Часть 1. **javax.script,** пакет  добавленный в Java SE 6

Java-разработчики знают, что Java не всегда является лучшим языком для решения некоторого рода задач. В этом году релизы версий 1.0 для JRuby и Groovy усилили интерес к встраиванию динамических языков в приложения Java. Groovy, JRuby, Rhino, Jython и прочие проекты с открытым кодом предоставляют возможность писать код на так называемых скриптовых языках и запускать его под управлением JVM (см. [Ресурсы](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#artrelatedtopics)). До сих пор интеграция таких языков с Java-кодом, обычно, подразумевала необходимость изучения для каждого интерпретатора его уникального API и особенностей реализации.

Пакет javax.script, добавленный в Java SE 6, облегчает процесс интеграции динамических языков. Он предоставляет единообразный, простой способ вызова множества скриптовых языков при использовании небольшого набора интерфейсов и реальных классов. Но Java scripting API — это больше, чем просто облегчение в создании скриптовых фрагментов приложения; пакет поддержки скриптинга позволяет вам считывать и вызывать внешние скрипты на лету, что означает возможность динамической модификации самих скриптов для изменения поведения исполняемого приложения.

Эта статья, первая в серии из двух частей, является введением в возможности и ключевые классы Java scripting API с использованием приложения в стиле "Hello World". [Часть 2](http://www.ibm.com/developerworks/java/library/j-javascripting2/) представляет более реалистичный пример приложения, раскрывающий дополнительные сильные стороны скриптового API. В этом приложении scripting API задействован для создания движка с динамически настраиваемыми правилами, в котором правила кодируются в виде внешних скриптов, написанных на Groovy, JavaScript и Ruby. Правила принимают решение — может ли соискатель внутреннего заема претендовать на отдельные ипотечные продукты. Воплощение таких правил посредством Java scripting API позволяет изменять сами правила и добавлять в процессе выполнения новые продукты ипотеки.

API Java-скриптинга

Пакет поддержки скриптинга был добавлен в язык Java в декабре 2006 для обеспечения унифицированного способа интеграции скриптовых языков в приложение Java. Для разработчиков скриптовых языков пакет предоставляет способ написания связующего кода, позволяющего их языкам вызываться динамически из Java-приложения. Для Java-разработчиков пакет предлагает небольшой набор классов и интерфейсов, которые дают возможность скриптам, написанным на любом множестве языков, быть вызванным через обобщенный API. Пакет скриптинга, таким образом, подобен пакету Java Database Connectivity (JDBC) — с ним различные языки (как различные базы данных) могут быть интегрированы в платформу Java посредством согласующего интерфейса.

Ранее динамический вызов скриптового языка из Java-кода усложнялся использованием специфичных классов, поставляемых в дистрибутиве каждого языка, или же использованием Bean Scripting Framework (BSF) от Apache Jakarta. BSF объединяет небольшое количество скриптовых языков под управлением единого API (см. [Ресурсы](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#artrelatedtopics)). Более двух дюжин скриптовых языков, включая AppleScript, Groovy, JavaScript, Jelly, PHP, Python, Ruby и Velocity, могут быть интегрированы в Java-код при использовании Java SE 6 Scripting API, главным образом базирующемся на BSF.

Scripting API обеспечивает двухстороннюю видимость между Java-приложениями и внешними скриптами. Ваш Java-код может не только вызывать внешние скрипты, но также может предоставить таким скриптам доступ к избранным Java-объектам. Внешний скрипт Ruby, к примеру, может вызывать методы Java-объектов и иметь доступ к их свойствам, что позволяет скриптам добавлять в исполняемое приложение поведение, не предусмотренное на момент разработки.

Обращение к внешним скриптам может быть использовано для расширения функциональности приложения по ходу его выполнения, конфигурирования, мониторинга или других оперативных манипуляций — таких как изменение бизнес-логики без необходимости останавливать приложение. Возможные применения пакета поддержки скриптинга включают:

* Написание бизнес-логики на более простом, чем Java, языке, не обращаясь при этом к развитым средам выполнения.
* Построение архитектуры на базе подключаемых модулей (плагинов), позволяющей пользователям настраивать приложение на лету.
* Интегрировать существующий скрипт в ваше Java-приложение, скажем, скрипт обрабатывающий или преобразовывающий текстовые файлы.
* Производить внешнюю оперативную конфигурацию поведения приложения с помощью полноценного языка программирования вместо настроечного файла.
* Добавлять в Java-приложение язык, специфичный для данной предметной области.
* Применять скриптовый язык на этапе прототипирования Java-приложения.
* Писать на скриптовом языке тестирующий код для приложения.

Простой пример из Википедии:

Вот пример Java-кода, запускающего JavaScript print('Hello, world!')

**import** **javax.script.ScriptEngine**;

**import** **javax.script.ScriptEngineManager**;

**import** **javax.script.ScriptException**;

**public** **class** **RhinoEngine** {

**public** **static** void main(String[] args) {

ScriptEngineManager mgr = **new** ScriptEngineManager();

ScriptEngine engine = mgr.getEngineByName("JavaScript");

*// Теперь у нас есть экземпляр движка и мы можем выполнить JavaScript*

**try** {

engine.put("name", args[0]); //устанавливает указанную пару ключ(*произвольно*

*названная переменная)*\значение*(тут может быть*

*массив/класс/переменная из Java)*вГлобальной

области видимости. Т.е. делает параметр из Java

доступным в скриптах

engine.eval("print('Hello ' + name + '!')"); //запускает строку кода на

JavaScript - print('Hello ' + name + '!')

} **catch** (ScriptException ex) {

ex.printStackTrace();

}

}

}

Метод **eval(**Reader ---/String ---**)** вызывает выполнение скрипта либо из файла/либо тут-же написанного

Привет, скриптовый мир

Класс HelloScriptingWorld, который вы можете загрузить наряду с прочим кодом для этой статьи (см. [Загрузка](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#artdownload)), демонстрирует ключевые возможности пакета поддержки Java-скриптинга. В нем используются жестко закодированные фрагменты на JavaScript, взятого в качестве примера скриптового языка.

package com.mcqueeney.scripting;

import javax.script.Invocable;

import javax.script.ScriptEngine;

import javax.script.ScriptEngineManager;

import javax.script.ScriptException;

public **class** **HelloScriptingWorld** {

public static void main(String[] args)

throws ScriptException, NoSuchMethodException

{

ScriptEngineManager scriptEngineMgr = new ScriptEngineManager();

ScriptEngine jsEngine = scriptEngineMgr.getEngineByName("JavaScript");

if (jsEngine == null)

{

System.err.println("No script engine found for JavaScript");

System.exit(1);

}

System.out.println("Calling invokeHelloScript...");

invokeHelloScript(jsEngine);

System.out.println("\nCalling defineScriptFunction...");

defineScriptFunction(jsEngine);

System.out.println("\nCalling invokeScriptFunctionFromEngine...");

invokeScriptFunctionFromEngine(jsEngine);

System.out.println("\nCalling invokeScriptFunctionFromJava...");

invokeScriptFunctionFromJava(jsEngine);

System.out.println("\nCalling invokeJavaFromScriptFunction...");

invokeJavaFromScriptFunction(jsEngine);

}

private static void invokeHelloScript(ScriptEngine jsEngine)

throws ScriptException

{ jsEngine.eval("println('Hello from JavaScript')"); }

private static void defineScriptFunction(ScriptEngine engine)

throws ScriptException

{

// Тут задается, **но не выполняется,** функция

engine.eval(

"function sayHello(name) {" +

" println('Hello, ' + name)" +

"}");

}

private static void invokeScriptFunctionFromEngine(ScriptEngine jsEngine)

throws ScriptException

{ jsEngine.eval("sayHello('World!')"); }

private static void invokeScriptFunctionFromJava(ScriptEngine jsEngine)

throws ScriptException, NoSuchMethodException

{

Invocable invocableEngine = (Invocable) jsEngine;

invocableEngine.invokeFunction("sayHello", "from Java");

}

private static void invokeJavaFromScriptFunction(ScriptEngine jsEngine)

throws ScriptException

{

jsEngine.put("helloScriptingWorld", new HelloScriptingWorld());

jsEngine.eval(

"println('Invoking getHelloReply method from JavaScript...');" +

"var msg = helloScriptingWorld.getHelloReply('JavaScript');" +

"println('Java returned: ' + msg)"

);

}

/\*\* Method invoked from the above script to return a string. \*/

public String getHelloReply(String name)

{ return "Java method getHelloReply says, 'Hello, " + name + "'"; }

}

То, что мы увидим на экране после запуска HelloScriptingWorld:

|  |  |
| --- | --- |
|  | Вызываю invokeHelloScript...  Hello from JavaScript    Вызываю defineScriptFunction...    Вызываю invokeScriptFunctionFromEngine...  Hello, World!    Вызываю invokeScriptFunctionFromJava...  Hello, from Java    Вызываю invokeJavaFromScriptFunction...  Вызов метода getHelloReply из JavaScript...  Получили из Java: Java-метод getHelloReply говорит 'Привет, JavaScript' |

**ДАЛЕЕ ЭТОТ ПРИМЕР ОПИСАН БОЛЕЕ ПОДРОБНО.**

Содержащийся в классе метод main(), показанный в **листинге 1**,

создает исполняющее окружение для JavaScript скрипта, а затем вызывает пять методов (показанных в последующих листингах), подчеркивающих возможности пакета:

Листинг 1. Метод main для HelloScriptingWorld

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public static void main(String[] args) throws ScriptException, NoSuchMethodException {    ScriptEngineManager scriptEngineMgr = new ScriptEngineManager();  ScriptEngine jsEngine = scriptEngineMgr.getEngineByName("JavaScript"); //тут "JavaScript"   это наименование языка, код которого будет использован      if (jsEngine == null) {          System.err.println("Для JavaScript не найдено скриптового движка");          System.exit(1);      }        System.out.println("Вызываем invokeHelloScript...");      invokeHelloScript(jsEngine);        System.out.println("\nВызываем defineScriptFunction...");      defineScriptFunction(jsEngine);        System.out.println("\nВызываем invokeScriptFunctionFromEngine...");      invokeScriptFunctionFromEngine(jsEngine);        System.out.println("\nВызываем invokeScriptFunctionFromJava...");      invokeScriptFunctionFromJava(jsEngine);        System.out.println("\nВызываем invokeJavaFromScriptFunction...");      invokeJavaFromScriptFunction(jsEngine);  } |

Главная задача метода main()— получение экземпляра javax.script.ScriptEngine (две первых конструкции в [листинге 1](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing1)). Скриптовый движок загружает и выполняет скрипты для некоторого конкретного языка. Это наиболее часто используемый и востребованный класс в пакете Java-скриптинга. Вы извлекаете скриптовый движок из javax.script.ScriptEngineManager (первое выражение присваивания). Типичная необходимость — получение в программе только одного экземпляра движка, если только не используется множество скриптовых языков.

**Почему в нашем примере JavaScript?**

В нашем примере Hello World мы используем JavaScript отчасти потому что его код прост для понимания, но в большей степени — потому что среды выполнения для Java 6 в поставке от Sun Microsystems и BEA Systems комплектуются JavaScript-интерпретатором на базе реализации с открытым кодом Mozilla Rhino. Для JavaScript у вас нет необходимости добавлять JAR-файлы поддержки скриптового языка в переменную окружения CLASSPATH.

Класс ScriptEngineManager

ScriptEngineManager, возможно, единственный реальный класс в пакете скриптинга к которому вы будете обращаться регулярно; большинство прочего — интерфейсы. И это, может быть, единственный класс из пакета скриптинга, экземпляры которого вы будете создавать напрямую (или косвенно — через механизм внедрения зависимости (*dependency-injection)* так, как это делается в Spring Framework.)

ScriptEngineManager может возвращать скриптовый движок одним из трех способов:

* По имени движка или языка, как запрашивается JavaScript в [листинге 1](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing1).
* По расширению файла, общепринятому в использовании для скриптов на этом языке, скажем, .rb для Ruby.
* По MIME-типу, в случае, когда в скриптовом движке заявлена его поддержка.

ScriptEngineManager-ы находят и создают скриптовые движки опосредованно. Т.е. при создании экземпляров ScriptEngine-менеджеров они обращаются к механизму поиска сервиса (добавлено в Java 6) для обнаружения всех зарегистрированных реализаций javax.script.ScriptEngineFactory в CLASSPATH.  Эти фабричные классы поставляются в пакетах с реализациями Java scripting API; вам, скорее всего, никогда не придется непосредственно иметь дело с этими классами.

После того, как ScriptEngineManager нашел все фабричные классы соответствующих скриптовых движков, он опрашивает каждую из фабрик чтобы выяснить: можно ли создать движок требуемого типа — JavaScript для случая в [листинге 1](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing1). Если фабрика отвечает утвердительно, менеджер просит фабрику создать движок и он будет возвращен потребителю. Менеджер возвращает null если фабрика для целевого языка не найдена, что предусмотрено в коде [листинга 1](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing1), в целях надежности проверяющем возвращаемое значение на null.

Интерфейс ScriptEngine

Как я уже упоминал, ваш код использует экземпляр ScriptEngine для выполнения скрипта.

**Интерфейс ScriptEngine определяет 6 перегруженных методов eval():**

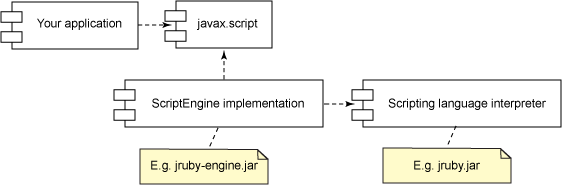
**- 3 принимающих скрипт к обработке как в виде строки *(дальнейшие примеры именно про это)*;**

**- 3 через объект java.io.Reader, для вычитывания скриптов из внешних источников, таких как файлы ( типа scriptEngine.eval(new FileReader(myFile)) )**.

Скриптовый движок действует как посредник между вашим скриптовым кодом и целевым языковым интерпретатором или компилятором, который, в конечном счете, и исполняет код. Следовательно, вам не нужно знать какие классы используются каждым интерпретатором для отработки кода. Например, скриптовый движок для JRuby может сначала передать ваш код в экземпляр класса org.jruby.Ruby для компиляции скрипта в некоторую промежуточную форму, затем вызвать его снова для прогона скрипта и обработки возвращаемых значений. Реализация скриптовой поддержки скрывает детали, включая то, как интерпретатор согласовывает определения классов, объекты приложения и потоки ввода/вывода с Java-кодом.

На рис. 1 в общем виде показаны взаимосвязи между вашим приложением, Java scripting API, реализацией ScriptEngine и интерпретатором скриптового языка. Вы можете отметить, что ваше приложение опирается только на API скриптинга, предоставляемое классом ScriptEngineManager и интерфейсом ScriptEngine. Компонент реализации интерфейса ScriptEngine обслуживает всю специфику использования конкретного языкового интерпретатора.

Рисунок 1: Взаимосвязи компонентов Scripting API



Вас, надо полагать, заинтересует вопрос где же взять необходимые JAR-файлы, имплементирующие скриптовый движок и языковый интерпретатор. Наилучшим местом для поиска реализации движка является, прежде всего, проект с открытым кодом Scripting, поддерживаемый java.net (см. [Ресурсы](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#artrelatedtopics)). Здесь вы найдете реализации скриптовых движков для многих языков и ссылки на прочие ресурсы по теме. Проект Scripting также предоставляет ссылки для загрузки интерпретаторов поддерживаемых скриптовых языков.

В [листинге1](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing1) метод main() передает ScriptEngine в каждый метод для отработки соответствующего JavaScript-кода.

Первый метод приведен в **листинге2**.

Метод invokeHelloScript() вызывает на стороне движка метод eval для вычисления и выполнения переданной строки кода от Java в JavaScript.

**Интерфейс ScriptEngine определяет 6 перегруженных методов eval():**

**- 3 принимающих скрипт к обработке как в виде строки *(дальнейшие примеры именно про это)*;**

**- 3 через объект java.io.Reader, для вычитывания скриптов из внешних источников, таких как файлы**.

Листинг 2. Метод invokeHelloScript

|  |  |
| --- | --- |
| 1  2  3 | private static void invokeHelloScript(ScriptEngine jsEngine) throws ScriptException {      jsEngine.eval("println('Hello from JavaScript')");  } |

JavaScript в методе invokeHelloScript() выводит полученный от Java текст “Hello from JavaScript” в поток стандартного вывода (standard output), являющийся, в данном случае, консольным окном. ([листинг 6](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing6) содержит окончательный вывод после запуска HelloScriptingWorldApplication.)

Обратите внимание - этот и другие методы класса декларируют, что они выбрасывают исключение javax.script.ScriptException. Это проверяемое исключение — единственное, определенное в пакете скриптинга — указывает на то, что движок потерпел неудачу при синтаксическом разборе или выполнении кода. Все методы eval() скриптового движка выбрасывают ScriptException, поэтому в коде вам необходимо соответствующим образом обработать эту ситуацию.

В **листинге3**

показаны два связанных метода: defineScriptFunction() и invokeScriptFunctionFromEngine(). Метод defineScriptFunction() также вызывает на движке метод eval() с явно заданным фрагментом JavaScript-кода. Однако отметьте — этот метод всего лишь задает определение JavaScript-функции sayHello(). Выполнения кода не происходит. Функция sayHello() принимает один параметр, который затем выводится в консоль следующим далее вызовом println(). JavaScript-интерпретатор скриптового движка добавляет эту функцию в свое глобальное окружение, делая ее доступной в последующих вызовах eval, что происходит (и это не удивительно) в методе invokeScriptFunctionFromEngine().

Листинг 3. Методы defineScriptFunction и invokeScriptFunctionFromEngine

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | private static void defineScriptFunction(ScriptEngine engine) throws ScriptException {      // Define a function in the script engine      engine.eval(          "function sayHello(name) {" +          "    println('Hello, ' + name)" +          "}"      );  }    private static void invokeScriptFunctionFromEngine(ScriptEngine engine)    throws ScriptException  {      engine.eval("sayHello('World!')");  } |

Эта пара методов демонстрирует, что скриптовые движки могут сохранять состояние компонентов приложения и делать это состояние доступным на протяжении последующих вызовов методов eval(). Метод invokeScriptFunctionFromEngine() использует преимущество, предоставленное сохраненным состоянием и вызывает JavaScript-функцию, определенную в предыдущем вызове eval().

Итак, многие скриптовые движки сохраняют состояние глобальных переменных и функций между вызовами eval(). Однако, и это важно учитывать, Java Scripting API не требует от движков поддержки такой возможности. Скриптовые движки JavaScript, Groovy и JRuby, используемые в этой статье, *обеспечивают* сохранность состояния между вызовами eval().

**Листинг 4**

является вариацией предыдущего примера. Метод invokeScriptFunctionFromJava() отличается тем, что вызывает JavaScript-функцию sayHello() не прибегая к методу eval(), принадлежащему ScriptEngine или к JavaScript-коду. Вместо этого он использует интерфейс javax.script.Invocable из состава Java Scripting API для вызова функции, поддерживаемой скриптовым движком. Метод invokeScriptFunctionFromJava() приводит объект скриптового движка к интерфейсному типу Invocable, а затем обращается к интерфейсному методу invokeFunction() для вызова JavaScript-функции sayHello() с заданным параметром. Если вызываемая функция возвращает значение, метод invokeFunction() вернет его, упаковав в Java-тип Object.

Листинг 4. Метод invokeScriptFunctionFromJava

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | private static void invokeScriptFunctionFromJava(ScriptEngine engine)    throws ScriptException, NoSuchMethodException  {  Invocable invocableEngine = (Invocable) engine;      invocableEngine.invokeFunction("sayHello", "from Java");  } |

Отметьте также, что [листинг 4](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing4) не содержит JavaScript. Интерфейс Invocable позволяет Java-коду вызывать скриптовую функцию не зная языка ее реализации. Метод invokeFunction() выбрасывает java.lang.NoSuchMethodException если скриптовый движок не смог найти функцию с данным именем или типом параметров.

Java Scripting API не требует от скриптового движка имплементации интерфейса Invocable. На самом деле, код в [листинге4](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing4) должен был бы перед приведением типа задействовать оператор instanceof, чтобы убедиться в том, что движок реализует интерфейс Invocable.

##### Продвинутый вызов скриптов с использованием прокси

Если скриптовая функция или метод реализуют Java-интерфейс — доступно более развитое использование Invocable. Интерфейс Invocable определяет метод getInterface(), принимающий в качестве параметра интерфейс и возвращающий прокси-объект Java, реализующий этот предоставленный ранее интерфейс. Как только вы получили прокси-объект из скриптового движка, вы можете рассматривать его как обыкновенный Java-объект. Вызываемые на прокси методы делегируются в скриптовый движок для выполнения скриптовым языком.

**Вызов Java-методов из скриптового кода**

**Почему-то дальше классы называются Java-объектами, а объекты - переменными**

Примеры в [листинге 3](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing3) и [листинге 4](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing4) показывают как Java-код может вызывать функции или методы, определенные в скриптовом языке. Теперь же вас, вероятно, заинтересует — может ли код скриптового языка, в свою очередь, вызывать методы Java-объектов(классов). Может.

Метод invokeJavaFromScriptFunction() в **листинге 5**

показывает — как получить доступ к Java-объектам(классам) со стороны скриптового движка и как скриптовый код может вызывать методы этих Java-объектов(классов). В частности, метод invokeJavaFromScriptFunction() использует предоставляемый движком метод **put()** для передачи экземпляра того же класса HelloScriptingWorld в движок. После того, как движок получил доступ к Java-объекту через имя, переданное при вызове put(), скриптовый код вызовом метода eval() использует его.

Листинг 5. Методы invokeJavaFromScriptFunction и getHelloReply

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | private static void invokeJavaFromScriptFunction(ScriptEngine engine)    throws ScriptException  {      engine.put("helloScriptingWorld", new HelloScriptingWorld()); //"helloScriptingWorld"  это название созданного объекта(дальше они называют ее переменной)      engine.eval(          "println('Вызываем метод getHelloReply из JavaScript...');" +          "var msg = helloScriptingWorld.getHelloReply(vJavaScript');" +          "println('Получили из Java: ' + msg)"      );  }    /\*\* Метод, вызываемый из вышеприведенного скрипта и возвращающий строку. \*/  public String getHelloReply(String name) {      return "Java-метод getHelloReply говорит 'Привет, " + name + "'";  } |

JavaScript-код, содержащийся в вызове метода eval() из [листинга 5](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing5) использует Java-объект(класс) HelloScriptingWorld посредством доступа к нему через переменную с именем helloScriptingWorld, переданным в вызов метода put() на стороне скриптового движка.

Вторая строка JavaScript-кода вызывает публичный Java-метод getHelloReply(), также приведенный в [листинге 5](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#listing5). Метод getHelloReply() возвращает строку Java-метод getHelloReply говорит 'Привет, <параметр>'. Код JavaScript в методе eval() присваивает возвращаемое из Java значение переменной msg, затем выводит это значение в консоль.

**Методы get() и put()**

ScriptEngine.**put** и связанный с ним метод get() являются основными способами распределения доступа к объектам и данным между Java-кодом и выполняемыми движком скриптами. (Расширенное обсуждение этой темы см. ниже в [Область видимости исполняемого скрипта](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#script-execution-scope).)

Когда вы вызываете на движке метод put(), скриптовый движок ассоциирует второй параметр (произвольный Java-объект(класс)) с заданным строковым ключем.

Большинство скриптовых движков обеспечивают доступность этих Java-объектов(классов) в скриптах посредством заданного имени переменной(объекта). Движки вольны в обращении с передаваемыми вами в метод put() именами. Например, скриптовый движок JRuby делает helloScriptingWorld доступной в Ruby-коде в виде глобальной переменной $helloScriptingWorld, что соответствует синтаксису Ruby для глобальных переменных.

Метод **get()** движка извлекает значения, доступные в скриптовом окружении. В общем случае, каждая глобальная переменная и функция из окружения JavaScript доступны в Java-коде через метод get(). Но для скриптов доступны только те объекты Java, которые заявлены скриптовому движку непосредственно — вызовом put().

Такая возможность доступа и манипулирования Java-объектами(классами) в исполняемом приложении со стороны внешних скриптов является мощной техникой расширения функциональности ваших Java-программ. (Эта техника задействована в примере из [Части 2](http://www.ibm.com/developerworks/java/library/j-javascripting2/) .)

##### Портирование Java-объектов

Когда скриптовый движок делает Java-объект(класс) доступным скрипту, запущенному в исполняющей среде, движок должен упаковать его в объектный тип, подходящий текущему скриптовому языку. Такая упаковка должна выполнять соответствующие преобразования объект-значение, к примеру, допускать использование Java-объекта Integer непосредственно в математических выражениях скриптового языка. Выяснение того, как Java-объекты портируются в скриптовые объекты специфично для каждого скриптового движка и выходит за рамки обсуждаемого в этой статье. И все же вы должны осознавать, что такая трансляция происходит, поэтому вы можете проводить тестирование используемого скриптового языка, чтобы убедиться в предсказуемости выполняемых преобразований.

Запуск приложения HelloScriptingWorld

Вы можете запустить приложение HelloScriptingWorld загрузив и скомпоновав исходный код.

Zip-файл содержит компоновочные сценарии как для Ant так и для Maven, чтобы облегчить компиляцию и запуск примера приложения. Выполните следующие шаги:

1. [Скачайте](https://www.ibm.com/developerworks/ru/library/j-javascripting1/index.html#artdownload) zip-архив.
2. Создайте новый каталог, скажем, java-scripting и распакуйте сюда полученный на предыдущем шаге архив.
3. Откройте командное окно и перейдите в этот каталог.
4. Запустите ant run-hello.

Вы должны увидеть консольный вывод из Ant, сходный с приведенным в **листинге 6.**

Обратите внимание на то, что метод defineScriptFunction() не генерирует вывода, т.к. он определяет, но не вызывает JavaScript-функцию.

Листинг 6. То, что мы увидим на экране после запуска HelloScriptingWorld:

|  |  |
| --- | --- |
|  | Вызываю invokeHelloScript...  Hello from JavaScript    Вызываю defineScriptFunction...    Вызываю invokeScriptFunctionFromEngine...  Hello, World!    Вызываю invokeScriptFunctionFromJava...  Hello, from Java    Вызываю invokeJavaFromScriptFunction...  Вызов метода getHelloReply из JavaScript...  Получили из Java: Java-метод getHelloReply говорит 'Привет, JavaScript' |

Bindings - ограничение области видимости переменных и объектов для данного скрипта *(типа своебразного ClassLoader-а)*.

За тем, как вы передаете Java-объекты(классы) в исполняемые движком скрипты, стоит более развитая реализация, чем просто вызов движковых методов get() и put(). Когда вы вызываете get() или put() на движке, он извлекает или сохраняет требуемый ключ в специально предусмотренном экземпляре интерфейса javax.script.Bindings. (Интерфейс Bindings это просто интерфейс Map, обслуживающий строковые ключи.)

Когда ваш код вызывает движковый метод eval(), на стороне движка используется предопределенное связывание ключей (находящихся в объекте интерфейса javax.script.Bindings) со значениями. Однако, вы можете предоставить **свой собственный объект Bindings** для обслуживания вызовов eval(), чтобы ограничить видимость переменных и объектов для данного скрипта. Тогда вызов будет выглядеть как **eval(String, Bindings)**или**eval(Reader, Bindings)**. Чтобы облегчить создание ваших специфичных Bindings, скриптовые движки предлагают метод createBindings(), возвращающий пустой объект Bindings. Вызов eval на объекте Bindings временно скрывает Java-объекты(классы), сохраненные ранее с использованием предопределенного в движке связывания.

Для накопления истории скриптовый движок имеет в своем составе два предопределенных механизма связывания:

- связывания с областью видимости на уровне движка (*engine scope bindings*) используются при вызовах get() и put(),

- связывания с глобальной областью видимости (*global scope bindings*) движок *может* применять для поиска объектов в случае, если их не удалось обнаружить на уровне связывания "engine scope". Формулировка *может*— существенна. Скриптовые движки не обязаны обеспечивать доступность глобального связывания для скриптов. Хотя многие скриптовые движки такой доступ предоставляют.

Конструктивное назначение "global scope"-связывания — совместное использование объектов различными скриптовыми движками.

Каждый движок, возвращаемый экземпляром ScriptEngineManager, комплектуется одним и тем же объектом глобального связывания.

Так до конца и не понял. Но возможно экземпляр интерфейса javax.script.Bindings (в котором будет храниться ключ) находится в глобальной области видимости (*global scope bindings*), а если мы сами зададим **Bindings,** то тогда наши ключи перестанут находиться в общей глобальной области видимости, а перейдут в нашу созданную глобальную область видимости.

Вы можете извлечь этот объект вызовом метода getBindings(ScriptContext.GLOBAL\_SCOPE) и назначить объект глобального связывания для движка с помощью setBindings(Bindings, ScriptContext.GLOBAL\_SCOPE).

ScriptContext— это интерфейс, который определяет и управляет контекстом времени исполнения скриптового движка. ScriptContext содержит связывания с "движковой" и "глобальной" областями видимости, а также потоки ввода/вывода, используемые движком для стандартных операций ввода/вывода. Вы можете получить контекст скриптового движка и манипулировать им с помощью движкового метода getContext().

Концепции Scripting API, такие как область видимости *scope*, связывания *bindings* и *контекст* могут, поначалу, сбивать с толку, по причине своих частично перекрывающихся смыслов. Загрузочный файл с исходными кодами для этой статьи включает тестовый файл JUnit, называющийся ScriptApiRhinoTest и расположенный в каталоге src/test/java. Содержащийся там Java-код призван помочь вам разобраться с этими концепциями.

Что дальше?

Теперь, когда вы располагаете базовыми знаниями по Java Scripting API, [Часть 2](http://www.ibm.com/developerworks/java/library/j-javascripting2/) этой статьи улучшит и расширит эти знания при помощи более реалистичного примера приложения. Это приложение использует комбинацию внешних скриптовых файлов, написанных на Groovy, Ruby и JavaScript для задания бизнес-логики с возможностью ее оперативного изменения. Как вы увидите, определение бизнес-правил в скриптовом языке облегчает написание этих правил и, возможно, упрощает восприятие при чтении непрограммистами, такими как бизнес-аналитики или специалисты по описанию предметной области.