripting technologies interact.

- **Scripting of Java Objects**

In this common use case, Java classes are used to extend the capabilities of scripting languages. The scripting languages are able to create Java objects and call public methods of the objects using the syntax of the scripting languages. In this way, functionality available in Java that does not exist in the scripting languages can be used in scripts.

Early examples of this technology include Netscape's Live Connect, which is used by client-side Javascript code in HTML pages to invoke the methods of Applets and other Java classes. Microsoft's Java implementations also allowed client-side scripts to invoke methods of Java objects through proprietary interfaces.

- **Java Implementations of Script Interpreters**

The Java programming language has been used to implement interpreters for scripting languages. Examples include the Mozilla

Rhino Javascript implementation; Jacl, a Java-based implementation of the TCL scripting language; and Jython, the Java-based implementation of Python

- **Embedded Java Interpreters**

Most Java-based scripting language interpreters have external interfaces that allow them to be used as components in other applications. The interfaces include methods that execute scripts and ones that associate application objects with scripting variables.

This enables applications to execute scripts provided by end-users that control the behavior of the applications. Uses for this include the use of scripting engines to execute macros which automate the applications and the use of scripts to generate web content.

- **Common Scripting Interfaces**

The Bean Scripting Framework (BSF), originally developed by IBM and now part of the Apache Jakarta project, is a set of common interfaces and classes that can be used with multiple Java-based scripting implementations. There are implementations of the interfaces using most of the Java-based scripting interpreters. These implementations are widely used by applications using scripting that call into scripting engines through the BSF interfaces.

- **Compilation of Java Bytecode from Script Sources**

There are several compilers for scripting languages that produce Java classes. The classes can be executed by the Java runtime. These include Kawa, a framework for compiling scripting languages that includes compilers for the Scheme and XQuery languages, and Jython, a compiler for the Python language which produces Java bytecode.

- **Hybrid Scripting Languages**

Several scripting languages have been developed whose syntaxes resemble the Java programming language. These languages often have internal constructs that allow objects defined in scripts and ordinary Java objects to be used interchangeably.

Programs written in these languages can easily be migrated to Java. These languages can be used as bridges for script programmers to

use in adopting Java. They can also be used by Java programmers for prototyping. Examples of these languages include BeanShell and Groovy, the language being standardized in JSR-241.

SCR.1.2 Goals of JSR-223

The original goal of JSR-223 was to define a standard, portable way to allow programs written in scripting languages to generate web content. In order to do this, it is necessary to have a common set of programming interfaces that can be used to execute scripts in scripting engines and bind application objects into the namespaces of the scripts. Therefore, in addition to a framework for web scripting, the specification includes a standardized Scripting API similar to the Bean Scripting Framework. It uses the Scripting API to define the elements of the Web Scripting Framework.

The main design goals of the Scripting API strive toward portability and backward-compatibility. The interfaces should allow developers to determine the expected behavior of implementations at runtime, allowing different code paths to handle different types of allowable behavior. The interfaces should resemble existing ones to the greatest extent possible, allowing their easy adoption by developers accustomed to the existing interfaces.

There are several areas which are intentionally omitted from the specification:

- The specification does not define how scripting languages should enable the use of Java objects in scripts, although it is assumed that the scripting languages implementing the specification have this functionality.
- The specification does not distinguish between scripting implementations that compile script sources to Java bytecode and those that do not. Script engines that do can be used to implement the specification, but it is not required.
- The specification makes no requirements of scripting languages or the syntax uses to invoke the methods of Java objects in the languages.

SCR.1.3 Who Should Read the Specification?

The specification is of interest to developers who wish to implement scripting interpreters that can be used as components in Java applications. Similarly, it is of interest to Java developers who wish to use scripting interpreters in their applications. It is also useful for Web Application developers wishing to deploy their applications in implements of the Web Scripting Framework.

It is of less interest to script programmers providing scripts to be run by the applications, since scripting language details and the mechanisms through which scripting languages use Java objects are largely unspecified.

SCR.2 Overview

SCR.2.1 Scripting Fundamentals and Terminology

In this specification, a ***scripting engine*** is a software component that executes programs written in some scripting language. The execution is generally performed by an ***interpreter***. Conceptually an interpreter consists of two parts: a ***front-end*** which parses the source code and produces an internal representation of the program known as intermediate code, and a back-end which uses the intermediate code to execute the program.

The **back-end** of the interpreter, also known as the ***executor***, uses ***symbol tables*** to store the values of variables in the scripts.

This specification assumes that all software components are Java objects. Some of them might use native methods.

There are Java implementations of interpreters for many scripting languages. The implementations vary widely in terms of external programming interfaces and functionality.

All the interpreters expose programming interfaces including methods that execute specified scripts. Most of the interfaces include methods that allow values of scripting variables in the symbol tables to be read and written.

Interpreters differ in the persistence and visibility of the state contained in intermediate code and symbol tables. These differences affect the programming interfaces and the behavior of the interpreters. The programming interfaces also differ in whether they expose the functionality of front-end and back-end separately and whether they expose the intermediate code.

Scripting engines which implement the fundamental scripting interface defined in this specification are known as **J*ava Script***

Engines. Conceptually, a Java Script Engine can be thought of as an interpreter, but this may not actually be the case. For instance scripts executed by a single Java Script Engine may be executed internally by different interpreters.

SCR.2.2 Technologies Discussed in this Document

Three specific technologies are discussed in this specification.

- Java Language Bindings – Mechanisms that allow scripts to load Java classes, create instances of them and call methods of the resulting objects. The discussion includes an exploration of how method calls in scripting languages are translated into Java method calls on the resulting objects, how scripting arguments are converted to Java arguments before the calls and how the resulting Java return values are represented in scripts after the calls.

- General Scripting API – Interfaces and classes that allow script engines to be used as components in Java applications. Methods of the classes and interfaces include ones which execute scripts written in the scripting languages as well as ones that create bindings between scripting language variables and Java objects in the host applications.

The specification does not deal with issues of scripting language design or interpreter implementation. Similarly, the specification is limited to script interpreters and does not deal specifically with interesting Java technologies which compile script source code to end-user-visible Java class files, although implementations might use these technologies internally.

SCR.2.3 Organization of this Document

The next three sections deal with Java Language Bindings, the Scripting API and Web Scripting respectively. Each section depends on its predecessors, since it is assumed that every implementation of the Scripting API incorporates Java Language Bindings and the

framework described in the Web Scripting section uses the Scripting API.

The section on Java Language Bindings is descriptive but not normative. This is because the way that bindings are defined depends on specifics of particular scripting languages. The section discusses general principals concerning Java Language Bindings .

The Scripting API is a specification of a scripting framework. It includes a discussion of which features are implementation-dependent and describes practices that application programmers should follow to maximize the probability that their applications will be portable between different implementations of the Scripting API.

SCR.3 Java Language Bindings

Mechanisms that allow scripting engines to store Java object and class references and use them in scripts are referred to as **Java Language Bindings**. Bindings allow scripts to access public methods and fields of objects and public static methods and fields of classes. The ways that bound objects are referred to in scripts and the ways that their methods are invoked vary according to the semantics of the scripting languages.

Typically, a Java Object or Class reference is stored in a scripting engine data structure associated with a script variable. This document will refer to such a data structure as a **Java Object Proxy.** Methods of the stored object are invoked from scripts using the associated variable name.

SCR.3.1 Creation of Bindings

Bindings belong to three types:

- Dynamic bindings – Bindings are created during execution of scripts using scripting language constructs.

- Programmatic bindings – The scripting engine is a component that can be embedded in an application. The bindings are created by the host application invoking methods of interfaces implemented by the scripting engine.

- Static bindings – Scripting language constructs are implemented by the scripting engine using the bindings. The bindings are created during the creation of the scripting engine.

SCR.3.1.1 Dynamic Bindings

Most scripting engines support the creation of bindings during script execution. These bindings are known as dynamic bindings.

For instance, in a a PHP scripting engine, the statement:

```
$javadate = new Java(“java.util.Date”)
```

might create a binding where the scripting variable $javadate is associated with a Java Object Proxy storing a reference to a java.util.Date object. The binding is created by a built-in scripting function. Other implementations might allow Java package and class names to be imported into script namespaces. The Rhino Javascript engine allows:

```
importPackage(java.util);

var hashtable = new Hashtable();
```

Here, the name of a class in a package imported into the script namespace is treated by the script runtime as a script object type. The Java Object Proxy is automatically created by the script interpreter.

In the previous examples, the Java Object references wrapped in the Java Object Proxies are created using default constructors. However, engines supporting dynamic binding creation also allow the use of non-default constructors. For instance, the PHP Java Scriping Engine

might allow:

```
$socket = new Java(“java.net.Socket”, “java.sun.com”, 80)
```

The Rhino Javascript engine allows:

```
importPackage(java.net);
var socket = new Socket(“java.sun.com”, 80);
```

In both cases, a binding to a java.net.Socket object is created using the constructor

```
Socket(String host, int port)
```

SCR.3.1.2 Programmatic Bindings

In cases where scripting engines are components embedded in host applications, bindings can be created programmatically using interfaces implemented by the engines. The Java Object Proxies in these bindings contain objects from the host application that can be used by scripts executed by the engines. This mechanism is discussed later in this specification in the discussion of Scripting API.

SCR.3.1.3 Static Bindings

A script engine might implement built-in functions using Java. In Java-based implementations, this will be the case with all built-in functions. PHP uses native callback functions in its host application, usually a web server plug-in, to implement some of its built-in functions such as **echo**, **header**, **cookies**, etc. In a PHP scripting engine designed to be used in a Java Web Application, the callbacks might implemented using JNI calls into ServletRequest and ServletResponse objects provided by a Web container. Bindings of this type are known as static bindings.

Static bindings are created by the implementer of a engine rather than a user of one.

SCR.3.2 Member Invocations

Members include constructors as well as static and instance methods. Invocations are generally made on the Objects stored in proxies using reflection. In the cases of constructors and static methods, the objects stored by the proxies might be java.lang.Class instances. The handling of arguments and return values is similar in all the cases, since they are handled similarly in the Java reflection API

SCR.3.2.1 Instance Methods

In languages which support objects, a method call on a Java object stored in a Java Object Proxy is generally made using the syntax ordinarily used to make scripting language method calls on scripting language objects.

For example, in a PHP scripting engine, the code:

```
//instantiate a java object
$javadate = new Java(“java.util.Date”);


//call a method
$date =$javadate->toString();


//display return value
echo($date);
```

Should display the current date.

A scripting language that does not have a special method invocation syntax, might use special functions instead.
For instance Kawa Scheme uses the following code to invoke the toString method of java.util.Date:

```
;; Instantiate Java object:
(define java-date (make <java.util.Date>))

;; Call a method:
(define date (invoke java-date 'toString))

;; Display results:
(display date)
```

SCR.3.2.2 Static Methods

Static methods can be called from scripts using variables bound to proxies containing either class references or instances of the classes. In a PHP Java Script Engine

```
$thread_class = JavaClass(“java.lang.Thread”)

$thread = $thread_class->currentThread()
```

might store a reference to the current thread. The thread can be paused for one second using either:

```
$thread_class->sleep(1000);
```

or

```
$thread->sleep(1000);
```

The script engine implementation might also contain built-in functions which invoke static methods directly without using java object proxies. For instance, in Kawa Scheme, the following code invokes the static currentThread method of java.lang.Thread.

```
(define thread (invoke-static
                <java.lang.Thread>'currentThread))
```

Finally, in languages that can include Java class and package names in their namespaces, static methods might be invoked directly from scripts as in the following Javascript code which can be executed by the Rhino Java Scripting Engine:

```
import(java.lang);
var thread = Thread.currentThread();
```

SCR.3.2.3 Constructors

Scripting engines may invoke constructors to create new bound objects. For instance, to implement the code:

```
import(java.net);

var x =  new Socket(“http://java.sun.com” , 80);
```

The Rhino Script Engine must invoke the java.net.Socket constructor with String and int arguments.

SCR.3.3 Member Invocation Process

In most of the previous examples, members of Java classes taking no arguments were invoked from the scripts. Java Scripting engines should allow members with arbitrary arguments and return values to be invoked. This means that for each invocation on a Java Object Proxy, the engines need to complete the following steps:

- **Conversion of Arguments**
  Convert the values represented by the script arguments to Java objects (or sometimes as jvalues if Java member invocations are done directly in native code using JNI calls)

- **Member Selection**
  Find all members in the underlying Java class with the specified name and type whose parameter lists match the array of arguments obtained by converting the arguments from the script.

- **Overloaded Method Resolution**
  If more than one matching member is found, determine which

signature most closely matches the argument list.

- **Invocation**
  Usually java.lang.reflect.Method.invoke or java.lang.reflect.Constructor.newInstance is used, passing the method or constructor found in the previous step and an Object[] whose members are the objects resulting from the conversion of the arguments. Sometimes, the invocation will be made using the JNI CallXXXMethod() methods.

- **Conversion of Return Value**
  Represent the return value, if any, in the script -- usually as a script variable. This might require creation of a new Java Object Proxy for the return value if the return value is not primitive.

The same steps must be performed whether an instance method, a static method or constructor is invoked. Depending on the scripting engine, one or more of these steps might be done using JNI in native code. Java-based scripting engines (engines implemented entirely in Java, such as Jython, Rhino and BeanShell) will perform all of them in Java. Native implementations might perform all of them in native code.

Each of the operations might result in errors. Operations taking place in the Java Virtual Machine may throw Exceptions. Each error must be handled so it can be reported in the output from the script. If the scripting language has an exception mechanism, Java Exceptions should be converted to script exceptions and these should be thrown in the calling script.

The following section will examine each of the operations in more detail.

SCR.3.3.1 Conversion of Arguments

The data types represented by script variables vary depending on the scripting language and the script engine implementation. Typically, script variable names are associated with internal data structures in script engine symbol tables. The values stored in the symbol tables might be Java Object Proxies or data structures wrapping other types.

In the Java-based scripting engines, the values are themselves Java objects. The following rules generally apply.

- Variables that reference Java Objects are resolved to the Objects.
- Variables referencing numeric types are converted to java.lang.Number instances. In the Java-based scripting languages, where the symbol table values are usually java.lang.Number instances or wrappers for them, the actual java.lang.Number instances are returned. An engine using JNI for individual method invocations may convert all numeric variables to jbyte, jshort, jint, jlong, jfloat or jdouble types.
- Boolean or Character variables are converted to instances of java.lang.Boolean or java.lang.Character (jboolean or jchar in the JNI case)
- String variables are converted to java.lang.String instances. (jstring in the JNI case).
- Array variables are converted to Object[] instances (in the JNI case, jarrays created by one of the CreateXXArray(), where the type represented by “XX” might be Object or primitive depending on the contents of the script array) Individual members of the script arrays are converted the same way individual variables are and the converted values are stored at the corresponding indexes in the target array.
- In languages with a special “null” datatype, that value is converted to null Java Object reference.
- Languages with language-specific datatypes which logically correspond to Java datatypes might convert variables representing these types to their java counterparts.

SCR.3.3.2 Member Selection

For bindings other than static ones, the engine must determine which member to call based on the type of the object wrapped in the proxy, the name of the member, its type (static, non-static, constructor) and the number and types of the objects resulting from the conversion of the arguments from scripting variables to Java objects. The names,

types and signatures of all the public members can be determined using the reflection APIs. The script engine must determine which of the members have names and types matching those requested in the script. To make this determination, it might need to allow variations of method names if the scripting language has case-insensitive identifiers. Also, variations of type must be allowed if the implementation allows static members to be invoked though instances of the classes. For each member with allowable name and type, the engine must determine whether it can be invoked with the supplied arguments by comparing the arguments with the types specified in the member's signature. For instance, a Javascript Scripting engine might use either constructor:

```
Socket()

Socket(String host, int port)
```

based on the script arguments.

It must use the first to execute the code:

```
import(java.net);

var sock = new Socket();

sock.connect(new InetAddress(“java.sun.com”, 80));
```

and the second to execute:

```
import(java.net);

var sock = new Socket(“java.sun.com”, 80);
```

When determining whether a set of arguments from a script matches a method signature, the engine should allow several types of implicit conversions. These include conversions allowed by Java, such as converting a class to an interface it implements or a superclass; conversion of java.lang.Number types to their corresponding primitive

types, and vice-versa; and conversions allowed by the scripting language which are not allowed by java.

For instance when the PHP Scripting Engine executes the code:

```
$x = new Java(“java.net.Socket”)

$x->connect(“java.sun.com”, “80”)
```

both arguments to connect will be converted to instances of

java.lang.String. These types do not exactly match any of the signatures of the java.net.Socket.connect methods, but since PHP allows the implicit conversion of strings representing integral values to the integral values themselves, the arguments can be coerced to match the signature of

```
void connect(String host, int part)
```

using the conversion of “80” to 80 which is allowed by PHP.

SCR.3.3.3 Overloaded Method Resolution

If multiple methods have signatures which match the types of the specified arguments after the conversion rules are applied, there needs to be a procedure to determine which method's arguments list is the “closest match” to the supplied arguments. For example, the PHP code:

```
$filewriter = new Java(“java.io.FileWriter”,
                       “/tmp/proc.pid”);

$filewriter->write(“32”);
```

might invoke either of:

```
void write(String s)

void write(int i)
```

However, the first method should be chosen, since the string argument passed in the script is an exact match for the signature of the method.

If no member is found with allowable name, type and signature, a descriptive error must be returned to the script output.

SCR.3.3.4 Invocation

If members matching the name, type and arguments specified in the script are found, the arguments may be converted to the exact types in the best matching member's signature and the member may be invoked. The script engine may have to:

- Handle exceptions thrown by the Java method call. This might entail returning data describing the exception, possibly including, message, type and stack trace of the exception to the script output. If the scripting language has its own exception mechanism, the descriptive data from the Java exception should be copied to a scripting language exception thrown from the script.

- Free any local and global references created by conversion of the arguments and handling of exceptions if the member is invoked using JNI.

SCR.3.3.5 Converting Return Values

If the script assigns the return value of the Java method call to a scripting variable, the return value must be associated with the variable by the engine. In general this entails creating a Java Object Proxy for the return value and binding it to the variable. Certain return value types such as java.lang.Number, java.lang.String, java.lang.Boolean, java.lang.Character and null should be represented by corresponding script data types.

SCR.3.4 Property Access

Objects in some scripting languages support the concept of "Properties." Java language bindings should follow the convention of translating "gets" and "sets" of scripting variables bound to Java Object Proxies to invocations of getXXX and setXXX accessors on the wrapped object references, so if an appropriate accessor is available, property access is a special case of method invocation.

If an accessor corresponding to property set or get is not available an implementation might use a public field in the underlying class with the correct name and type. In the case of a set operation, the datatype of the field must match the type of the converted value from the script. In the case of a get operation, the value of the field must be returned to the script as described in the previous section.

SCR.4 Scripting API

The Scripting API consists of a framework that allows Java Scripting Engines to be used in Java Applications and classes and interfaces and that define Java Scripting Engines. The API is intended for use by application programmers who wish to execute programs written in scripting languages in their Java applications. The scripting language programs are usually provided by the end-users of the applications.

The classes and interfaces specified here belong to the javax.script package unless otherwise stated.

SCR.4.1 Goals

SCR.4.1.1 Portability

Portability is the extent to which programs behave identically in different implementations of a specification. The Scripting API is designed to maximize portability from the point of view of application programmers using the API to embed scripting implementations in their applications. It is less feasible for the API to provide portability from the point of view of script programmers. Different Java Scripting Engines might support different scripting languages. Even if two Java Scripting Engines support the same scripting language, their behavior from the point of view of the script programmer will be affected by their implementations of Java Language Bindings.

Even from the point of view of an application programmer, implementation details of Java Scripting Engines affect portability. For instance, some engines may not be designed to work in multi-threaded applications. Others may use features which are not required for all implementations of the specification. Applications that use these features cannot be expected to work with all Java Scripting Engines.

A reasonable portability goal is that the API provide application programmers with the means to determine at runtime what features and behavior to expect from a given implementation when the features and behavior are not strictly defined by the specification. This allows the developer to provide code paths for specific cases, or fail correctly
when running with an implementation missing a necessary optional feature.

Portability issues are addressed in several ways:

- Metadata

  The specification requires that each Java Scripting Engine be accompanied by a class that implements ScriptEngineFactory, an interface that includes ScriptEngine metadata methods, as well as a method used to obtain instances of ScriptEngines. The metadata methods include ones describing attributes of the engine, including its name, supported language and language version. An instance of this ScriptEngineFactory class is returned by one of the methods of the Java Scripting Engine.

- Layered Interfaces

  The fundamental interface implemented by all Java Script Engines, javax.script.ScriptEngine, contains only methods which are expected to be fully functional in every Java Script Engine used by an implementation, and the specification clearly describes what behavior of these methods might be implementation-dependent. An application using this interface which avoids implementation-dependent usage can be expected to work with every implementation.

  Features that are not required in all Java Script Engines, including those that functionally separate the front-end and back-end of an interpreter, are exposed in sub-interfaces (Compilable, Invocable). It is not required that Java Script Engines implement these interfaces. Applications that use them can provide alternate code paths or fail gracefully when running with Java Script Engines that do not implement them.

SCR.4.1.2 Backward Compatibility

The API is designed to resemble existing scripting engine interfaces as much as possible to facilitate compatibility with applications that use the existing interfaces and ease of use by programmers familiar with the interfaces.

SCR.4.2 Functional Components

This section discusses the main areas of functionality of the Scripting API.

SCR.4.2.1 Script Execution

Scripts are streams of Characters used as sources for programs executed by interpreters. Methods defined in this specification do not distinguish between types of scripts. The methods differ only in the form in which scripts are passed to the interpreter. Some use a String, others a Reader.

In particular, the Scripting API does not distinguish between scripts which return values and those which do not, nor do they make the corresponding distinction between evaluating or executing scripts. All scripts are executed using methods which returns an Object. When scripts for which the interpreter returns no value are executed, null is returned. Implementations using existing interpreters that use different methods to execute different types of scripts must internally determine which interpreter method to call.

Scripts are executed synchronously. There is no mechanism defined in this specification to execute them asynchronously or to interrupt script execution from another thread. These must be implemented at application level.

Script execution uses the **eval** methods of the ScriptEngine and Compilable interfaces which are discussed in later sections.

SCR.4.2.2 Compilation

Compilation functionality allows the intermediate code generated by the front-end of an interpreter to be stored and executed repeatedly. This can benefit applications that execute single scripts multiple times. These applications can gain efficiency since the interpreter's front-end only needs to execute once per script rather than once per script execution.

This can only be implemented by Java Script Engines that have programming interfaces exposing front-end and back-end functionality separately and retain intermediate code generated by the front-end between script executions. This is not the case with all Java Script Engines, so this functionality is optional. Engines with this functionality implement the Compilable interface. The intermediate code produced by the interpreters is stored by instances of the CompiledScript interface. These interfaces are discussed in later sections.

SCR.4.2.3 Invocation

Invocation functionality also allows the reuse of intermediate code generated by a script interpreter's front-end. Whereas Compilation allows entire scripts represented in intermediate code to be re-executed, Invocation functionality allows individual procedures in the scripts to be re-executed.

As is the case with compilation, not all interpreters provide invocation functionality, so it is not required for Java Script Engines. For those that do provide it, it is exposed in the methods of the Invocable interface discussed in later sections.

SCR.4.2.4 Engine Discovery and Metadata

Applications written to the the Scripting API might have specific requirements of a Java Scripting Engine. Some may require a particular language or language version, others may require a particular Java Scripting Engine or a particular version of an engine.

The specification includes a requirement that its implementations be packaged in a way that allows them to be discovered at runtime and queried for attributes that an application can use to determine their capabilities and usability. Because of this mechanism, there is no need for applications to be built or configured with a list of available Script Engines. The mechanism is referred to as the **Script Engine Discovery Mechanism**. It is used by the ScriptEngineManager to locate Java Script Engines satisfying specific properties, including Engine name and version, language name and version, and supported extensions and mime types. The ScriptEngineManager is described later in this specification.

SCR.4.2.4.1 Discovery Mechanism

The Discovery Mechanism is based on the Service Provider mechanism described in the **Jar File Specification.** The mechanism specifies a way to deploy a Java Script Engine in a Jar file.
As discussed in later sections, each ScriptEngine must be accompanied by a corresponding ScriptEngineFactory. The implementing classes must be packaged in a Jar file which includes the text file resource:

META-INF/services/javax.script.ScriptEngineFactory.

This resource must have a line for each class implementing ScriptEngineFactory that is packaged in the Jar file. The line consists of the (fully-qualified) name of the implementing class. The parsing of the file ignores whitespace other than line breaks. Any characters following a '#' on a line are ignored.

An example of the

META-INF/services/javax.script.ScriptEngineFactory.

resource in a Jar file containing two ScriptEngineFactory implementations is:

```
#list of ScriptEngineFactory's in this package

com.sun.script.javascript.RhinoScriptEngineFactory #javascript
com.sun.script.beanshell.BeanShellEngineFactory #BeanShell
```

ScriptEngineManager includes a method that returns a List of instances of all implementations of ScriptEngineFactory deployed according to this specification which are visible in the current ClassLoader.

SCR.4.2.4.2 Script Engine Metadata

Each implementation of ScriptEngine must be accompanied by an implementation of ScriptEngineFactory whose methods include ones that expose metadata associated with the corresponding Java Script Engine. The metadata includes Engine name and version as well as Language name and version. Therefore, each ScriptEngineFactory enumerated by the the Script Engine Discovery Mechanism can be queried for the capabilities of its associated ScriptEngine class.

SCR.4.2.5 Script Engine Instantiation

Java Script Engines may simply be instantiated using their public constructors as well as by using the methods of ScriptEngineManager.

As mentioned previously, each Java Script Engine must have a corresponding ScriptEngineFactory. The ScriptEngineFactory has a method used to create an instance of the ScriptEngine. This method is used by ScriptEngineManager to instantiate Java Script Engines.

Engines which are instantiated without the use of a ScriptEngineManager do not have access to the global collection (Global Scope) of key/value pairs associated with each ScriptEngineManager. Otherwise they have the same functionality.

SCR.4.2.6 Bindings

State in the Scripting API is managed in sets of key/value pairs known as scopes**.** Scopes are accessed using the bindings interface, which

extends Map<String, Object> with the same methods, with the additional requirement that all of its keys all be non-empty and non-null.

Each Java Script Engine has a Bindings known as its **Engine Scope** containing the mappings of script variables to their values. The values are often Objects in the host application. Reading the value of a key in the Engine Scope of a ScriptEngine returns the value of the corresponding script variable. Adding an Object as a value in the scope usually makes the object available in scripts using the specified key as a variable name. Methods of the Object can be invoked using that name.

In some cases, naming requirements of scripting languages force the names of scripting variables to differ from the keys they represent in the Engine Scope of a ScriptEngine. For example, PHP requires that varible names begin with the '$' character. In this case, the scripting variable associated a key in an Engine Scope might be formed by prepending '$' to the key. Some languages use two-level qualified names, consisting of a "package" or "namespace uri" together with a "local name". Examples include languages that use XML-style QNames (such as XPath, XQuery, and XSLT), and also Common Lisp. It is suggested that implementors encode QNames as Strings such as **{NAMESPACE_URI}LOCAL_NAME** -- the namespace URI in braces followed by the local name. This format is consistent with the toString method of javax.xml.namespace.QName.
In addition, engines are encouraged to recognize **NAMESPACE_PREFIX:LOCAL_NAME**. Assuming NAMESPACE_PREFIX has been defined using a namespace declaration as an alias for NAMESPACE_URI, this should be equivalent to **{NAMESPACE_URI}LOCAL_NAME**. For a language such as Common Lisp, which has packages with names and aliases but no namespace declarations, the second form is recommended: **PACKAGE_NAME:LOCAL_NAME**.

Another class of key/value pairs in the Engine Scope consists of ones with reserved keys. The values in these pairs have specified meanings. The keys of this type defined in this specification are of the form:

```
javax.script.XXX.
```

The values of these keys may also be exposed in scripts, with or without the

```
javax.script
```

prefix. For instance, the value of the key

```
javax.script.filename
```

is defined to be the name of the file or resource containing the source of the script. Setting this value makes the name of the source available to the underlying scripting implementation for use in constructing error messages. Optionally, the Script Engine might make the name available in scripts using one of the names:

```
javax.script.filename

filename
```

Other scopes of attributes defined in the Scripting API represent state in the host application. They are initialized with data from the host and presented to the Java Script Engine, allowing the data to be used internally by the engine as well as by scripts running in the engine.

One such scope is the Bindings maintained by each ScriptEngineManager known as its Global Scope. This Bindings is available through the API to each ScriptEngine created by a ScriptEngineManager.

SCR.4.2.7 Contexts

Sets of scopes are passed to ScriptEngines in ScriptContexts. Each ScriptContext has a well-defined set of Bindings that it exposes.

Every ScriptContext exposes a **Global Scope** that is shared by all ScriptEngines using the ScriptContext and an **Engine Scope** that usually coincides with the Engine Scope of a ScriptEngine using the ScriptContext. ScriptContext subinterfaces designed to be implemented by specific types of applications might define additional scopes that are meaningful in those applications.

In addition to scopes, ScriptContexts expose other aspects of host applications to the ScriptEngines using them. For instance, the ScriptContext interface has methods returning Readers and Writers for ScriptEngines to use in displaying output and reading input.

Every script execution by a ScriptEngine uses a ScriptContext. In some cases, an instance of ScriptContext is passed to the ScriptEngine method that executes the script. In other cases the default ScriptContext of the ScriptEngine is used.

Each ScriptEngine is associated with a default ScriptContext. The default ScriptContext can be changed using the scripting API. The Engine Scope and Global Scope of a ScriptEngine are exposed as the Engine Scope and Global Scope of its default ScriptContext.

SCR.4.3 Architectural Components

The major architectural components of the framework defined by the Scripting API are the ScriptEngine, the ScriptContext, the ScriptEngineFactory, and the ScriptEngineManager. The components are named according to the interface or abstract class that provides their primary programming interface. The components were referred to in the discussion of functional components.

The ScriptContext interface is used to provide a view of the host application to a ScriptEngine. It exposes scopes of name/value pairs. The scopes differ in their visibility and their meaning. The ScriptContext interface exposes a GlobalScope, which is shared by all ScriptEngines using a ScriptContext and an Engine Scope which contains the mappings of scripting variables to their values for an engine using the ScriptContext. Subclasses of ScriptContext might

define other scopes.

The ScriptEngine interface is an abstraction of a script interpreter. Its primary interface, which is implemented by all Java Script Engines, has methods that execute scripts and ones that allow key/value pairs to be passed to their interpreters. Optional interfaces allow repeated execution of intermediate code generated by an interpreter front-end and invocation of procedures in the intermediate code.

Every ScriptEngine class is associated with a ScriptEngineFactory, whose methods create instances of the class. The ScriptEngineFactory interface also includes methods that return metadata associated with the particular ScriptEngine class, so an instance can be queried for the capabilities of the ScriptEngines it creates.

The ScriptEngineManager class implements the Discovery Mechanism, that locates ScriptEngineFactories for all of the ScriptEngines classes that can be loaded by the current ClassLoader. Each ScriptEngineManager also maintains a collection of key/value pairs, known as its Global Scope, which is available to all ScriptEngines created by the ScriptEngineManager.

SCR.4.3.1 ScriptContext

The ScriptContext interface exposes key/value pairs in various scopes. The scopes are identified by integer values defined as static final fields in the ScriptContext interface: ENGINE_SCOPE and GLOBAL_SCOPE. Subinterfaces may define additional scopes and int fields identifying them.

The

```
List<Integer> getScopes()
```

methods returns an immutable List of all the valid scopes in a particular subinterface.

The methods:

```
void setAttribute(String key, Object value, int scope)

Object getAttribute(String key, int scope)

Object removeAttribute(String key, int scope)
```

are used to access and manipulate individual attributes exposed by the ScriptContext. In each case, the method must have the same effect as the put, get or remove methods of the Bindings identified by the scope argument.

The methods:

```
int getAttributesScope(String name)

Object getAttribute(String name)
```

are used to return information about attributes and scopes. The method getAttribute(String) returns getAttribute(String, int) for the lowest scope in which it returns a non-null value.

The method getAttributesScope(String) returns the integer identifying the lowest value of scope for which the attribute is defined.

The methods:

```
void setBindings(Bindings bindings, int scope)

Bindings getBindings(int scope)
```

are used by host applications to set and replace the Bindings of attributes in a particular scope.

The Bindings interface is defined in a later section.

In addition to methods exposing scopes of attributes, ScriptContext and its subinterfaces contain methods that expose other functionality of host application to ScriptEngines using them. The methods of this type defined in the ScriptContext interface are:

```
Writer getWriter()

Writer getErrorWriter()

Reader getReader()
```

These methods provide streams that are used for script input and output. Implementations whose scripting languages have well-defined constructs to display character output, must use the Writer returned by the getWriter method for this purpose.

Subinterfaces of ScriptContext designed for use in specific types of applications might have methods specific to those applications.

SCR.4.3.2 SimpleScriptContext

The SimpleScriptContext class is a simple default implementation of ScriptContext. Its scopes of attributes are stored in its fields **globalScope** and **engineScope**, which are instances of SimpleBindings. The initial values of the fields contain no mappings.

The

```
void setBindings(Bindings bindings, int scope)

Bindings getBindings(int scope)
```

methods simply read and write these fields.

The engineScope and globalScope fields are used to implement the methods that read and write the values of attributes. The

```
Writer getWriter()

Writer getErrorWriter()

Reader getReader()
```

methods are implemented using the System.out, Sysem.err and System.in streams.

SCR.4.3.3 Bindings and SimpleBindings

Bindings is an interface exposing a collection of key/value pairs. SimpleBindings is an implementation of Bindings using an instance of Map<String, Object> passed to a constructor to store and expose the pairs.

The methods of Bindings are exactly those of Map<String, Object> The only difference is that the

```
Object put(String key, Object value)
```

method is required to throw a NullPointerException if passed a null value for the key argument and is required to throw an IllegalArgumentException if the key argument is an empty String.

Similarly, the

```
void putAll(Map<? extends String, ? extends Object>
toMerge)
```

method must throw an NullPointerException if some key in the specified Map is null, or if any key is thempty String.

Every method of SimpleBindings simply calls the corresponding method of the underlying Map. The implementations of

```
Object put(String key, Object value)

void putAll(Map<? extends String, ? extends Object>
toMerge)
```

also validate the condition that all keys are non-null, non-empty Strings.

SCR.4.3.4 ScriptEngine

SCR.4.3.4.1 ScriptEngine(Primary Interface)

Every Java Script Engine supported by an implementation of this specification must implement the ScriptEngine interface.

SCR.4.3.4.1.1 Bindings, Bound Values and State

Every ScriptEngine has a default ScriptContext, which can be set and read using the:

```
ScriptContext  getContext()

void setContext(ScriptContext ctxt)
```

methods.

Each ScriptEngine has an associated Engine Scope and Global Scope Bindings. They can be set and read using the

```
Bindings getBindings(int scope)

void setBindings(Bindings n, int scope)
```

methods passing the integer values

```
ScriptContext.ENGINE_SCOPE

ScriptContext.GLOBAL_SCOPE
```

defined as static final fields in the ScriptContext interface.

Values of the key/value pairs in the engine scope can be read and written using the returned Bindings.

The default ScriptContext and the Engine Scope and Global Scope Bindings are related in the following ways:

- The Engine Scope Bindings of the ScriptEngine must be identical to the Engine Scope Bindings of its default ScriptContext.
- The Global Scope Bindings of the ScriptEngine must be identical to the Global Scope Bindings of its default ScriptContext.

The ScriptEngine interface also has

```
Object put(String key, Object value)

void get(String key)
```

methods that must map directly to

```
getBindings(ScriptContext.ENGINE_SCOPE).put(String key,
Object value)

getBindings(ScriptContext.ENGINE_SCOPE).get(String key)
```

The setBindings methods of ScriptEngine and ScriptContext must accept any implementation of the Bindings interface. However, some ScriptEngine implementations might have Bindings implementations, which are preferred for one reason or another. For example, a ScriptEngine might provide a Bindings implemented using internal constructs in an underlying script interpreter. The

```
Bindings createBindings()
```

method of the ScriptEngine interface is provided to allow an implementation to return a Bindings of its preferred type.

Most of the key/value pairs in an Engine Scope Bindings represent bindings of script variable names to the values of the variables. The other class of key/value pairs consists of ones whose values are assigned special meanings by this specification.

The reserved keys for these pairs are:

Key	***Meaning of Value***
javax.script.argv	An Object[] used to pass a set of positional parameters to the ScriptEngine where this is meaningful and appropriate.
javax.script.filename	The name of the file or resource containing the source of the current script.
javax.script.engine	The name of the Java Script Engine. Determined by the implementor.
javax.script.engine_version	The version of the Java Script Engine.
javax.script.language	The name of the Language supported by the Java Script Engine.
javax.script.language_version	The version of the scripting language supported by the Java Script Engine.

There is no requirement that an Engine Scope Bindings contain mappings of all these keys. Implementations may define additional reserved keys. However, the use of keys beginning with javax.script

is reserved for future versions of this specification.

SCR.4.3.4.1.2 Script Execution

The ScriptEngine interface has methods:

```
Object eval(String script)

Object eval(Reader reader)

Object eval(String script, Bindings bindings)

Object eval(Reader reader, Bindings bindings)

Object eval(String script, ScriptContext context)

Object eval(Reader reader, ScriptContext context)
```

that execute scripts. The source of the script is passed as the first argument and the ScriptContext used during ScriptExecution is determined by the second argument. To say that a ScriptContext is used by a ScriptEngine during a script execution means that:

- The value of each key in the ScriptContext's Engine Scope Bindings that is not assigned a special meaning must be accessible from the script. Usually, each key corresponds to a scripting variable. The value associated with the key in the Engine Scope Bindings and the value of the scripting variable must be the same. In some cases, a key in the Engine Scope Bindings and the name of its associated scripting variable may not be the same due to requirements of the scripting language. In these cases an implementor should document the scheme for deriving a scripting variable name from an Engine Scope Bindings key.

- The Reader and Writers returned by the methods of the ScriptContext may be used by scripting constructs that read input and display output.

In the **eval** methods taking a single argument, the default ScriptContext of the ScriptEngine is used during execution of the script.

In the **eval** methods where a Bindings is passed as the second argument, the Engine Scope of the ScriptContext used during script execution must be the Bindings specified in the second argument. The Global Scope Bindings, Reader, Writer and Error Writer of the ScriptContext used during execution must be identical to those of the default ScriptContext.

The **eval** methods taking a ScriptContext as a second argument must use that ScriptContext during the execution of the script.

In no case should the ScriptContext used during ScriptExecution permanently replace the default ScriptContext of the ScriptEngine. If the Engine Scope Bindings of the ScriptContext used during execution is different from the Engine Scope of the ScriptEngine, the mappings in the Engine Scope of the ScriptEngine should not be altered as a side-effect of script execution.

In all cases, the ScriptContext used during a script execution must be a value in the Engine Scope of the ScriptEngine whose key is the String **“context”.**

SCR.4.3.4.1.3 Global Scope

A ScriptEngineManager must initialize the Global Scope Bindings of every ScriptEngine it creates using the setBindings method of the ScriptEngine interface, so calls to :

```
ScriptEngine.getBindings(ScriptContext.GLOBAL_SCOPE)
```

or

```
ScriptContext.getBindings(ScriptContext.GLOBAL_SCOPE)
```

on the default ScriptContext of the ScriptEngine will initially return the Global Scope Bindings of the ScriptEngineManager. A different Global Scope Bindings can be returned only if a different one has been passed to the setBindings method of either the ScriptEngine or its default ScriptContext passing ScriptContext.GLOBAL_SCOPE as the scope argument.

Even in applications not using a ScriptEngineManager, it might be appropriate to maintain a global scope of attributes visible in all ScriptEngines, and the

```
void setBindings(int scope)
```

can be used with the

```
ScriptContext.GLOBAL_SCOPE
```

argument to set references to that Bindings in each ScriptEngine.

It is the responsibility of the application programmer to ensure that the same Bindings appears as the global scope of every ScriptEngine. In other words,

```
setBindings(Bindings bindings, ScriptContext.GLOBAL_SCOPE)
```

is not required to set the Global Scope Bindings in any ScriptEngine other than the one on which it was called.

The keys in the Global Scope Bindings of a ScriptEngine may be exposed as scripting variables during script executions, but this is not required.

SCR.4.3.4.1.4 Metadata

Each Java Script Engine implementation must have an accompanying implementation of ScriptEngineFactory, an interface that include methods which describe implementation-dependent features of the Java Script Engine. The ScriptEngine interface has a:

```
ScriptEngineFactory getFactory()
```

method which must return an instance of its associated ScriptEngineFactory.

SCR.4.3.4.2 Compilable

The optional Compilable interface is implemented by Java Script Engines that support the re-execution of intermediate code retained from previous script compilations. The interface has two methods:

```
CompiledScript compile(String str)

CompiledScript compile(Reader reader)
```

that return implementations of the CompiledScript interface. That interface is an abstraction for the intermediate code produced by script compilations, and has methods:

```
Object eval()

Object eval(Bindings bindings)

Object eval(ScriptContext context)
```

analogous to the eval methods of ScriptEngine. A CompiledScript is associated to the ScriptEngine in which it was compiled and shares a default ScriptContext with its associated ScriptEngine. The rules determining the ScriptContext to be used during script execution for a

CompiledScript are identical to those specified for the eval methods of the ScriptEngine interface.

If a Java Script Engine implements Compilable, it can be assumed that intermediate code associated with every CompiledScript instance is unaffected by other script executions or compilations performed by the same Java Script Engine. In other words, the CompiledScript will always represent exactly the same program, even after other scripts have been executed or compiled by the same interpreter.

SCR.4.3.4.3 Invocable

The methods of the optional Invocable interface allow procedures/functions/methods in scripts that have been compiled by the front-end of a ScriptEngine to be invoked directly from Java code in an application. The procedures may be top-level subroutines defined in scripts, or in cases where the scripting languages support objects, may be objects defined in a script. The

```
Object invokeFunction(String methodName , Object... args)

Object invokeMethod(Object thiz, String methodName,
Object... args)
```

methods invoke a script procedure with the given name. The invokeFunction method is for top-level procedures. The first argument of invokeMethod is a reference to an object defined in the script. he specified procedure is called using the argument as its “this” reference.

Each element of the varargs argument is passed to the procedure according to the conventions for conversion of arguments in the engine's implementation of Java Language Bindings. If the procedure returns a value, it is converted to a Java object according to the conventions for the Java Language Bindings and returned to the caller in the Java Application.

The invokeFunction and invokeMethod methods must throw ScriptExceptions if invocation of the procedure or method throws a checked Exception in the underlying interpreter. It must throw a NoSuchMethodException if it can be determined that no procedure with the given name is defined in a script compiled in the ScriptEngine. In other words, if the call method returns successfully:

- a procedure with the specified name exists in the state of the interpreter
- the procedure was invoked and executed without error when passed the converted form of the arguments

If the attempt to invoke the procedure in the interpreter fails, the

resulting ScriptException should return data describing the cause of the failure.

The third method of Invocable

```
<T> T getInterface(Class<T> clasz)
```

returns an instance of a Java class whose methods are implemented using top-level procedures in a script represented in intermediate code in an interpreter. The argument is an interface that the returned class must implement.

The similar

```
<T> T getInterface(Object scriptObject, Class<T> clasz)
```

method returns an instance of the specified interface whose methods are implemented using instance methods of the specified script object.

The getInterface methods can always be implemented using a java.lang.reflect.Proxy whose InvocationHandler uses the corresponding invokeFunction or invokeMethod method.

The invokeMethod method and the version of the getInterface method that takes an Object parameter may throw an UnsupportedOperationException if the functionality is not available in the underlying interpreter.

SCR.4.3.5 ScriptEngineFactory

SCR.4.3.5.1 Factory Method

Instances of ScriptEngineFactory are the Objects located by the Script Engine Discovery Mechanism.

The method

```
ScriptEngine getScriptEngine()
```

is the ScriptEngineFactory method used to instantiate a ScriptEngine. Before invoking the getScriptEngine method, the caller can one or more of the metadata methods to determine implementation specific characteristics of the associated ScriptEngine.

SCR.4.3.5.2 Metadata Methods

Every ScriptEngine must have a corresponding implementation of ScriptEngineInfo, whose methods return data describing the implementation. The methods will generally return hard-coded values.

The methods

```
String getEngineName()

String getEngineVersion()

String getLanguage()

String getLanguageVersion()
```

return Strings whose meanings are described in the discussion of the corresponding reserved keys for key/value pairs in the ScriptEngine interface.

The

```
List<String> getExtensions()
```

returns an immutable List of Strings that are file extensions typically used for files containing scripts written in the languages supported by the Java Script Engine.

The

```
List<String> getMimeTypes()
```

method returns an immutable List of strings containing mime types describing content which can be processed using the Java Script Engine.

The

```
List<String> getNames()
```

method returns an immutable List of short descriptive names such as {“javascript” , “rhino”} describing the language supported by the JavaScriptEngine. These names are used as keys in the method

```
ScriptEngine ScriptEngineManager.getEngineByName(String key)
```

used in the ScriptEngine Discovery process.

The

```
Object getParameter(String key)
```

method allows the ScriptEngineInfo to be queried for the expected behavior of the ScriptEngine with regard to certain behaviors that are implementation-specific. Implementations may also define implementation-specific keys and specify the meanings of the return values for those keys. The following table lists the keys defined in this specification and provides interpretations of the return values.

Name	***Type of Value***	***Meaning of Value***
ENGINE	java.lang.String	same as getEngine()
ENGINE_VERSION	java.lang.String	same as getEngineVersion()
NAME	java.lang.String	same as getName()
LANGUAGE	java.lang.String	same as getLanguage()
LANGUAGE_VERSION	java.lang.String	same as getLanguageVersion()
THREADING	java.lang.String	null – Non threadsafe engine. “MULTITHREADED” - Multithreaded Engine (see below) “THREAD-ISOLATED” - Thread-isolated engine (see below) “STATELESS” - Stateless engine (see below)

Multithreaded Engine

Multi-threaded Evaluation - The implementation of the API and the engine itself are capable of supporting concurrent evaluations by multiple threads on a single engine instance. However the exact behavior of such concurrent execution is ultimately determined by the script or scripts themselves. An engine which supports concurrent execution is "multi-threaded" in the same sense that the Java language is "multi-threaded": Evaluation by concurrent threads is allowed and produces side effects that can be seen by other threads. The threads may interact or not interact, utilize synchronization or not utilize synchronization, in scripting language dependent ways.

Thread-Isolated Engine

Satisfies the requirements for Multithreaded. Also, the side-effects for threads are isolated from one another. Specifically, the isolation limits the visibility of changes to the state of variables in the engine scope of the interpreter. Each thread will effectively have its own thread-local engine scope for the engine instance. Note that this does not limit the visibility of side effects outside the engine scope – for example, mutation of application-level Java objects.

Stateless Engine

Satisfies the requirements for Thread-Isolated. In addition, the mappings in the Bindings used as the engine scope of the ScriptEngine are not modified by any ScriptExecution. The keys in the Bindings and their associated values are exactly the same before and after each script execution.

SCR.4.3.5.3 Script Generation Methods

The ScriptEngineFactory interface also includes several methods returning Strings that can be used as executable script code by a ScriptEngine described by the ScriptEngineFactory. The methods can be used by applications hosting ScriptEngines to record macros that can be executed later. These methods are:

```
String getOutputStatement(String toDisplay)

String getMethodCallSyntax(String obj,
                           String m,
                           String... args)

String getProgram(String... statements)
```

The getOutputStatement method returns a string that can be used as a script statement that outputs a given string. For instance, for a PHP ScriptEngine,

```
getOutputStatement(“hello”)
```

might return

```
“echo(\”hello\”);”
```

The getMethodCallSyntax method returns a String that can be used as a script statement to call a Java method on an Object represented by a scripting variable. The first argument is the name of the scripting

variable, the second argument is the name of the method, the third argument is an array of variable names representing Java Objects to be passed to the specified method. The actual names of the scripting variables may be decorated forms of the arguments depending on the conventions of the scripting language. The arguments should be of the form used in the Bindings APIs to create the Java Language Bindings. For example, for a PHP ScriptEngine,

```
getMethodCallSyntax(“x”, “someMethod”,
                    new String[]{“a”, “b”})
```

might return

```
“$x->someMethod($a, $b);”
```

since by convention, PHP scripting variables begin with the '$' character.

The getProgram method returns a String that is an executable program whose statements are the elements of the variable argument list of Strings passed to the method. These statements might have been returned by calls to getMethodCallSyntax and getOutputStatement methods. The getProgram method supplies required program components. For instance, for a PHP ScriptEngine,

```
getProgram(“$x->someMethod()”,”echo(\“hello\”)”)
```

might return

```
“<? $x->someMethod();echo(\“hello\”); ?>”
```

since in PHP, script code must be enclided in **<? ... ?>** blocks and statements must be delineated by semicolons.

SCR.4.3.6 AbstractScriptEngine

The javax.script package includes an abstract class, AbstractScriptEngine that implements ScriptEngine and provides default implementations of

```
Object eval(String script)

Object eval(String script, Bindings bindings)

Object eval(Reader reader)

Object eval(Reader reader, Bindings bindings)
```

using the abstract methods.

```
Object eval(String script, ScriptContext context)

Object eval(Reader reader, ScriptContext context)
```

The initial default ScriptContext is a SimpleScriptContext.

SCR.4.3.7 ScriptException

ScriptException is the generic checked exception thrown by methods of the Scripting API. Checked exceptions thrown by underlying interpreters must be caught and used to initialize ScriptExceptions.

The constructor:

```
ScriptException(Exception e)
```

is be used to wrap checked exceptions thrown by underlying interpreters.

If exceptions thrown by the underlying scripting implementations provide pertinent line number, column number or source-file information, a ScriptException storing this information may be initialized using one of the constructors

```
ScriptException(String message, String source, int line)

ScriptException(String message, String source, int line,
int column)
```

The error message, source file and line number of a ScriptException can be accessed using their

```
String getMessage()

String getFileName()

int getLineNumber()

int getColumnNumber()
```

methods . If the line number or column number cannot be determined, `getLineNumber` or `getColumnNumber` must return -1. If no description of the script source is available, an application-defined value such as *"<unknown>"* may be returned by `getFileName`.

SCR.4.3.8 ScriptEngineManager

The concrete ScriptEngineManager class is used to implement the Discovery Mechanism for Java Script Engines. Each ScriptEngineManager maintains the Global Scope Bindings of key/value pairs that it makes available to every ScriptEngine it creates.

SCR.4.3.8.1 Discovery Mechanism

The ScriptEngineManager class implements the Discovery Mechanism described in the discussion of Functional Components of the Scripting API. The

List<ScriptEngineFactory> getEngineFactories()

method returns an immutable List of instances of all the ScriptEngineFactory classes that have been packaged in Jar files according to this specification, whose Jar file resources are visible in the current ClassLoader. An application can determine the attributes of each of the ScriptEngines by using the metadata methods of the ScriptEngineFactories in the array.

The ScriptEngineManager also has methods that search this List for a ScriptEngineFactory with attributes matching specified criteria. If found, the methods use the ScriptEngineFactory to create an instance of its associated ScriptEngine class.

These methods are:

```
ScriptEngine getEngineByExtension(String extension)

ScriptEngine getEngineByMimeType(String mimeType)

ScriptEngine getEngineByName(String name)
```

In each case, the ScriptEngineFactory found first in the List returned by the getEngineFactories method whose metadata value matches the attribute specified in the first argument is returned. In all cases, the String comparison is case-sensitive and extensions do not include initial dots.

In the cases of mimeType and extension attributes, methods

```
void registerEngineMimeType(String mimeType,
                            ScriptEngineFactory factory)
void registerEngineExtension(String extension,
                            ScriptEngineFactory factory)
```

are provided to customize the Discovery Mechanism. If a specified ScriptEngineFactory has been registered to handle a specified mimeType (extension), the getEngineByMimeType (getEngineByExtension) returns the registered ScriptEngineFactory rather than the one located by the usual means.

SCR.4.3.8.2 Global Scope

Each ScriptEngineManager stores a Bindings of key/value pairs, which is returned by the

```
Bindings getBindings()
```

method. This Bindings is used to initialize the Global Scope Bindings of every ScriptEngine created by the ScriptEngineManager using the

```
void setBindings(Bindings bindings, int scope)
```

of the ScriptEngine, passing

```
ScriptContext.GLOBAL_SCOPE
```

as the scope parameter.

ScriptEngineManager also has

```
void put(String key, Object value)

Object get(String key)
```

methods that map directly to its:

```
getBindings().put(String key, Object value)

getBindings().get(String key)
```

methods.

Multiple ScriptEngineManager instances may exist in an application, so multiple Global Scopes might exist. ScriptEngines associated with a particular ScriptEngineManager might not have access to the Global Scope maintained by another ScriptEngineManager.

-

SCR.5
Package javax.script

Interfaces	
Compilable	The optional interface implemented by ScriptEngines able to compile scripts to a form that can be executed repeatedly without recompilation.
Invocable	The optional interface implemented by ScriptEngines whose methods allow the invocation of procedures in scripts that have already been executed.
Bindings	A mapping of key/value pairs, all of whose keys are `Strings`.
ScriptContext	The interface whose implementing classes are used to connect Script Engines with objects, such as scoped Bindings, in hosting applications.
ScriptEngine	`ScriptEngine` is the fundamental interface whose methods must be fully functional in every implementation of this specification.
ScriptEngineFactory	`ScriptEngineFactory` is used to describe and instantiate `ScriptEngine` instances.

Class Summary	
CompiledScript	Extended by classes that store results of compilations.
SimpleScriptContext	Simple implementation of ScriptContext.
AbstractScriptEngine	Provides a standard implementation for several of the variants of the `eval` method.
ScriptEngineManager	The `ScriptEngineManager` implements a discovery and instantiation mechanism for `ScriptEngine` classes and also maintains a

	collection of key/value pairs storing state shared by all engines created by the Manager.
SimpleBindings	Simple implementation of Namespace backed by a `HashMap` or some other specified `Map`.

Exception Summary	
ScriptException	Generic Exception class for the Scripting API

SCR.5.1 Compilable

javax.script

Interface Compilable

public interface Compilable

The optional interface implemented by ScriptEngines whose methods compile scripts to a form that can be executed repeatedly without recompilation.

Since:
1.6

SCR.5.1.1 Methods

compile

```
CompiledScript compile(java.lang.String script)
                       throws ScriptException
```

Compiles the script (source represented as a String) for later execution.

Parameters:
script - The source of the script, represented as a String.

Returns
An subclass of CompiledScript to be executed later using one of the eval methods of CompiledScript.

Throws
ScriptException - if compilation fails.
java.lang.NullPointerException - if the argument is null.

compile

```
CompiledScript compile(java.io.Reader script)
                       throws ScriptException
```

Compiles the script (source read from Reader) for later execution. Functionality is identical to compile(String) other than the way in which the source is passed.

Parameters:
script - The reader from which the script source is obtained.

Returns
An implementation of CompiledScript to be executed later using one of its eval methods of CompiledScript.

Throws
ScriptException - if compilation fails.
java.lang.NullPointerException - if argument is null.

SCR.5.2 Invocable

javax.script

Interface Invocable

public interface Invocable

The optional interface implemented by ScriptEngines whose methods allow the invocation of procedures in scripts that have previously been executed.

Since:
1.6

getInterface(java.lang.Class<T> clasz)
Returns an implementation of an interface using functions compiled in the interpreter. **getInterface**(java.lang.Object thiz, java.lang.Class<T> clasz)

Returns an implementation of an interface using member functions of a scripting object compiled in the interpreter. `java.lang.Object` **`invokeFunction`**`(java.lang.String name, java.lang.Object... args)`
Used to call top-level procedures and functions defined in scripts.
`java.lang.Object` **`invokeMethod`**`(java.lang.Object thiz, java.lang.String name, java.lang.Object... args)`
Calls a method on a script object compiled during a previous script execution, which is retained in the state of the `ScriptEngine`. -->

SCR.5.2.1 Methods

invokeMethod

```
java.lang.Object invokeMethod(java.lang.Object thiz,
                              java.lang.String name,
                              java.lang.Object... args)
                              throws ScriptException,
                                     java.lang.NoSuchMethodExcept
ion
```

Calls a method on a script object compiled during a previous script execution, which is retained in the state of the ScriptEngine.

Parameters:

name - The name of the procedure to be called.
thiz - If the procedure is a member of a class defined in the script and thiz is an instance of that class returned by a previous execution or invocation, the named method is called through that instance.
args - Arguments to pass to the procedure. The rules for converting the arguments to scripting variables are implementation-specific.

Returns

The value returned by the procedure. The rules for converting the scripting variable returned by the script method to a Java Object are implementation-specific.

Throws

ScriptException - if an error occurrs during invocation

of the method.
java.lang.NoSuchMethodException - if method with given name or matching argument types cannot be found.
java.lang.NullPointerException - if the method name is null.
java.lang.IllegalArgumentException - if the specified thiz is null or the specified Object is does not represent a scripting object.

invokeFunction

```
java.lang.Object invokeFunction(java.lang.String name,
                                java.lang.Object... args)
                                throws ScriptException,
                                       java.lang.NoSuchMethodExce
ption
```

Used to call top-level procedures and functions defined in scripts.

Parameters:
args - Arguments to pass to the procedure or function

Returns
The value returned by the procedure or function

Throws
ScriptException - if an error occurrs during invocation of the method.
java.lang.NoSuchMethodException - if method with given name or matching argument types cannot be found.
java.lang.NullPointerException - if method name is null.

getInterface

```
<T> T getInterface(java.lang.Class<T> clasz)
```

Returns an implementation of an interface using functions compiled in the interpreter. The methods of the interface may be implemented using the invokeFunction method.

Parameters:

clasz - The Class object of the interface to return.

Returns

An instance of requested interface - null if the requested interface is unavailable, i. e. if compiled functions in the ScriptEngine cannot be found matching the ones in the requested interface.

Throws

java.lang.IllegalArgumentException - if the specified Class object is null or is not an interface.

getInterface

```
<T> T getInterface(java.lang.Object thiz,
                   java.lang.Class<T> clasz)
```

Returns an implementation of an interface using member functions of a scripting object compiled in the interpreter. The methods of the interface may be implemented using the invokeMethod method.

Parameters:

thiz - The scripting object whose member functions are used to implement the methods of the interface.
clasz - The Class object of the interface to return.

Returns

An instance of requested interface - null if the requested interface is unavailable, i. e. if compiled methods in the ScriptEngine cannot be found matching the ones in the requested interface.

Throws

java.lang.IllegalArgumentException - if the specified Class object is null or is not an interface, or if the specified Object is null or does not represent a scripting object.

SCR.5.3 Bindings

javax.script

Interface Bindings

All Superinterfaces:
java.util.Map<java.lang.String,java.lang.Object>

All Known Implementing Classes:
SimpleBindings

public interface Bindingsextends
java.util.Map<java.lang.String,java.lang.Object>

A mapping of key/value pairs, all of whose keys are Strings.

Since:
1.6

SCR.5.3.1 Methods

put

```
java.lang.Object put(java.lang.String name,
                     java.lang.Object value)
```

Set a named value.

Specified by:
put in interface
java.util.Map<java.lang.String,java.lang.Object>

Parameters:
name - The name associated with the value.
value - The value associated with the name.

Returns
The value previously associated with the given name.
Returns null if no value was previously associated with
the name.

Throws
java.lang.NullPointerException - if the name is null.
java.lang.IllegalArgumentException - if the name is empty String.

putAll

void **putAll**(java.util.Map<? extends java.lang.String,? extends java.lang.Object> toMerge)

Adds all the mappings in a given Map to this Bindings

Specified by:
putAll in interface java.util.Map<java.lang.String,java.lang.Object>
Parameters:
toMerge - The Map to merge with this one.

Throws
java.lang.NullPointerException - if toMerge map is null or if some key in the map is null.
java.lang.IllegalArgumentException - if some key in the map is an empty String.

containsKey

boolean **containsKey**(java.lang.Object key)

Returns true if this map contains a mapping for the specified key. More formally, returns true if and only if this map contains a mapping for a key k such that (key==null ? k==null : key.equals(k)). (There can be at most one such mapping.)

Specified by:
containsKey in interface java.util.Map<java.lang.String,java.lang.Object>
Parameters:
key - key whose presence in this map is to be tested.

Returns
true if this map contains a mapping for the specified key.

Throws

java.lang.NullPointerException - if key is null
java.lang.ClassCastException - if key is not String
java.lang.IllegalArgumentException - if key is empty String

get

java.lang.Object **get**(java.lang.Object key)

Returns the value to which this map maps the specified key. Returns null if the map contains no mapping for this key. A return value of null does not *necessarily* indicate that the map contains no mapping for the key; it's also possible that the map explicitly maps the key to null. The containsKey operation may be used to distinguish these two cases.

More formally, if this map contains a mapping from a key k to a value v such that (key==null ? k==null : key.equals(k)), then this method returns v; otherwise it returns null. (There can be at most one such mapping.)

Specified by:

get in interface java.util.Map<java.lang.String,java.lang.Object>

Parameters:

key - key whose associated value is to be returned.

Returns

the value to which this map maps the specified key, or null if the map contains no mapping for this key.

Throws

java.lang.NullPointerException - if key is null
java.lang.ClassCastException - if key is not String
java.lang.IllegalArgumentException - if key is empty String

remove

java.lang.Object **remove**(java.lang.Object key)

Removes the mapping for this key from this map if it is present (optional operation). More formally, if this map contains a mapping from key k to value v such that (key==null ? k==null : key.equals(k)), that mapping is removed. (The map can contain at most one such mapping.)

Returns the value to which the map previously associated the key, or null if the map contained no mapping for this key. (A null return can also indicate that the map previously associated null with the specified key if the implementation supports null values.) The map will not contain a mapping for the specified key once the call returns.

Specified by:
remove in interface java.util.Map<java.lang.String,java.lang.Object>

Parameters:
key - key whose mapping is to be removed from the map.

Returns
previous value associated with specified key, or null if there was no mapping for key.

Throws
java.lang.NullPointerException - if key is null
java.lang.ClassCastException - if key is not String
java.lang.IllegalArgumentException - if key is empty String

SCR.5.4 ScriptContext

javax.script

Interface ScriptContext

All Known Implementing Classes:
SimpleScriptContext

public interface ScriptContext

The interface whose implementing classes are used to connect Script Engines with objects, such as scoped Bindings, in hosting applications. Each scope is a set of named attributes whose values can be set and retrieved using the `ScriptContext` methods. ScriptContexts also expose Readers and Writers that can be used by the ScriptEngines for input and output.

Since:
1.6

SCR.5.4.1 Fields

ENGINE_SCOPE

`static final int` **ENGINE_SCOPE**

EngineScope attributes are visible during the lifetime of a single ScriptEngine and a set of attributes is maintained for each engine.

GLOBAL_SCOPE

`static final int` **GLOBAL_SCOPE**

GlobalScope attributes are visible to all engines created by same ScriptEngineFactory.

SCR.5.4.2 Methods

setBindings

```
void setBindings(Bindings bindings,
                 int scope)
```

Associates a Bindings instance with a particular scope in this ScriptContext. Calls to the getAttribute and setAttribute methods must map to the get and put methods of the Bindings for the specified scope.

Parameters:

bindings - The Bindings to associate with the given scope
scope - The scope

Throws

java.lang.IllegalArgumentException - If no Bindings is defined for the specified scope value in ScriptContexts of this type.
java.lang.NullPointerException - if value of scope is ENGINE_SCOPE and the specified Bindings is null.

getBindings

```
Bindings getBindings(int scope)
```

Gets the Bindings associated with the given scope in this ScriptContext.

Returns

The associated Bindings. Returns null if it has not been set.

Throws

java.lang.IllegalArgumentException - If no Bindings is defined for the specified scope value in ScriptContext of this type.

setAttribute

```
void setAttribute(java.lang.String name,
                  java.lang.Object value,
                  int scope)
```

Sets the value of an attribute in a given scope.

Parameters:

name - The name of the attribute to set
value - The value of the attribute
scope - The scope in which to set the attribute

Throws

java.lang.IllegalArgumentException - if the name is empty or if the scope is invalid.
java.lang.NullPointerException - if the name is null.

getAttribute

```
java.lang.Object getAttribute(java.lang.String name,
                              int scope)
```

Gets the value of an attribute in a given scope.

Parameters:

name - The name of the attribute to retrieve.
scope - The scope in which to retrieve the attribute.

Returns

The value of the attribute. Returns null is the name does not exist in the given scope.

Throws

java.lang.IllegalArgumentException - if the name is empty or if the value of scope is invalid.
java.lang.NullPointerException - if the name is null.

removeAttribute

java.lang.Object **removeAttribute**(java.lang.String name,
int scope)

Remove an attribute in a given scope.

Parameters:
name - The name of the attribute to remove
scope - The scope in which to remove the attribute

Returns
The removed value.

Throws
java.lang.IllegalArgumentException - if the name is empty or if the scope is invalid.
java.lang.NullPointerException - if the name is null.

getAttribute

java.lang.Object **getAttribute**(java.lang.String name)

Retrieves the value of the attribute with the given name in the scope occurring earliest in the search order. The order is determined by the numeric value of the scope parameter (lowest scope values first.)

Parameters:
name - The name of the the attribute to retrieve.

Returns
The value of the attribute in the lowest scope for which an attribute with the given name is defined. Returns null if no attribute with the name exists in any scope.

Throws
java.lang.NullPointerException - if the name is null.
java.lang.IllegalArgumentException - if the name is empty.

getAttributesScope

int **getAttributesScope**(java.lang.String name)

Get the lowest scope in which an attribute is defined.

Parameters:
name - Name of the attribute .

Returns
The lowest scope. Returns -1 if no attribute with the given name is defined in any scope.

Throws
java.lang.NullPointerException - if name is null.
java.lang.IllegalArgumentException - if name is empty.

getWriter

java.io.Writer **getWriter**()

Returns the Writer for scripts to use when displaying output.

Returns
The Writer.

getErrorWriter

java.io.Writer **getErrorWriter**()

Returns the Writer used to display error output.

Returns
The Writer

setWriter

void **setWriter**(java.io.Writer writer)

Sets the Writer for scripts to use when displaying output.

Parameters:
writer - The new Writer.

setErrorWriter

void **setErrorWriter**(java.io.Writer writer)

Sets the Writer used to display error output.

Parameters:
writer - The Writer.

getReader

java.io.Reader **getReader**()

Returns a Reader to be used by the script to read input.

Returns
The Reader.

setReader

void **setReader**(java.io.Reader reader)

Sets the Reader for scripts to read input .

Parameters:
reader - The new Reader.

getScopes

java.util.List<java.lang.Integer> **getScopes**()

Returns immutable List of all the valid values for scope in the ScriptContext.

Returns
list of scope values

SCR.5.5 ScriptEngine

javax.script

Interface ScriptEngine

All Known Implementing Classes:

AbstractScriptEngine

public interface ScriptEngine

ScriptEngine is the fundamental interface whose methods must be fully functional in every implementation of this specification.

These methods provide basic scripting functionality. Applications written to this simple interface are expected to work with minimal modifications in every implementation. It includes methods that execute scripts, and ones that set and get values.

The values are key/value pairs of two types. The first type of pairs consists of those whose keys are reserved and defined in this specification or by individual implementations. The values in the pairs with reserved keys have specified meanings.

The other type of pairs consists of those that create Java language Bindings, the values are usually represented in scripts by the corresponding keys or by decorated forms of them.

Since:

1.6

SCR.5.5.1 Fields

ARGV

static final java.lang.String **ARGV**

Reserved key for a named value that passes an array of positional arguments to a script.

FILENAME

static final java.lang.String **FILENAME**

Reserved key for a named value that is the name of the file being executed.

ENGINE

static final java.lang.String **ENGINE**

Reserved key for a named value that is the name of the ScriptEngine implementation.

ENGINE_VERSION

static final java.lang.String **ENGINE_VERSION**

Reserved key for a named value that identifies the version of the ScriptEngine implementation.

NAME

static final java.lang.String **NAME**

Reserved key for a named value that identifies the short name of the scripting language. The name is used by the ScriptEngineManager to locate a ScriptEngine with a given name in the getEngineByName method.

LANGUAGE

static final java.lang.String **LANGUAGE**

Reserved key for a named value that is the full name of Scripting Language supported by the implementation.

LANGUAGE_VERSION

static final java.lang.String **LANGUAGE_VERSION**

Reserved key for the named value that identifies the version of the scripting language supported by the implementation.

SCR.5.5.2 Methods

eval

java.lang.Object **eval**(java.lang.String script,
ScriptContext context)
throws ScriptException

Causes the immediate execution of the script whose source is the String passed as the first argument. The script may be reparsed or recompiled before execution. State left in the engine from previous executions, including variable values and compiled procedures may be visible during this execution.

Parameters:

script - The script to be executed by the script engine.
context - A ScriptContext exposing sets of attributes in different scopes. The meanings of the scopes ScriptContext.GLOBAL_SCOPE, and ScriptContext.ENGINE_SCOPE are defined in the specification.

The ENGINE_SCOPE Bindings of the ScriptContext contains the bindings of scripting variables to application objects to be used during this script execution.

Returns

The value returned from the execution of the script.

Throws

ScriptException - if an error occurrs in script. ScriptEngines should create and throw ScriptException wrappers for checked Exceptions thrown by underlying scripting implementations.
java.lang.NullPointerException - if either argument is null.

eval

```
java.lang.Object eval(java.io.Reader reader,
                      ScriptContext context)
                      throws ScriptException
```

Same as eval(String, ScriptContext) where the source of the script is read from a Reader.

Parameters:

reader - The source of the script to be executed by the script engine.
context - The ScriptContext passed to the script engine.

Returns

The value returned from the execution of the script.

Throws

ScriptException - if an error occurrs in script.
java.lang.NullPointerException - if either argument is null.

eval

```
java.lang.Object eval(java.lang.String script)
                      throws ScriptException
```

Executes the specified script. The default ScriptContext for the ScriptEngine is used.

Parameters:

script - The script language source to be executed.

Returns

The value returned from the execution of the script.

Throws

ScriptException - if error occurrs in script.
java.lang.NullPointerException - if the argument is null.

eval

```
java.lang.Object eval(java.io.Reader reader)
                      throws ScriptException
```

Same as eval(String) except that the source of the script is provided as a Reader

Parameters:
reader - The source of the script.

Returns
The value returned by the script.

Throws
ScriptException - if an error occurrs in script.
java.lang.NullPointerException - if the argument is null.

eval

```
java.lang.Object eval(java.lang.String script,
                      Bindings n)
                      throws ScriptException
```

Executes the script using the Bindings argument as the ENGINE_SCOPE Bindings of the ScriptEngine during the script execution. The Reader, Writer and non-ENGINE_SCOPE Bindings of the default ScriptContext are used. The ENGINE_SCOPE Bindings of the ScriptEngine is not changed, and its mappings are unaltered by the script execution.

Parameters:
script - The source for the script.
n - The Bindings of attributes to be used for script execution.

Returns
The value returned by the script.

Throws

ScriptException - if an error occurrs in script.
java.lang.NullPointerException - if either argument is null.

eval

```
java.lang.Object eval(java.io.Reader reader,
                      Bindings n)
                      throws ScriptException
```

Same as eval(String, Bindings) except that the source of the script is provided as a Reader.

Parameters:

reader - The source of the script.
n - The Bindings of attributes.

Returns

The value returned by the script.

Throws

ScriptException - if an error occurrs.
java.lang.NullPointerException - if either argument is null.

put

```
void put(java.lang.String key,
         java.lang.Object value)
```

Sets a key/value pair in the state of the ScriptEngine that may either create a Java Language Binding to be used in the execution of scripts or be used in some other way, depending on whether the key is reserved. Must have the same effect as getBindings(ScriptContext.ENGINE_SCOPE).put.

Parameters:

key - The name of named value to add
value - The value of named value to add.

Throws
java.lang.NullPointerException - if key is null.
java.lang.IllegalArgumentException - if key is empty.

get

java.lang.Object **get**(java.lang.String key)

Retrieves a value set in the state of this engine. The value might be one which was set using setValue or some other value in the state of the ScriptEngine, depending on the implementation. Must have the same effect as getBindings(ScriptContext.ENGINE_SCOPE).get

Parameters:
key - The key whose value is to be returned

Returns
the value for the given key

Throws
java.lang.NullPointerException - if key is null.
java.lang.IllegalArgumentException - if key is empty.

getBindings

Bindings **getBindings**(int scope)

Returns a scope of named values. The possible scopes are:

- ScriptContext.GLOBAL_SCOPE - The set of named values representing global scope. If this ScriptEngine is created by a ScriptEngineManager, then the manager sets global scope bindings. This may be null if no global scope is associated with this ScriptEngine
- ScriptContext.ENGINE_SCOPE - The set of named values representing the state of this ScriptEngine. The values are generally visible in scripts using the associated keys as variable names.
- Any other value of scope defined in the default ScriptContext of the ScriptEngine.

The Bindings instances that are returned must be identical to those returned by the getBindings method of ScriptContext called with corresponding arguments on the default ScriptContext of the ScriptEngine.

Parameters:
scope - Either ScriptContext.ENGINE_SCOPE or ScriptContext.GLOBAL_SCOPE which specifies the Bindings to return. Implementations of ScriptContext may define additional scopes. If the default ScriptContext of the ScriptEngine defines additional scopes, any of them can be passed to get the corresponding Bindings.

Returns
The Bindings with the specified scope.

Throws
java.lang.IllegalArgumentException - if specified scope is invalid

setBindings

void **setBindings**(Bindings bindings,
int scope)

Sets a scope of named values to be used by scripts. The possible scopes are:

- ScriptContext.ENGINE_SCOPE - The specified Bindings replaces the engine scope of the ScriptEngine.
- ScriptContext.GLOBAL_SCOPE - The specified Bindings must be visible as the GLOBAL_SCOPE.
- Any other value of scope defined in the default ScriptContext of the ScriptEngine.

The method must have the same effect as calling the setBindings method of ScriptContext with the corresponding value of scope on the default ScriptContext of the ScriptEngine.

Parameters:
bindings - The Bindings for the specified scope.

scope - The specified scope. Either ScriptContext.ENGINE_SCOPE, ScriptContext.GLOBAL_SCOPE, or any other valid value of scope.

Throws

java.lang.IllegalArgumentException - if the scope is invalid
java.lang.NullPointerException - if the bindings is null and the scope is ScriptContext.ENGINE_SCOPE

createBindings

Bindings **createBindings**()

Returns an uninitialized Bindings.

Returns

A Bindings that can be used to replace the state of this ScriptEngine.

getContext

ScriptContext **getContext**()

Returns the default ScriptContext of the ScriptEngine whose Bindings, Reader and Writers are used for script executions when no ScriptContext is specified.

Returns

The default ScriptContext of the ScriptEngine.

setContext

void **setContext**(ScriptContext context)

Sets the default ScriptContext of the ScriptEngine whose Bindings, Reader and Writers are used for script executions when no ScriptContext is specified.

Parameters:
context - A ScriptContext that will replace the default ScriptContext in the ScriptEngine.

Throws
java.lang.NullPointerException - if context is null.

getFactory

ScriptEngineFactory **getFactory**()

Returns a ScriptEngineFactory for the class to which this ScriptEngine belongs.

Returns
The ScriptEngineFactory

SCR.5.6 ScriptEngineFactory

javax.script

Interface ScriptEngineFactory

public interface ScriptEngineFactory

ScriptEngineFactory is used to describe and instantiate ScriptEngines.

Each class implementing ScriptEngine has a corresponding factory that exposes metadata describing the engine class.

The ScriptEngineManager uses the service provider mechanism described in the *Jar File Specification* to obtain instances of all

ScriptEngineFactories available in the current ClassLoader.

Since:
1.6

SCR.5.6.1 Methods

getEngineName

java.lang.String **getEngineName**()

Returns the full name of the ScriptEngine. For instance an implementation based on the Mozilla Rhino Javascript engine might return *Rhino Mozilla Javascript Engine*.

Returns
The name of the engine implementation.

getEngineVersion

java.lang.String **getEngineVersion**()

Returns the version of the ScriptEngine.

Returns
The ScriptEngine implementation version.

getExtensions

java.util.List<java.lang.String> **getExtensions**()

Returns an immutable list of filename extensions, which generally identify scripts written in the language supported by this ScriptEngine. The array is used by the ScriptEngineManager to implement its getEngineByExtension method.

Returns
The list of extensions.

getMimeTypes

java.util.List<java.lang.String> **getMimeTypes**()

Returns an immutable list of mimetypes, associated with scripts that can be executed by the engine. The list is used by the ScriptEngineManager class to implement its getEngineByMimetype method.

Returns
The list of mime types.

getNames

java.util.List<java.lang.String> **getNames**()

Returns an immutable list of short names for the ScriptEngine, which may be used to identify the ScriptEngine by the ScriptEngineManager. For instance, an implementation based on the Mozilla Rhino Javascript engine might return list containing {"javascript", "rhino"}.

getLanguageName

java.lang.String **getLanguageName**()

Returns the name of the scripting langauge supported by this ScriptEngine.

Returns
The name of the supported language.

getLanguageVersion

java.lang.String **getLanguageVersion**()

Returns the version of the scripting language supported by this ScriptEngine.

Returns

The version of the supported language.

getParameter

java.lang.Object **getParameter**(java.lang.String key)

Returns the value of an attribute whose meaning may be implementation-specific. Keys for which the value is defined in all implementations are:

- ScriptEngine.ENGINE
- ScriptEngine.ENGINE_VERSION
- ScriptEngine.NAME
- ScriptEngine.LANGUAGE
- ScriptEngine.LANGUAGE_VERSION

The values for these keys are the Strings returned by getEngineName, getEngineVersion, getName, getLanguageName and getLanguageVersion respectively.

A reserved key, **THREADING**, whose value describes the behavior of the engine with respect to concurrent execution of scripts and maintenance of state is also defined. These values for the **THREADING** key are:

null - The engine implementation is not thread safe, and cannot be used to execute scripts concurrently on multiple threads.

"MULTITHREADED" - The engine implementation is internally thread-safe and scripts may execute concurrently although effects of script execution on one thread may be visible to scripts on other threads.

"THREAD-ISOLATED" - The implementation satisfies the requirements of "MULTITHREADED", and also, the engine maintains independent values for symbols in scripts

executing on different threads.

"STATELESS" - The implementation satisfies the requirements of "THREAD-ISOLATED". In addition, script executions do not alter the mappings in the Bindings which is the engine scope of the ScriptEngine. In particular, the keys in the Bindings and their associated values are the same before and after the execution of the script.

Implementations may define implementation-specific keys.

Parameters:
key - The name of the parameter

Returns
The value for the given parameter. Returns null if no value is assigned to the key.

getMethodCallSyntax

```
java.lang.String getMethodCallSyntax(java.lang.String obj,
                                     java.lang.String m,
                                     java.lang.String... args)
```

Returns a String which can be used to invoke a method of a Java object using the syntax of the supported scripting language. For instance, an implementaton for a Javascript engine might be;

```
 public String getMethodCallSyntax(String obj,
                                   String m, String... args)
{
      String ret = obj;
      ret += "." + m + "(";
      for (int i = 0; i < args.length; i++) {
          ret += args[i];
          if (i == args.length - 1) {
              ret += ")";
          } else {
              ret += ",";
          }
      }
```

```
      return ret;
 }
```

Parameters:

obj - The name representing the object whose method is to be invoked. The name is the one used to create bindings using the put method of ScriptEngine, the put method of an ENGINE_SCOPE Bindings,or the setAttribute method of ScriptContext. The identifier used in scripts may be a decorated form of the specified one.
m - The name of the method to invoke.
args - names of the arguments in the method call.

Returns

The String used to invoke the method in the syntax of the scripting language.

getOutputStatement

java.lang.String **getOutputStatement**(java.lang.String toDisplay)

Returns a String that can be used as a statement to display the specified String using the syntax of the supported scripting language. For instance, the implementaton for a Perl engine might be;

```
 public String getOutputStatement(String toDisplay) {
      return "print(" + toDisplay + ")";
 }
```

Parameters:

toDisplay - The String to be displayed by the returned statement.

Returns

The string used to display the String in the syntax of the scripting language.

getProgram

java.lang.String **getProgram**(java.lang.String... statements)

Returns A valid scripting language executable progam with given statements. For instance an implementation for a PHP engine might be:

```
 public String getProgram(String... statements) {
      $retval = "<?\n";
      int len = statements.length;
      for (int i = 0; i < len; i++) {
          $retval += statements[i] + ";\n";
      }
      $retval += "?>";

 }
```

Parameters:

statements - The statements to be executed. May be return values of calls to the getMethodCallSyntax and getOutputStatement methods.

Returns

The Program

getScriptEngine

ScriptEngine **getScriptEngine**()

Returns an instance of the ScriptEngine associated with this ScriptEngineFactory. A new ScriptEngine is generally returned, but implementations may pool, share or reuse engines.

Returns

A new ScriptEngine instance.

SCR.5.7 CompiledScript

javax.script
Class CompiledScript

public abstract class CompiledScript

Extended by Classes that store results of compilations. State might be stored in the form of Java classes, Java class files or scripting language opcodes. The script may be executed repeatedly without reparsing.

Each CompiledScript is associated with a ScriptEngine -- A call to an eval method of the CompiledScript causes the execution of the script by the ScriptEngine. Changes in the state of the ScriptEngine caused by execution of tne CompiledScript may visible during subsequent executions of scripts by the engine.

Since:
1.6

SCR.5.7.1 Constructors

CompiledScript

public **CompiledScript**()

SCR.5.7.2 Methods

eval

public abstract java.lang.Object **eval**(ScriptContext context)
throws ScriptException

Executes the program stored in this CompiledScript object.

Parameters:
context - A ScriptContext that is used in the same way

as the ScriptContext passed to the eval methods of ScriptEngine.

Returns

The value returned by the script execution, if any. Should return null if no value is returned by the script execution.

Throws

ScriptException - if an error occurs.
java.lang.NullPointerException - if context is null.

eval

public java.lang.Object **eval**(Bindings bindings)
throws ScriptException

Executes the program stored in the CompiledScript object using the supplied Bindings of attributes as the ENGINE_SCOPE of the associated ScriptEngine during script execution. If bindings is null, then the effect of calling this method is same as that of eval(getEngine().getContext()).

. The GLOBAL_SCOPE Bindings, Reader and Writer associated with the default ScriptContext of the associated ScriptEngine are used.

Parameters:

bindings - The bindings of attributes used for the ENGINE_SCOPE.

Returns

The return value from the script execution

Throws

ScriptException - if an error occurs.

eval

public java.lang.Object **eval**()

throws ScriptException

Executes the program stored in the CompiledScript object. The default ScriptContext of the associated ScriptEngine is used. The effect of calling this method is same as that of eval(getEngine().getContext()).

Returns
The return value from the script execution

Throws
ScriptException - if an error occurs.

getEngine

public abstract ScriptEngine **getEngine**()

Returns the ScriptEngine wbose compile method created this CompiledScript. The CompiledScript will execute in this engine.

Returns
The ScriptEngine that created this CompiledScript

SCR.5.8 SimpleScriptContext

javax.script
Class SimpleScriptContext

All Implemented Interfaces:
ScriptContext

public class SimpleScriptContext implements ScriptContext

Simple implementation of ScriptContext.

Since:
1.6

SCR.5.8.1 Fields

writer

protected java.io.Writer **writer**

This is the writer to be used to output from scripts. By default, a PrintWriter based on System.out is used. Accessor methods getWriter, setWriter are used to manage this field.

errorWriter

protected java.io.Writer **errorWriter**

This is the writer to be used to output errors from scripts. By default, a PrintWriter based on System.err is used. Accessor methods getErrorWriter, setErrorWriter are used to manage this field.

reader

```
protected java.io.Reader reader
```

This is the reader to be used for input from scripts. By default, a InputStreamReader based on System.in is used and default charset is used by this reader. Accessor methods getReader, setReader are used to manage this field.

engineScope

```
protected Bindings engineScope
```

This is the engine scope bindings. By default, a SimpleBindings is used. Accessor methods setBindings, getBindings are used to manage this field.

globalScope

```
protected Bindings globalScope
```

This is the global scope bindings. By default, a null value (which means no global scope) is used. Accessor methods setBindings, getBindings are used to manage this field.

SCR.5.8.2 Constructors

SimpleScriptContext

```
public SimpleScriptContext()
```

SCR.5.8.3 Methods

setBindings

```
public void setBindings(Bindings bindings,
                        int scope)
```

Sets a Bindings of attributes for the given scope. If the value of scope is ENGINE_SCOPE the given Bindings replaces the engineScope field. If the value of scope is GLOBAL_SCOPE the given Bindings replaces the globalScope field.

Specified by:
setBindings in interface ScriptContext
Parameters:
bindings - The Bindings of attributes to set.
scope - The value of the scope in which the attributes are set.

Throws
java.lang.IllegalArgumentException - if scope is invalid.
java.lang.NullPointerException - if the value of scope is ENGINE_SCOPE and the specified Bindings is null.

getAttribute

public java.lang.Object **getAttribute**(java.lang.String name)

Retrieves the value of the attribute with the given name in the scope occurring earliest in the search order. The order is determined by the numeric value of the scope parameter (lowest scope values first.)

Specified by:
getAttribute in interface ScriptContext
Parameters:
name - The name of the the attribute to retrieve.

Returns
The value of the attribute in the lowest scope for which an attribute with the given name is defined. Returns null if no attribute with the name exists in any scope.

Throws
java.lang.NullPointerException - if the name is null.
java.lang.IllegalArgumentException - if the name is empty.

getAttribute

```
public java.lang.Object getAttribute(java.lang.String name,
                                      int scope)
```

Gets the value of an attribute in a given scope.

Specified by:
getAttribute in interface ScriptContext
Parameters:
name - The name of the attribute to retrieve.
scope - The scope in which to retrieve the attribute.

Returns
The value of the attribute. Returns null is the name does not exist in the given scope.

Throws
java.lang.IllegalArgumentException - if the name is empty or if the value of scope is invalid.
java.lang.NullPointerException - if the name is null.

removeAttribute

```
public java.lang.Object removeAttribute(java.lang.String name,
                                        int scope)
```

Remove an attribute in a given scope.

Specified by:
removeAttribute in interface ScriptContext
Parameters:
name - The name of the attribute to remove
scope - The scope in which to remove the attribute

Returns
The removed value.

Throws
java.lang.IllegalArgumentException - if the name is empty or if the scope is invalid.
java.lang.NullPointerException - if the name is null.

setAttribute

```
public void setAttribute(java.lang.String name,
                         java.lang.Object value,
                         int scope)
```

Sets the value of an attribute in a given scope.

Specified by:
setAttribute in interface ScriptContext
Parameters:
name - The name of the attribute to set
value - The value of the attribute
scope - The scope in which to set the attribute

Throws
java.lang.IllegalArgumentException - if the name is empty or if the scope is invalid.
java.lang.NullPointerException - if the name is null.

getWriter

```
public java.io.Writer getWriter()
```

Returns the Writer for scripts to use when displaying output.

Specified by:
getWriter in interface ScriptContext

Returns
The Writer.

getReader

```
public java.io.Reader getReader()
```

Returns a Reader to be used by the script to read input.

Specified by:
getReader in interface ScriptContext

Returns
The Reader.

setReader

public void **setReader**(java.io.Reader reader)

Sets the Reader for scripts to read input .

Specified by:
setReader in interface ScriptContext
Parameters:
reader - The new Reader.

setWriter

public void **setWriter**(java.io.Writer writer)

Sets the Writer for scripts to use when displaying output.

Specified by:
setWriter in interface ScriptContext
Parameters:
writer - The new Writer.

getErrorWriter

public java.io.Writer **getErrorWriter**()

Returns the Writer used to display error output.

Specified by:
getErrorWriter in interface ScriptContext

Returns
The Writer

setErrorWriter

public void **setErrorWriter**(java.io.Writer writer)

Sets the Writer used to display error output.

Specified by:
setErrorWriter in interface ScriptContext
Parameters:
writer - The Writer.

getAttributesScope

public int **getAttributesScope**(java.lang.String name)

Get the lowest scope in which an attribute is defined.

Specified by:
getAttributesScope in interface ScriptContext
Parameters:
name - Name of the attribute .

Returns
The lowest scope. Returns -1 if no attribute with the given name is defined in any scope.

Throws
java.lang.NullPointerException - if name is null.
java.lang.IllegalArgumentException - if name is empty.

getBindings

public Bindings **getBindings**(int scope)

Returns the value of the engineScope field if specified scope is ENGINE_SCOPE. Returns the value of the globalScope field if the specified scope is GLOBAL_SCOPE.

Specified by:
getBindings in interface ScriptContext
Parameters:
scope - The specified scope

Returns
The value of either the engineScope or globalScope field.

Throws
java.lang.IllegalArgumentException - if the value of scope is invalid.

getScopes

public java.util.List<java.lang.Integer> **getScopes**()

Returns immutable List of all the valid values for scope in the ScriptContext.

Specified by:
getScopes in interface ScriptContext

Returns
list of scope values

SCR.5.9 AbstractScriptEngine

javax.script
Class AbstractScriptEngine

All Implemented Interfaces:
ScriptEngine

public abstract class AbstractScriptEngine implements ScriptEngine

Provides a standard implementation for several of the variants of the `eval` method.

eval(Reader)

eval(String)

eval(String, Bindings)

eval(Reader, Bindings)

are implemented using the abstract methods

eval(Reader,ScriptContext) or **eval(String, ScriptContext)**

with a `SimpleScriptContext`.

A `SimpleScriptContext` is used as the default `ScriptContext` of the `AbstractScriptEngine`..

Since:
1.6

SCR.5.9.1 Fields

context

`protected ScriptContext` **`context`**

The default ScriptContext of this AbstractScriptEngine.

SCR.5.9.2 Constructors

AbstractScriptEngine

public **AbstractScriptEngine**()

Creates a new instance of AbstractScriptEngine using a SimpleScriptContext as its default ScriptContext.

AbstractScriptEngine

public **AbstractScriptEngine**(Bindings n)

Creates a new instance using the specified Bindings as the ENGINE_SCOPE Bindings in the protected context field.

Parameters:
n - The specified Bindings.

Throws
java.lang.NullPointerException - if n is null.

SCR.5.9.3 Methods

setContext

public void **setContext**(ScriptContext ctxt)

Sets the value of the protected context field to the specified ScriptContext.

Specified by:
setContext in interface ScriptEngine
Parameters:
ctxt - The specified ScriptContext.

Throws
java.lang.NullPointerException - if ctxt is null.

getContext

public ScriptContext **getContext**()

Returns the value of the protected context field.

Specified by:
getContext in interface ScriptEngine

Returns
The value of the protected context field.

getBindings

public Bindings **getBindings**(int scope)

Returns the Bindings with the specified scope value in the protected context field.

Specified by:
getBindings in interface ScriptEngine
Parameters:
scope - The specified scope

Returns
The corresponding Bindings.

Throws
java.lang.IllegalArgumentException - if the value of scope is invalid for the type the protected context field.

setBindings

public void **setBindings**(Bindings bindings,
int scope)

Sets the Bindings with the corresponding scope value in the context field.

Specified by:
setBindings in interface ScriptEngine
Parameters:
bindings - The specified Bindings.
scope - The specified scope.

Throws
java.lang.IllegalArgumentException - if the value of scope is invalid for the type the context field.
java.lang.NullPointerException - if the bindings is null and the scope is ScriptContext.ENGINE_SCOPE

put

public void **put**(java.lang.String key,
java.lang.Object value)

Sets the specified value with the specified key in the ENGINE_SCOPE Bindings of the protected context field.

Specified by:
put in interface ScriptEngine
Parameters:
key - The specified key.
value - The specified value.

Throws
java.lang.NullPointerException - if key is null.
java.lang.IllegalArgumentException - if key is empty.

get

public java.lang.Object **get**(java.lang.String key)

Gets the value for the specified key in the ENGINE_SCOPE of the protected context field.

Specified by:
get in interface ScriptEngine

Parameters:
key - The key whose value is to be returned

Returns
The value for the specified key.

Throws
java.lang.NullPointerException - if key is null.
java.lang.IllegalArgumentException - if key is empty.

eval

```
public java.lang.Object eval(java.io.Reader reader,
                             Bindings bindings)
                      throws ScriptException
```

eval(Reader, Bindings) calls the abstract eval(Reader, ScriptContext) method, passing it a ScriptContext whose Reader, Writers and Bindings for scopes other that ENGINE_SCOPE are identical to those members of the protected context field. The specified Bindings is used instead of the ENGINE_SCOPE Bindings of the context field.

Specified by:
eval in interface ScriptEngine
Parameters:
reader - A Reader containing the source of the script.
bindings - A Bindings to use for the ENGINE_SCOPE while the script executes.

Returns
The return value from eval(Reader, ScriptContext)

Throws
ScriptException - if an error occurs in script.
java.lang.NullPointerException - if any of the parameters is null.

eval

```
public java.lang.Object eval(java.lang.String script,
                             Bindings bindings)
                      throws ScriptException
```

Same as eval(Reader, Bindings) except that the abstract eval(String, ScriptContext) is used.

Specified by:
eval in interface ScriptEngine

Parameters:
script - A String containing the source of the script.
bindings - A Bindings to use as the ENGINE_SCOPE while the script executes.

Returns
The return value from eval(String, ScriptContext)

Throws
ScriptException - if an error occurs in script.
java.lang.NullPointerException - if any of the parameters is null.

eval

```
public java.lang.Object eval(java.io.Reader reader)
                      throws ScriptException
```

eval(Reader) calls the abstract eval(Reader, ScriptContext) passing the value of the context field.

Specified by:
eval in interface ScriptEngine

Parameters:
reader - A Reader containing the source of the script.

Returns
The return value from eval(Reader, ScriptContext)

Throws
ScriptException - if an error occurs in script.

java.lang.NullPointerException - if any of the parameters is null.

eval

public java.lang.Object **eval**(java.lang.String script)
throws ScriptException

Same as eval(Reader) except that the abstract eval(String, ScriptContext) is used.

Specified by:
eval in interface ScriptEngine
Parameters:
script - A String containing the source of the script.

Returns
The return value from eval(String, ScriptContext)

Throws
ScriptException - if an error occurrs in script.
java.lang.NullPointerException - if any of the parameters is null.

getScriptContext

protected ScriptContext **getScriptContext**(Bindings nn)

Returns a SimpleScriptContext. The SimpleScriptContext:

- Uses the specified Bindings for its ENGINE_SCOPE
- Uses the Bindings returned by the abstract getGlobalScope method as its GLOBAL_SCOPE
- Uses the Reader and Writer in the default ScriptContext of this ScriptEngine

A SimpleScriptContext returned by this method is used to implement eval methods using the abstract eval(Reader,Bindings) and eval(String,Bindings) versions.

Parameters:

```
        nn - Bindings to use for the ENGINE_SCOPE
```

Returns
```
        The SimpleScriptContext
```

SCR.5.10 ScriptEngineManager

javax.script
Class ScriptEngineManager

public class ScriptEngineManager

The `ScriptEngineManager` implements a discovery and instantiation mechanism for `ScriptEngine` classes and also maintains a collection of key/value pairs storing state shared by all engines created by the Manager. This class uses the service provider mechanism to enumerate all the implementations of `ScriptEngineFactory`.

The `ScriptEngineManager` provides a method to return an array of all these factories as well as utility methods which look up factories on the basis of language name, file extension and mime type.

The `Bindings` of key/value pairs, referred to as the "Global Scope" maintained by the manager is available to all instances of `ScriptEngine` created by the `ScriptEngineManager`. The values in the `Bindings` are generally exposed in all scripts.

Since:
```
    1.6
```

SCR.5.10.1 Constructors

ScriptEngineManager

`public` **`ScriptEngineManager`**`()`

```
    If the thread context ClassLoader can be accessed by the
    caller, then the effect of calling this constructor is the
```

same as calling ScriptEngineManager(Thread.currentThread().getContextClassLoader()). Otherwise, the effect is the same as calling ScriptEngineManager(null).

ScriptEngineManager

public **ScriptEngineManager**(java.lang.ClassLoader loader)

This constructor loads the implementations of ScriptEngineFactory visible to the given ClassLoader using the service provider mechanism.

If loader is null, the script engine factories that are bundled with the platform and that are in the usual extension directories (installed extensions) are loaded.

Parameters:
loader - ClassLoader used to discover script engine factories.

SCR.5.10.2 Methods

setBindings

public void **setBindings**(Bindings bindings)

setBindings stores the specified Bindings in the globalScope field. ScriptEngineManager sets this Bindings as global bindings for ScriptEngine objects created by it.

Parameters:
bindings - The specified Bindings

Throws
java.lang.IllegalArgumentException - if bindings is null.

getBindings

public Bindings **getBindings**()

getBindings returns the value of the globalScope field. ScriptEngineManager sets this Bindings as global bindings for ScriptEngine objects created by it.

Returns

The globalScope field.

put

public void **put**(java.lang.String key,
java.lang.Object value)

Sets the specified key/value pair in the Global Scope.

Parameters:

key - Key to set
value - Value to set.

Throws

java.lang.NullPointerException - if key is null.
java.lang.IllegalArgumentException - if key is empty string.

get

public java.lang.Object **get**(java.lang.String key)

Gets the value for the specified key in the Global Scope

Parameters:

key - The key whose value is to be returned.

Returns

The value for the specified key.

getEngineByName

public ScriptEngine **getEngineByName**(java.lang.String shortName)

Looks up and creates a ScriptEngine for a given name. The algorithm first searches for a ScriptEngineFactory that has been registered as a handler for the specified name using the registerEngineName method.

If one is not found, it searches the array of ScriptEngineFactory instances stored by the constructor for one with the specified name. If a ScriptEngineFactory is found by either method, it is used to create instance of ScriptEngine.

Parameters:

shortName - The short name of the ScriptEngine implementation. returned by the getNames method of its ScriptEngineFactory.

Returns

A ScriptEngine created by the factory located in the search. Returns null if no such factory was found. The ScriptEngineManager sets its own globalScope Bindings as the GLOBAL_SCOPE Bindings of the newly created ScriptEngine.

Throws

java.lang.NullPointerException - if shortName is null.

getEngineByExtension

public ScriptEngine **getEngineByExtension**(java.lang.String extension)

Look up and create a ScriptEngine for a given extension. The algorithm used by getEngineByName is used except that the search starts by looking for a ScriptEngineFactory registered to handle the given extension using registerEngineExtension.

Parameters:

extension - The given extension

Returns
The engine to handle scripts with this extension. Returns null if not found.

Throws
java.lang.NullPointerException - if extension is null.

getEngineByMimeType

public ScriptEngine **getEngineByMimeType**(java.lang.String mimeType)

Look up and create a ScriptEngine for a given mime type. The algorithm used by getEngineByName is used except that the search starts by looking for a ScriptEngineFactory registered to handle the given mime type using registerEngineMimeType.

Parameters:
mimeType - The given mime type

Returns
The engine to handle scripts with this mime type. Returns null if not found.

Throws
java.lang.NullPointerException - if mimeType is null.

getEngineFactories

public java.util.List<ScriptEngineFactory> **getEngineFactories**()

Returns an array whose elements are instances of all the ScriptEngineFactory classes found by the discovery mechanism.

Returns
List of all discovered ScriptEngineFactorys.

registerEngineName

```
public void registerEngineName(java.lang.String name,
                               ScriptEngineFactory factory)
```

Registers a ScriptEngineFactory to handle a language name. Overrides any such association found using the Discovery mechanism.

Parameters:

name - The name to be associated with the ScriptEngineFactory.
factory - The class to associate with the given name.

Throws

java.lang.NullPointerException - if any of the parameters is null.

registerEngineMimeType

```
public void registerEngineMimeType(java.lang.String type,
                                   ScriptEngineFactory factory)
```

Registers a ScriptEngineFactory to handle a mime type. Overrides any such association found using the Discovery mechanism.

Parameters:

type - The mime type to be associated with the ScriptEngineFactory.
factory - The class to associate with the given mime type.

Throws

java.lang.NullPointerException - if any of the parameters is null.

registerEngineExtension

```
public void registerEngineExtension(java.lang.String extension,
                                    ScriptEngineFactory factory)
```

Registers a ScriptEngineFactory to handle an extension.

Overrides any such association found using the Discovery mechanism.

Parameters:
extension - The extension type to be associated with the ScriptEngineFactory.
factory - The class to associate with the given extension.

Throws
java.lang.NullPointerException - if any of the parameters is null.

SCR.5.11 SimpleBindings

javax.script
Class SimpleBindings

All Implemented Interfaces:
java.util.Map<java.lang.String,java.lang.Object>, Bindings

public class SimpleBindings implements Bindings

A simple implementation of Bindings backed by a HashMap or some other specified Map.

Since:
1.6

SCR.5.11.1 Constructors

SimpleBindings

public
SimpleBindings(java.util.Map<java.lang.String,java.lang.Object>

m)

Constructor uses an existing Map to store the values.

Parameters:
m - The Map backing this SimpleBindings.

Throws
java.lang.NullPointerException - if m is null

SimpleBindings

public **SimpleBindings**()

Default constructor uses a HashMap.

SCR.5.11.2 Methods

put

public java.lang.Object **put**(java.lang.String name,
java.lang.Object value)

Sets the specified key/value in the underlying map field.

Specified by:
put in interface
java.util.Map<java.lang.String,java.lang.Object>
Specified by:
put in interface Bindings
Parameters:
name - Name of value
value - Value to set.

Returns
Previous value for the specified key. Returns null if key was previously unset.

Throws

java.lang.NullPointerException - if the name is null.
java.lang.IllegalArgumentException - if the name is empty.

putAll

public void **putAll**(java.util.Map<? extends java.lang.String,? extends java.lang.Object> toMerge)

putAll is implemented using Map.putAll.

Specified by:
putAll in interface java.util.Map<java.lang.String,java.lang.Object>
Specified by:
putAll in interface Bindings
Parameters:
toMerge - The Map of values to add.

Throws
java.lang.NullPointerException - if toMerge map is null or if some key in the map is null.
java.lang.IllegalArgumentException - if some key in the map is an empty String.

clear

public void **clear**()

Specified by:
clear in interface java.util.Map<java.lang.String,java.lang.Object>

containsKey

public boolean **containsKey**(java.lang.Object key)

Returns true if this map contains a mapping for the specified key. More formally, returns true if and only if this map contains a mapping for a key k such that (key==null ? k==null : key.equals(k)). (There can be at most one such mapping.)

Specified by:
containsKey in interface
java.util.Map<java.lang.String,java.lang.Object>
Specified by:
containsKey in interface Bindings
Parameters:
key - key whose presence in this map is to be tested.

Returns
true if this map contains a mapping for the specified key.

Throws
java.lang.NullPointerException - if key is null
java.lang.ClassCastException - if key is not String
java.lang.IllegalArgumentException - if key is empty String

containsValue

public boolean **containsValue**(java.lang.Object value)

Specified by:
containsValue in interface
java.util.Map<java.lang.String,java.lang.Object>

entrySet

public
java.util.Set<java.util.Map.Entry<java.lang.String,java.lang.Object>> **entrySet**()

Specified by:
entrySet in interface
java.util.Map<java.lang.String,java.lang.Object>

get

public java.lang.Object **get**(java.lang.Object key)

Returns the value to which this map maps the specified key.

Returns null if the map contains no mapping for this key. A return value of null does not *necessarily* indicate that the map contains no mapping for the key; it's also possible that the map explicitly maps the key to null. The containsKey operation may be used to distinguish these two cases.

More formally, if this map contains a mapping from a key k to a value v such that (key==null ? k==null : key.equals(k)), then this method returns v; otherwise it returns null. (There can be at most one such mapping.)

Specified by:
get in interface
java.util.Map<java.lang.String,java.lang.Object>
Specified by:
get in interface Bindings
Parameters:
key - key whose associated value is to be returned.

Returns
the value to which this map maps the specified key, or null if the map contains no mapping for this key.

Throws
java.lang.NullPointerException - if key is null
java.lang.ClassCastException - if key is not String
java.lang.IllegalArgumentException - if key is empty String

isEmpty

public boolean **isEmpty**()

Specified by:
isEmpty in interface
java.util.Map<java.lang.String,java.lang.Object>

keySet

public java.util.Set<java.lang.String> **keySet**()

Specified by:

keySet in interface
java.util.Map<java.lang.String,java.lang.Object>

remove

public java.lang.Object **remove**(java.lang.Object key)

Removes the mapping for this key from this map if it is present (optional operation). More formally, if this map contains a mapping from key k to value v such that (key==null ? k==null : key.equals(k)), that mapping is removed. (The map can contain at most one such mapping.)

Returns the value to which the map previously associated the key, or null if the map contained no mapping for this key. (A null return can also indicate that the map previously associated null with the specified key if the implementation supports null values.) The map will not contain a mapping for the specified key once the call returns.

Specified by:
remove in interface
java.util.Map<java.lang.String,java.lang.Object>
Specified by:
remove in interface Bindings
Parameters:
key - key whose mapping is to be removed from the map.

Returns
previous value associated with specified key, or null if there was no mapping for key.

Throws
java.lang.NullPointerException - if key is null
java.lang.ClassCastException - if key is not String
java.lang.IllegalArgumentException - if key is empty String

size

public int **size**()

Specified by:
size in interface
java.util.Map<java.lang.String,java.lang.Object>

values

```
public java.util.Collection<java.lang.Object> values()
```

Specified by:
values in interface
java.util.Map<java.lang.String,java.lang.Object>

SCR.5.12 ScriptException

javax.script

Class ScriptException

All Implemented Interfaces:
java.io.Serializable

public class ScriptException extends java.lang.Exception

The generic `Exception` class for the Scripting APIs. Checked exception types thrown by underlying scripting implementations must be wrapped in instances of `ScriptException`. The class has members to store line and column numbers and filenames if this information is available.

Since:
1.6

SCR.5.12.1 Constructors

ScriptException

```
public ScriptException(java.lang.String s)
```

Creates a ScriptException with a String to be used in its message. Filename, and line and column numbers are unspecified.

Parameters:

s - The String to use in the message.

ScriptException

```
public ScriptException(java.lang.Exception e)
```

Creates a ScriptException wrapping an Exception thrown by an underlying interpreter. Line and column numbers and filename are unspecified.

Parameters:

e - The wrapped Exception.

ScriptException

```
public ScriptException(java.lang.String message,
                       java.lang.String fileName,
                       int lineNumber)
```

Creates a ScriptException with message, filename and linenumber to be used in error messages.

Parameters:

message - The string to use in the message
fileName - The file or resource name describing the location of a script error causing the ScriptException to be thrown.
lineNumber - A line number describing the location of a script error causing the ScriptException to be thrown.

ScriptException

```
public ScriptException(java.lang.String message,
                       java.lang.String fileName,
```

```
                    int lineNumber,
                    int columnNumber)
```

ScriptException constructor specifying message, filename, line number and column number.

Parameters:
message - The message.
fileName - The filename
lineNumber - the line number.
columnNumber - the column number.

SCR.5.12.2 Methods

getMessage

public java.lang.String **getMessage**()

Returns a message containing the String passed to a constructor as well as line and column numbers and filename if any of these are known.

Overrides:
getMessage in class java.lang.Throwable

Returns
The error message.

getLineNumber

public int **getLineNumber**()

Get the line number on which an error occurred.

Returns
The line number. Returns -1 if a line number is unavailable.

getColumnNumber

public int **getColumnNumber**()

Get the column number on which an error occurred.

Returns

The column number. Returns -1 if a column number is unavailable.

getFileName

public java.lang.String **getFileName**()

Get the source of the script causing the error.

Returns

The file name of the script or some other string describing the script source. May return some implementation-defined string such as <*unknown*> if a description of the source is unavailable.