Часть 2. Оперативный поиск, выполнение и изменение скриптов

Изменяйте бизнес-логику на лету

[Этот контент является частью серии:**Используйте динамические языки динамично**](javascript:void();)

Java Scripting API, появившийся в Java SE 6, предлагает унифицированный подход для запуска (а также предоставления совместного доступа к коду и данным) внешних программ, написанных на различных динамических языках. Усиление Java-приложения благодаря мощности и гибкости скриптового языка особенно полезно в случае, когда скрипт может выполнить задачу более наглядно, просто или лаконично. Но Java Scripting API не просто позволяет добавлять множество скриптовых языков в Java-программу единообразным способом. Он также реализует нахождение скриптов, их чтение и оперативное выполнение. Такие динамические возможности позволяют модифицировать скрипты для изменения логики приложения во время исполнения программы. Эта статья демонстрирует как вы можете использовать Java Scripting API для вызова внешних скриптов и динамически влиять на программную логику. Здесь также рассматриваются проблемы с которыми вы можете столкнуться при интеграции одного или нескольких скриптовых языков в Java-приложение.

[Часть 1](http://www.ibm.com/developerworks/ru/library/j-javascripting1/) знакомит с Java Scripting API используя приложение в стиле Hello World. Более реалистичный пример, который будет здесь приведен, использует Scripting API для создания механизма динамических правил, определяющего правила в виде внешних скриптов, написанных на Groovy, JavaScript и Ruby. Правила принимают решение, могут ли соискатели жилищного займа претендовать на отдельные ипотечные продукты. Определение бизнес-правил средствами скриптового языка упрощает написание этих правил и скорее всего облегчает процесс их чтения для непрограммистов, например, менеджеров по займам. Вынесение правил наружу посредством Java Scripting API также позволяет по ходу выполнения приложения изменять эти правила и добавлять новые ипотечные продукты.

Привет, реальный мир

Этот пример приложения обслуживает типовые задачи по жилищному займу для вымышленной компании Shaky Ground Financial. Индустрия жилищного займа непрерывно выдумывает новые ипотечные продукты и изменяет правила предоставления доступа к ним. Shaky Ground Financial желает не только быстро добавлять и удалять имеющиеся предложения, но и оперативно вносить изменения в бизнес-правила, определяющие доступность каждого продукта для конкретного потребителя.

На выручку приходит Java Scripting API. Приложение состоит из класса ScriptMortgageQualifier ответственного за ответ на вопрос доступен ли данный ипотечный продукт для заемщика, предпринимающего попытку купить конкретную собственность. Класс приведен в **Листинге 1**, он **используется для**:

- по расширению скриптового файла запускает необходимый движок для такого языка,

- делает видимыми для этого движка данные, передаваемы ему от Java,

- движком запускает этот скриптовый файл на выполнение и получает из него результат.

Метод “.put(---)” делает доступными для скрипта, введенные из Java данные(*т.е. их теперь станут видетьт и Java и скрипт одновременно*), для возможности работы с ними/ изменения их в скрипте.

Листинг 1. Класс ScriptMortgageQualifier

|  |  |
| --- | --- |
| 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  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | // “Директивы импорта и комментарии Javadoc не показаны” – странно, а внизу???.  public **class** ScriptMortgageQualifier {      private ScriptEngineManager scriptEngineManager = new ScriptEngineManager();   // метод **qualifyMortgage** будет запускать скриптовый файл необходимым движком, передавать ему входные параметры и получать результат работы скриптового файла  **public MortgageQualificationResult** **qualifyMortgage**(Borrower borrower,         Property property, Loan loan, File **mortgageRulesFile**) throws  FileNotFoundException, IllegalArgumentException, ScriptException  // класс “**MortgageQualificationResult**” описан в Листинге3, содержит одни геттеры/сеттеры и выводит четыре разные типы данных (Boolean, Doble и два String-a). Именно эти данные будут передаваться как результат работы скриптового файла обратно в Java-приложение.      {          ScriptEngine scriptEngine = getEngineForFile(**mortgageRulesFile**); //получаем  необходимый движок в зависимости от расширения файла (*и как следствие*  *скриптового языка*), на котором будет написан используемый скрипт          if (scriptEngine == null)  {              throw new IllegalArgumentException( "По classpath отсутствует скриптовый  движок для обработки файла: " + mortgageRulesFile);           }            // Сделаем параметры доступными в скриптах, добавив их в контекст движка *(это будут* ***входные данные для скрипта****)*         1. scriptEngine.put("borrower", borrower);         2. scriptEngine.put("property", property);         3. scriptEngine.put("loan", loan);           // Создаем объект который будет возвращать скриптовый файл после своей работы и делаем его доступным в  скриптовых файлах. Т.е. передаем скриптовым файлам объект для заполнения и возврата *(****выходные данные из скрипта****)*         4. MortgageQualificationResult scriptResult =new MortgageQualificationResult();         4. scriptEngine.put("result", scriptResult);            // Добавим объект, который скрипты могут использовать для досрочного выхода из обработки.         5. scriptEngine.put("scriptExit", new ScriptEarlyExit());            try {  **scriptEngine.eval(new FileReader(mortgageRulesFile))**;  //это и есть  } основная строка кода, которая запускает скриптовый файл *(с параметрами бизнес-сущности)*  catch (ScriptException e) //тут будет написана работа с исключениями. Нужно писать это  именно так, т.к. это позволит получать исключения от всех  скриптовых языков (даже таких как JavaScript, в котором такого  механизма в интерпретаторе Rhino JavaScript просто нет)  {             // Повторно генерируем исключение,если только это не наше исключение досрочного выхода.              if(e.getMessage() == null ||                   !e.getMessage().contains("ScriptEarlyExitException"))  { throw e; }             // Установим результирующее сообщение скрипта если исключение досрочного выхода вложено.              Throwable t = e.getCause();              while (t != null)  {                  if (t instanceof ScriptEarlyExitException)  {                      scriptResult.setMessage(t.getMessage());                      break;                   }                  t = t.getCause();               }           }            return scriptResult; // выводим результат работы скриптового файла(“scriptResult” значение которого изменил скрипт)      }        /\*\* Возвращает скриптовый движок на основе расширения переданного файла. \*/      private ScriptEngine getEngineForFile(File f) {          String fE = getFileExtension(f);          return scriptEngineManager.getEngineByExtension(fE); // getEngineByExtension() ищет и  создает скриптовый движок для необходимого скриптового языка по введенному расширению      }        /\*\* Возвращает расширение файла или "" если файл не имеет расширения \*/      private String getFileExtension(File file)  {          String scriptName = file.getName();          int dotIndex = scriptName.lastIndexOf('.');            if (dotIndex != -1) {return scriptName.substring(dotIndex + 1); }  else {return "";}       }        /\*\* Внутреннее исключение для случая досрочного завершения ScriptEarlyExit.exit \*/      private static **class** ScriptEarlyExitException extends Exception {          public ScriptEarlyExitException(String msg) {              super(msg);          }      }        /\*\* Объект, передаваемый во все скрипты для сигнализации о досрочном выходе. \*/      private static **class** ScriptEarlyExit {          public void noMessage() throws ScriptEarlyExitException {              throw new ScriptEarlyExitException(null);          }          public void withMessage(String msg) throws ScriptEarlyExitException {              throw new ScriptEarlyExitException(msg);          }      }  } |

Этот класс делегирует все бизнес-решения внешним скриптам.

Каждый скрипт представляет один ипотечный продукт. Код в каждом скриптовом файле содержит бизнес-правила для определения типа заемщика, собственности или ссуды доступных на соискание для этого ипотечного продукта. Если это так, то новые ипотечные продукты могут быть добавлены копированием нового скриптового файла в директорию скриптовых продуктов. Если же меняются бизнес-правила оценки претендента, это учитывается обновлением соответствующего скрипта.

Написание бизнес-правил под ипотечные предложения на скриптовых языках удачно подчеркивает возможности Java Scripting API. Это также показывает, что скриптовый язык иногда может быть легче для прочтения, изменения и понимания, даже для непрограммистов.

Как работает класс ScriptMortgageQualifier

Наиболее важным методом в классе ScriptMortgageQualifier является qualifyMortgage(). Этот метод принимает в качестве параметров:

* заемщика
* приобретаемую собственность
* подробности по займу
* Объект File, содержащий скрипт для выполнения

Задача метода — запустить скриптовый файл с параметрами бизнес-сущности и вернуть результирующий объект, определяющий право заемщика на ипотечный продукт. Код для Borrower, Property и Loan здесь не показан. Это просто сущностные классы, доступные в исходных кодах, прилагаемых к этой статье (см. [Загрузка](https://www.ibm.com/developerworks/ru/library/j-javascripting2/index.html#artdownload)).

При поиске ScriptEngine для запуска скриптового файла метод qualifyMortgage() использует внутренний впомогательный метод getEngineForFile(). Метод getEngineForFile() задействует переменную экземпляра scriptEngineManager: она инициализирована на этапе конструирования класса через вызов ScriptEngineManager() для поиска скриптового движка, способного отработать скрипт с заданным файловым расширением. Метод getEngineForFile() обращается к методу ScriptEngineManager.getEngineByExtension(), выделенному полужирным шрифтом в [листинге 1](https://www.ibm.com/developerworks/ru/library/j-javascripting2/index.html#listing1), для поиска и получения ScriptEngine.

Получив скриптовый движок, qualifyMortgage() связывает входные сущностные параметры с контекстом движка, обеспечивая доступность этих данных в скрипте (т.е. передает данные из Java в скрипт). Первых три вызова scriptEngine.put(), как-раз и выполняют такое связывание.

Четвертый вызов scriptEngine.put() создает новый Java-объект MortgageQualificationResult и разделяет доступ к нему со скриптовым движком. Такой разделяемый объект, возвращаемый qualifyMortgage(), позволяет скрипту передавать результаты своей работы обратно в Java-приложение через установку свойств объекта (т.е. через этот объект мы получим назад данные со скрипта после расчета).

Скрипты взаимодействуют с этим Java-объектом используя глобальную переменную result. Каждый скрипт несет ответственность за сообщение результата Java-программе посредством такого разделяемого скриптового объекта.

Заключительный пятый вызов scriptEngine.put() обеспечивает доступность экземпляра внутреннего вспомогательного класса ScriptEarlyExit, также приведенного в [листинге 1](https://www.ibm.com/developerworks/ru/library/j-javascripting2/index.html#listing1), в скриптах через переменную с именем scriptExit. ScriptEarlyExit определяет два простых метода: withMessage() и noMessage(), их единственная задача выбросить исключение. Если скрипт вызывает scriptExit.withMessage() или scriptExit.noMessage(), метод выбрасывает ScriptEarlyExitException. Скриптовый движок перехватывает это исключение, прекращает отработку скрипта и выбрасывает ScriptException в метод eval(), вызвавший скрипт.

Такой обходной маневр для преждевременного завершения скрипта обеспечивает устойчивый способ выхода из скриптового процесса и возврат в функцию или метод. Не все скриптовые языки поддерживают конструкции для такой цели. В JavaScript, например, вы не можете использовать оператор return при исполнении кода верхнего уровня, наподобие того, как оформлены скрипты ипотечной обработки в текущем примере приложения. Разделяемый объект scriptExit обеспечивает решение проблемы, позволяя скрипту, реализованному на любом языке, завершать работу сразу же, как только скрипт выясняет, что заемщику данный ипотечный продукт не доступен.

Обращение из qualifyMortgage к методу eval скриптового движка, использует блок try/catch для перехвата ScriptException. Исследовав сообщение об ошибке для ScriptException, код в блоке catch определяет, является ли причиной скриптового исключения ScriptEarlyExitException или же другая, специфичная для скриптинга ошибка. Если сообщение об ошибке содержит название ScriptEarlyExitException, код предполагает, что все происходит в рамках допустимого и игнорирует такое исключение.

Такая техника определения скриптовых исключений уровня Java Scripting API по строковым литералам несколько аляповата, но она работает для языковых интерпретаторов Groovy, JavaScript и Ruby, используемых в рассматриваемом примере. Было бы лучше, если бы все реализации скриптовых языков добавляли Java-исключения, выбрасываемые из вызываемого Java-кода, в стек исключений с последующим доступом через метод Throwable.getCause(). Такие интерпретаторы как JRuby и Groovy именно так и поступают, а вот встроенный интерпретатор Rhino JavaScript — нет.

Запускаем код: ScriptMortgageQualifierRunner

Для тестирования класса ScriptMortgageQualifier вы будете использовать:

- тестовые данные, представляющие четырех простых заемщиков,

- экземпляр имущества, которое заемщик попытается приобрести,

- вариант ипотечной ссуды.

Вы запустите заемщика в связке с имуществом и ссудой в комбинации с каждым из трех скриптов, чтобы увидеть бизнес-правила для соответствующего ипотечного продукта, представленного скриптом.

**Листинг 2** *(в котором частично приведена программа ScriptMortgageQualifierRunner)* **используется для:**

- создания тестовых объектов (заполнения данных для тестирования),

- поиска скриптовых файлов в директории,

- запуска найденых скриптовых файлов, передачи в них данных из Java, проведения расчетов на в скрипте и получения из них расчитаных данных посредством класса ScriptMortgageQualifier из [листинга 1](https://www.ibm.com/developerworks/ru/library/j-javascripting2/index.html#listing1).

В классах Borrower, Property, Loan находятся только переменные и геттеры/сеттеры для получения и передачи инфы по ним дальше.

Вспомогательные **6** **методов этой** программы createGoodBorrower(), createAverageBorrower(), createInvestorBorrower(), createRiskyBorrower(), createProperty() и createLoan() не приводятся для экономии места. В этих методах уже введены соответствующие значения для тестирования (*чтобы вручную не вводить*). Их код смотри в Zip-файле 2 в этой-же папке на компе. (*Как по мне, то «по нормальному» эти методы должны находиться в отдельном классе и вызываться сюда из его объекта, чтобы не перегружать и так перегруженный методами класс ScriptMortgageQualifierRunner)*

Листинг 2. Программа ScriptMortgageQualifierRunner

|  |  |
| --- | --- |
| 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  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119 | // Импорт и некоторые вспомогательные методы не показаны.  public **class** ScriptMortgageQualifierRunner {      private static File scriptDirectory;      private static Borrower goodBorrower = createGoodBorrower();      private static Borrower averageBorrower = createAverageBorrower();      private static Borrower investorBorrower = createInvestorBorrower();      private static Borrower riskyBorrower = createRiskyBorrower();      private static Property property = createProperty();      private static Loan loan = createLoan();        /\*\*       \* Метод main - создание экземпляра File по имени директории из командной строки,       \* затем вызываем метод run, если такая директория существует.       \*/      public static void **main**(String[] args) {          if (args.length > 0 && args[0].contains("-help")) {              printUsageAndExit();          }          String dirName;          if (args.length == 0) {              dirName = "."; // Текущая директория.          } else {              dirName = args[0];          }            scriptDirectory = new File(dirName);          if (!scriptDirectory.exists() || !scriptDirectory.isDirectory()) {              printUsageAndExit();          }            run();      }        /\*\*       \* Определяет статус доступности ипотечного займа для четырех тестовых заемщиков       \* посредством отработки всех скриптовых файлов в заданной директории. Каждый скрипт       \* определит, способен ли данный заемщик претендовать на конкретный тип ипотеки.       \*/      public static void run() {          ScriptMortgageQualifier mortgageQualifier = new ScriptMortgageQualifier();            for(;;) { // Для выхода ожидается Ctrl-C  // это делается не бесконечным циклом + методом waitOneMinute(), а SheduledExecutorService !!!!              runQualifications(mortgageQualifier, goodBorrower, loan, property);              runQualifications(mortgageQualifier, averageBorrower, loan, property);                loan.setDownPayment(30000.0); // Задаток 10%              runQualifications(mortgageQualifier, investorBorrower, loan, property);                loan.setDownPayment(10000.0); // Задаток 3 1/3%              runQualifications(mortgageQualifier, riskyBorrower, loan, property);                waitOneMinute();          }      }        /\*\*       \* Вычитывает все скриптовые файлы из scriptDirectory и запускает их, передавая данные по заемщику       \* чтобы выяснить доступность для него/нее каждого ипотечного предложения.       \*/      private static void **runQualifications**(ScriptMortgageQualifier mortgageQualifier,         Borrower borrower, Loan loan, Property property) //где ScriptMortgageQualifier это  класс из Листинг1  {          for (File scriptFile : getScriptFiles(scriptDirectory)) {              // Печатаем инфу по заемщику, ссуде и имуществу.              System.out.println("Processing file: " + scriptFile.getName());              System.out.println("  Borrower: " + borrower.getName());  // в следующих трех строках будут вызываться собственные методы этих классов              System.out.println("  Credit score: " + borrower.getCreditScore());              System.out.println("  Sales price: " + property.getSalesPrice());              System.out.println("  Down payment: " + loan.getDownPayment());                MortgageQualificationResult result = null;              try {                 //Дальше идет основная рабочая строка кода этого метода  // Запускаем скриптовые правила займа для этого заемщика написанные в скриптовом файле.  **result = mortgageQualifier.qualifyMortgage(borrower, property,**  **loan, scriptFile);** //запускаем необходимым движком скриптовый файл,  вызывая метод “qualifyMortgage” из Листинга1              } catch (FileNotFoundException fnfe) {                  System.out.println(                      "Can't read script file: " + fnfe.getMessage()                  );              } catch (IllegalArgumentException e) {                  System.out.println(                      "No script engine available to handle file: " +                      scriptFile.getName()                  );              } catch (ScriptException e) {                  System.out.println(                      "Script '" + scriptFile.getName() +                      "' encountered an error: " + e.getMessage()                  );              }                if (result == null) continue; // Обязательно учитываем исключения.    //Результат работы этого метода              // Печать результата.              System.out.println(                  "\* Mortgage product: " + result.getProductName() +                  ", Qualified? " + result.isQualified() +                  "\n\* Interest rate: " + result.getInterestRate() +                  "\n\* Message: " + result.getMessage()              );              System.out.println();          }      }        /\*\*       \* Возвращает файлы с точкой, кроме случаев, когда '.' - первый или последний символ.       \*/      private static File[] getScriptFiles(File directory) {          return directory.listFiles(new FilenameFilter() {              public boolean accept(File dir, String name) {                  int indexOfDot = name.indexOf('.');                  // Игнорируем файлы без точки или же с точкой в первой или последней позиции имени.                  if (indexOfDot < 1 || indexOfDot == (name.length() - 1)) {                      return false;                  } else {                      return true;                  }              }          });      }        /\*\*       \* Заставляет ждать 1 минуту.       \*/      private static void waitOneMinute() {          System.out.println(              "\nSleeping for one minute before reprocessing files." +              "\nUse Ctrl-C to exit..."          );          System.out.flush();          try {              Thread.sleep(1000 \* 60);          } catch (InterruptedException e) {              System.exit(1);          }      }  } |

Метод main() в ScriptMortgageQualifierRunner ищет по параметру из командной строки директорию, из которой необходимо читать скриптовые файлы, инициализирует статическую переменную объектом File, если директория существует и вызывает метод run() для выполнения дальнейшей обработки.

Метод run() создает экземпляр класса ScriptMortgageQualifier из [листинга 1](https://www.ibm.com/developerworks/ru/library/j-javascripting2/index.html#listing1), затем в бесконечном цикле вызывает внутренний метод runQualifications() с четырьмя сценариями заемщик/заем. Бесконечный цикл имитирует продолжительное функционирование ипотечного приложения. Бесконечный цикл позволяет добавлять или изменять скрипты (продукты ипотечного заема) в рабочей директории и отслеживать эти изменения динамически, без остановки приложения (хотя это должно делаться через ScheduledExecutorService, а не через написание и использование дополнительного метода waitOneMinute()!!!). Динамическая возможность изменения бизнес-логики в процессе выполнения обеспечивается тем фактом, что бизнес-логика приложения живет во внешних скриптах.

Вспомогательный метод runQualifications() на самом деле вызывает ScriptMortgageQualifer.qualifyMortgage, по одному вызову для каждого скриптового файла, обнаруженного в скриптовой директории. Каждый вызов предваряется операторами вывода описания скриптового файла и текущего заемщика, затем следуют остальные операторы вывода, показывающие, может ли заемщик претендовать на очередной ипотечный продукт. Квалификационные итоги определяются атрибутами разделяемого Java-объекта MortgageQualificationResult, используемого скриптовым кодом для возврата результатов.

ZIP-файл с исходными кодами для этой статьи содержит простые скриптовые файлы, написанные на Groovy, JavaScript и Ruby. Каждый представляет различный тип стандартного 30-летнего ипотечного займа с фиксированным процентом по ссуде. Код в скриптах определяет, квалифицируется ли заемщик на этот тип ипотеки и возвращает результат вызовом методов на разделяемой глобальной переменной result, внесенной в скриптовый движок методом put(), описанным мною ранее. Глобальная переменная result является экземпляром класса MortgageQualificationResult, частично приведенного в Листинге 3, в котором тоже одни геттеры/сеттеры.

Листинг 3. Результирующий класс MortgageQualificationResult

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class MortgageQualificationResult {      private boolean qualified;      private double interestRate;      private String message;      private String productName;        // .. Стандартные setters/getters не показаны.  } |

Скрипты устанавливают свойства result, приведен ные в листинге 3, для возврата данных о праве заемщика на ипотеку и о размере процентной ставки.

Свойства message и productName позволяют скрипту указать причину, по которой заемщику отказано в ипотеке и вернуть имя соответствующего продукта.

Скриптовые файлы

Прежде чем я покажу вывод от запуска ScriptMortgageQualifierRunner, давайте взглянем на скриптовые файлы Groovy, JavaScript и Ruby, вызываемые программой:

- Бизнес-логика в Groovy-скрипте определяет ипотечный продукт, на который относительно просто квалифицироваться, но для него заявлена повышенная процентная ставка, отражающая более высокий финансовый риск.

- JavaScript-скрипт представляет спонсируемый государством ипотечный заем, ориентированный на максимальную доходность и прочие ограничения, которым должен соответствовать заемщик.

- Ипотечный продукт в Ruby-скрипте содержит бизнес-правила, ограничивающие квалификационные возможности заемщика хорошей кредитной историей и гарантированной способностью оплатить задаток. В качестве поощрения предлагается пониженная процентная ставка.

Листинг 4 демонстрирует Groovy-скрипт, с которым вы сможете разобраться даже не зная Groovy.

Листинг 4. Ипотечный Groovy-скрипт

|  |  |
| --- | --- |
| 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  26  27  28  29  30  31  32  33  34 | /\*     Этот скрипт на Groovy определяет продукт "Groovy Ипотека".     Данный продукт предъявляет облегченные требования к заемщику.     Повышенная процентная ставка призвана компенсировать занижение стандартов соответствия.     Все заемщики будут квалифицированы если у них приемлемая кредитная история,     они в состоянии оплатить задаток хотя бы в 5% и они или имеют доход более     $2,000 в месяц, или владеют имуществом, стоимость которого (за вычетом обязательств)     порядка $25,000.  \*/    // Наименование нашего продукта.  result.productName = 'Groovy Mortgage'    // Проверим минимальный доход и стоимость имущества  def netWorth = borrower.totalAssets - borrower.totalLiabilities  if (borrower.monthlyIncome < 2000 && netWorth < 25000) {      scriptExit.withMessage "Low monthly income of ${borrower.monthlyIncome}" +         ' requires a net worth of at least $25,000.'  }    def downPaymentPercent = loan.downPayment / property.salesPrice \* 100  if (downPaymentPercent < 5) {      scriptExit.withMessage 'Down payment of ' +          "${String.format('%1$.2f', downPaymentPercent)}% is insufficient." +          ' 5% minimum required.'  }  if (borrower.creditScore < 600) {      scriptExit.withMessage 'Credit score of 600 required.'  }    // Кто-то квалифицировался. Определим процентную ставку на основании процента задатка.  result.qualified = true  result.message = 'Groovy! You qualify.'    switch (downPaymentPercent) {      case 0..5:   result.interestRate = 0.08; break      case 6..10:  result.interestRate = 0.075; break      case 11..15: result.interestRate = 0.07; break      case 16..20: result.interestRate = 0.065; break      default:     result.interestRate = 0.06; break  } |

Обратите внимание на глобальные переменные result, borrower, loan и property, используемые скриптом для доступа и установки значений в разделяемых Java-объектах. Это имена тех переменных, которые были назначены вызовом метода ScriptEngine.put().

Также отметьте такие конструкции Groovy как result.productName = 'Groovy Mortgage'. Здесь напрямую устанавливается строковое свойство productName объекта MortgageQualificationResult, даже невзирая на то, что в [листинге 3](https://www.ibm.com/developerworks/ru/library/j-javascripting2/index.html#listing3) четко показано, что это приватная переменная экземпляра класса. Это *не* коварство Java Scripting API, допускающего нарушение принципов инкапсуляции. Наоборот, Groovy и большинство других интерпретаторов скриптовых языков, доступных вам через Java Scripting API, достаточно хорошо взаимодействуют с разделяемыми Java-объектами. Если оператор Groovy намеревается установить или прочитать значение приватного свойства некоторого Java-объекта, Groovy подыскивает и использует публичные методы типа setter/getter в JavaBean-стилистике, если таковые определены. Например, выражение result.productName = 'Groovy Mortgage' автоматически транслируется в легко предсказуемый Java-оператор: result.setProductName("Groovy Mortgage"). Такой эквивалентный Java-setter также допустим в Groovy и также смог бы замечательно работать в скрипте, но прямое использование оператора присвоения для свойства считается "Groov-астее".

Теперь перейдем к JavaScript-продукту ипотеки в листинге 5. JavaScript-ипотека призвана презентовать спонсируемый государством заем для поддержки домовладельцев. Бизнес-правила в данном случае требуют, чтобы у заемщика это было первое приобретение недвижимости и чтобы заемщик планировал использовать ее именно для проживания, а не, скажем, для сдачи в аренду с целью получения прибыли.

Листинг 5. Ипотечный JavaScript-скрипт

|  |  |
| --- | --- |
| 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  26  27  28  29  30  31  32  33 | **result.productName** = 'JavaScript FirstTime Mortgage'    if (!borrower.intendsToOccupy) {  **result.message** = 'This mortgage is not intended for investors.'      scriptExit.noMessage()  }  if (!borrower.firstTimeBuyer) {  **result.message** = 'Only first-time home buyers qualify for this mortgage.'      scriptExit.noMessage()  }  if (borrower.monthlyIncome > 4000) {  **result.message** = 'Monthly salary of $' + borrower.monthlyIncome +          ' exceeds the $4,000 maximum.'      scriptExit.noMessage()  }  if (borrower.creditScore < 500) {  **result.message** = 'Your credit score of ' + borrower.creditScore +          ' does not meet the 500 requirement.'      scriptExit.noMessage()  }    // Квалифицировались. Определим процентную ставку,  //основываясь на размере займа и кредитных баллах.  **result.qualified** = true  **result.message** = 'Congratulations, you qualify.'    if (loan.loanAmount > 450000) {      // Большие займы и неважная кредитная история требуют более высокой ставки.  **result.interestRate** = 0.08  } else if (borrower.creditScore < 550) {  **result.interestRate** = 0.08  } else if (borrower.creditScore < 600) {  **result.interestRate** = 0.07  } else if (borrower.creditScore < 700) {  **result.interestRate** = 0.065  } else { // Хорошая кредитная история дает лучшую ставку.  **result.interestRate** = 0.06  } |

Обратите внимание: JavaScript-код не может использовать Java-метод scriptExit.withMessage(), задействованный Groovy-скриптом для установки сообщения о дисквалификации и выхода из скрипта в одном выражении. Причина в том, что интерпретатор Rhino JavaScript не обеспечивает всплывания (*bubble-up*) выброшенного Java-исключения как вложенной "причины" в результирующей трассировке стека ScriptException. Таким образом, весьма затруднительно отыскать в стеке сообщение об исключении на уровне скрипта, выброшенное из Java-кода. Поэтому JavaScript-код в [листинге 5](https://www.ibm.com/developerworks/ru/library/j-javascripting2/index.html#listing5) должен установить причину в результирующем сообщении отдельно, перед вызовом scriptExit.noMessage(), чтобы затем возбудить исключение, приводящее к завершению работы скрипта.

Третий и заключительный ипотечный продукт, показанный в листинге 6, написан на Ruby. Продукт предназначен для заемщиков с хорошей кредитной историей, способных оплатить 20%-ный задаток.

Листинг 6. Ипотечный Ruby-скрипт

|  |  |
| --- | --- |
| 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  26  27  28  29 | # Этот скрипт на Ruby определяет продукт "Ruby Mortgage".  # Он предназначен для заемщиков премиум-класса с пониженной ставкой  # и 20%-ным требованием по задатку.    # Наименование нашего продукта  $result.product\_name = 'Ruby Mortgage'    # Заемщики с никуда не годящейся кредитной историей не квалифицируются.  if $borrower.credit\_score < 700      $scriptExit.with\_message "Credit score of #{$borrower.credit\_score}" +          " is lower than 700 minimum"  end    $scriptExit.with\_message 'No bankruptcies allowed' if $borrower.hasDeclaredBankruptcy    # Проверим прочие минусы  down\_payment\_percent = $loan.down\_payment / $property.sales\_price \* 100  if down\_payment\_percent < 20      $scriptExit.with\_message 'Down payment must be at least 20% of sale price.'  end    # Заемщик квалифицирован. Определим ставку займа  $result.message = "Qualified!"  $result.qualified = true    # Даем наилучшую ставку за минимальные кредитные риски.  if $borrower.credit\_score > 750 || down\_payment\_percent > 25      $result.interestRate = 0.06  elsif $borrower.credit\_score > 700 && $borrower.totalAssets > 100000      $result.interestRate = 0.062  else      $result.interestRate = 0.065  end |

**Не забудьте про $ в JRuby 1.0**

Важно помнить о синтаксисе для глобальных переменных в Ruby, когда вы используете доступ к разделяемым Java-объектам из Ruby-скриптов. Если вы опустите $ у глобальных переменных, JRuby 1.0 и текущий бинарный релиз JRuby 1.0.1 выбросят RaiseException без какой-либо информации о проблеме. Этот баг исправлен в репозитории с исходниками JRuby, а значит он будет исправлен и в последующем бинарном релизе.

Просматривая листинг 6, отметьте, что разделяемые Java-объекты, помещенные в пространство движка, требуют доступа из Ruby с помощью $ в начале имени. Таков синтаксис Ruby для глобальных переменных. Скриптовые движки со скриптовыми языками разделяют Java-переменные как глобальные, поэтому в Ruby должен использоваться соответствующий синтаксис.

Также обратите внимание в листинге и на то, как JRuby автоматически конвертирует "Ruby-змы" в "Java-измы" при вызове разделяемых Java-объектов. Например, если JRuby видит, что вызываемый на Java-объекте метод следует принятому в Ruby соглашению по разделению слов подчеркиваниями типа $result.product\_name = 'Ruby Mortgage', JRuby ищет в качестве альтернативы имя метода со смешанным регистром, если не удалось отыскать исходное имя с подчеркиваниями. Таким образом, оформленное в стилистике Ruby имя метода product\_name= корректно оттранслируется в Java-вызов result.setProductName("Ruby Mortgage").

Программный вывод

Теперь взглянем на вывод программы ScriptMortgageQualifierRunner при ее запуске с тремя скриптовыми файлами ипотечных продуктов. Вы можете запустить эту программу, используя Ant-скрипт из состава предлагаемых к загрузке исходников.

Если же вы предпочитаете Maven, файл README.txt из загрузочного ZIP-архива содержит инструкции по построению и запуску проекта под Maven.

Команда для Ant такая: ant run. Задание run гарантирует, что скриптовые движки и языковые JAR-файлы доступны через classpath. Листинг 7 показывает результат вывода для Ant.

Листинг 7. Программный вывод для Ant

|  |  |
| --- | --- |
| 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  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102 | > ant run  Buildfile: build.xml    compile:      [mkdir] Created dir: C:\temp\script-article\build-main\classes      [javac] Compiling 10 source files to C:\temp\script-article\build-main\classes    run:       [java] Processing file: GroovyMortgage.groovy       [java]   Borrower: Good Borrower       [java]   Credit score: 800       [java]   Sales price: 300000.0       [java]   Down payment: 60000.0       [java] \* Mortgage product: Groovy Mortgage, Qualified? true       [java] \* Interest rate: 0.06       [java] \* Message: Groovy! You qualify.         [java] Processing file: JavaScriptFirstTimeMortgage.js       [java]   Borrower: Good Borrower       [java]   Credit score: 800       [java]   Sales price: 300000.0       [java]   Down payment: 60000.0       [java] \* Mortgage product: JavaScript FirstTime Mortgage, Qualified? false       [java] \* Interest rate: 0.0       [java] \* Message: Only first-time home buyers qualify for this mortgage.         [java] Processing file: RubyPrimeMortgage.rb       [java]   Borrower: Good Borrower       [java]   Credit score: 800       [java]   Sales price: 300000.0       [java]   Down payment: 60000.0       [java] \* Mortgage product: Ruby Mortgage, Qualified? true       [java] \* Interest rate: 0.06       [java] \* Message: Qualified!         [java] Processing file: GroovyMortgage.groovy       [java]   Borrower: Average Borrower       [java]   Credit score: 700       [java]   Sales price: 300000.0       [java]   Down payment: 60000.0       [java] \* Mortgage product: Groovy Mortgage, Qualified? true       [java] \* Interest rate: 0.06       [java] \* Message: Groovy! You qualify.         [java] Processing file: JavaScriptFirstTimeMortgage.js       [java]   Borrower: Average Borrower       [java]   Credit score: 700       [java]   Sales price: 300000.0       [java]   Down payment: 60000.0       [java] \* Mortgage product: JavaScript FirstTime Mortgage, Qualified? false       [java] \* Interest rate: 0.0       [java] \* Message: Monthly salary of $4500 exceeds the $4,000 maximum.         [java] Processing file: RubyPrimeMortgage.rb       [java]   Borrower: Average Borrower       [java]   Credit score: 700       [java]   Sales price: 300000.0       [java]   Down payment: 60000.0       [java] \* Mortgage product: Ruby Mortgage, Qualified? true       [java] \* Interest rate: 0.065       [java] \* Message: Qualified!         [java] Processing file: GroovyMortgage.groovy       [java]   Borrower: Investor Borrower       [java]   Credit score: 720       [java]   Sales price: 300000.0       [java]   Down payment: 30000.0       [java] \* Mortgage product: Groovy Mortgage, Qualified? true       [java] \* Interest rate: 0.06       [java] \* Message: Groovy! You qualify.         [java] Processing file: JavaScriptFirstTimeMortgage.js       [java]   Borrower: Investor Borrower       [java]   Credit score: 720       [java]   Sales price: 300000.0       [java]   Down payment: 30000.0       [java] \* Mortgage product: JavaScript FirstTime Mortgage, Qualified? false       [java] \* Interest rate: 0.0       [java] \* Message: This mortgage is not intended for investors.         [java] Processing file: RubyPrimeMortgage.rb       [java]   Borrower: Investor Borrower       [java]   Credit score: 720       [java]   Sales price: 300000.0       [java]   Down payment: 30000.0       [java] \* Mortgage product: Ruby Mortgage, Qualified? false       [java] \* Interest rate: 0.0       [java] \* Message: Down payment must be at least 20% of sale price.         [java] Processing file: GroovyMortgage.groovy       [java]   Borrower: Risk E. Borrower       [java]   Credit score: 520       [java]   Sales price: 300000.0       [java]   Down payment: 10000.0       [java] \* Mortgage product: Groovy Mortgage, Qualified? false       [java] \* Interest rate: 0.0       [java] \* Message: Down payment of 3.33% is insufficient. 5% minimum required.         [java] Processing file: JavaScriptFirstTimeMortgage.js       [java]   Borrower: Risk E. Borrower       [java]   Credit score: 520       [java]   Sales price: 300000.0       [java]   Down payment: 10000.0       [java] \* Mortgage product: JavaScript FirstTime Mortgage, Qualified? true       [java] \* Interest rate: 0.08       [java] \* Message: Congratulations, you qualify.         [java] Processing file: RubyPrimeMortgage.rb       [java]   Borrower: Risk E. Borrower       [java]   Credit score: 520       [java]   Sales price: 300000.0       [java]   Down payment: 10000.0       [java] \* Mortgage product: Ruby Mortgage, Qualified? false       [java] \* Interest rate: 0.0       [java] \* Message: Credit score of 520 is lower than 700 minimum           [java] Sleeping for one minute before reprocessing files.       [java] Use Ctrl-C to exit... |

Вывод содержит 12 секций, так как программа направляет каждый из четырех тестовых профилей заемщиков ссуды в каждый из трех скриптов для выяснения возможности заемщика квалифицироваться на любой (или на все) из трех ипотечных продуктов. Далее, для демонстрационных целей в рамках данной статьи, программа приостанавливается на одну минуту, затем повторяет отработку ипотечных скриптов. Во время паузы вы можете редактировать любой из скриптовых файлов для изменения бизнес-правил или же добавлять новые файлы в скриптовую директорию, чтобы презентовать ипотечные продукты собственного изготовления. Программа сканирует скриптовую директорию на каждом проходе и отрабатывает любой из обнаруженных скриптовых файлов.

Например, предположим, что вы хотите повысить минимальный уровень кредитных баллов, необходимых для прохождения квалификации на заем. В течение минутной паузы вы могли бы отредактировать скрипт JavaScriptFirstTimeMortgage.js из директории src/main/scripts/mortgage-products (см. [листинг 5](https://www.ibm.com/developerworks/ru/library/j-javascripting2/index.html#listing5)) для изменения бизнес-правила в строке 23 с if (borrower.creditScore < 500) { на if (borrower.creditScore < 550) {. При следующем запуске правил вы заметите, что Risk E. Borrower более не квалифицируется на "JavaScript FirstTime Mortgage". Для этого заемщика стало бы недостаточно 520 кредитных баллов. Сообщение об ошибке по-прежнему гласило бы: "Количество ваших кредитных баллов в 520 единиц не соответствует требованию в 500", но вы смогли бы сразу же подправить и это устаревшее на данный момент сообщение.

Как избежать опасностей динамического скриптинга

Возможность изменять ваши программы во время их исполнения неоценима. И потенциально опасна. Новая функциональность и новые бизнес-правила могут быть добавлены в работающее приложение без его остановки и перезапуска. Таким же образом новые баги, особенно потенциально фатальные, могут быть добавлены с той же легкостью.

Однако динамическое изменение работающего приложения не более опасно, чем изменение остановленного. Статическая техника подразумевает лишь то, что вы должны перезапустить приложение перед поиском этих новых ошибок. Хорошая практика разработки программного обеспечения диктует, что любое изменение — динамическое или статическое — в результирующем приложении должно быть предварительно протестировано, прежде чем оно будет внедрено. Java Scripting API не отменяет этого правила.

Внешние скриптовые файлы могуть быть протестированы в процессе разработки в рамках регулярного модульного тестирования. Вы можете задействовать JUnit или же другую тестовую оснастку с подставными (*mock*) Java-объектами, необходимыми скрипту в процессе работы, затем выполнить скрипт с такими подстановками, чтобы убедиться, что скрипт отрабатывает корректно и продуцирует ожидаемые результаты. Воплощение логики приложения на не-Java скриптовых файлах не дает нам права отказаться от тестирования этих скриптов.

Если в прошлом или же на данный момент вы являетесь программистом скриптов Web CGI, для вас не будет сюрпризом необходимость соблюдать осторожность в том, что вы передаете в метод eval() объекта ScriptEngine. Скриптовый движок немедленно выполнит переданный в метод eval код. Поэтому вам не следует передавать строку или объект Reader, содержащие текст из ненадежного источника, в скриптовый движок на выполнение.

Например, вполне возможно использовать скриптовый API для удаленного мониторинга Web-приложения. Вы можете предоставить скриптовому движку доступ к ключевым Java-объектам, содержащим статусную информацию о Web-приложении и создать простую Web-страницу, принимающую произвольное скриптовое выражение для отработки скриптовым движком с последующими вычислениями и выводом результата обратно на Web-страницу. Таким способом вы могли бы запросить и выполнить методы на запущенных Java-объектах для простого определения статуса и работоспособности приложения.

В приведенном сценарии кто угодно, имея доступ к такой Web-странице, смог бы выполнить любые выражения, доступные в скриптовом языке и добраться до любого из разделяемых Java-объектов. Небрежное программирование, конфигурационные просчеты или бреши в системе безопасности могут привести к утечке конфиденциальной информации к неуполномоченным пользователям или же подставить ваше приложение под атаку "отказа в обслуживании" (*denial-of-service attack*), если хакер выполнит скриптовую конструкцию вроде System.exit или /bin/rm -fr /. Как и любой другой мощный инструмент, Java Scripting API требует от вас соблюдения определенной осторожности.

Дальнейшее изучение

Эта статья сфокусирована на способности Java-приложений динамически читать и оперативно выполнять внешние скрипты, а также возможностях доступа из таких скриптов к Java-объектам, явным образом опубликованным для этой цели. Java Scripting API предоставляет дополнительные возможности. Например, вы можете:

* Имплементировать Java-интерфейсы, используя скриптовый язык, и вызывать скриптовый код из Java-кода, как если бы это была любая другая ссылка на Java-интерфейс.
* Создавать экземпляры и использовать Java-объекты внутри скриптов, потенциально предоставляя этим объектам возможность быть доступными позже в самом Java-приложении.
* Предварительно компилировать динамические скрипты при загрузке с целью ускорения последующего вызова на выполнение.
* Назначать потоки ввода/вывода для использования скриптом, что облегчает как применение файла в качестве источника консольного ввода для скрипта, так и перенаправление консольного вывода в файл или другой поток.
* Задавать позиционные параметры, которые скрипт сможет использовать как аргументы командной строки.

Java Scripting API определяет некоторые из этих возможностей как опциональные для разработчиков скриптовых движков, поэтому не все скриптовые движки поддерживают их. См. [Ресурсы](https://www.ibm.com/developerworks/ru/library/j-javascripting2/index.html#artrelatedtopics) для дальнейшего ознакомления и онлайновые руководства по этому и прочим аспектам.