# 動的言語を動的に呼び出す: 第 2 回 実行時にスクリプトを発見し、実行し、そして変更する

## ビジネス・ロジックを即時処理で変更する

Tom McQueeney (tom.mcqueeney@gmail.com)
Lead Technical Consultant
Idea Integration

2007年 9月 11日

Java SE 6 に追加された Java™ スクリプト API、そして Java SE 5 との後方互換性によって、非常にさまざまな種類のスクリプト言語を、実行中の Java アプリケーションから単純で統一された方法で呼び出すことができます。この 2 回シリーズの記事の『第 1 回』では、この API の基本的な機能を紹介しています。第 2 回では、この API の強力さを、さらに詳しく紹介します。Ruby と Groovy、そして JavaScript で記述された外部スクリプトをプログラムの実行時に実行でき、また変更できることを説明し、それによってアプリケーションを停止して再起動せずにビジネス・ロジックを変更できることを示します。

このシリーズの他の記事を見る

Java SE 6 に追加された Java スクリプト API によって、さまざまな動的言語で記述された外部プログラムを統一した方法で実行する (またコードとデータを共有する) ことができます。スクリプト言語の強力さと柔軟性を活用して Java アプリケーションを増強することは、特にスクリプト言語の方がより明確に、または単純に、あるいは簡潔に作業を実行できる場合には、非常に有効です。しかし Java スクリプト API は、単に統一された方法で Java プログラムに非常にさまざまな種類のスクリプト言語を追加するためだけのものではありません。この API によって、プログラムの実行時にスクリプトを発見し、読み取り、そして実行できるのです。こうした動的機能を利用すると、プログラムの実行中にスクリプトを変更してアプリケーションのロジックを変更することができます。この記事では、Java スクリプト API を使って外部スクリプトを呼び出し、プログラムのロジックを動的に変更する方法を説明します。また、1 つ以上のスクリプト言語を 1 つの Java アプリケーションに統合する際に直面する可能性のある問題についても調べます。

第 1 回では Hello World スタイルのアプリケーションを使って Java スクリプト API を紹介しています。今回紹介する現実的なサンプル・アプリケーションでは、Groovy と JavaScript、そして Ruby で記述された外部スクリプトとしてルールを定義する動的なルール・エンジンを、Java スクリプト API を使って作成します。これらのルールによって、住宅ローンの申込者が特定の不動産担保ローン商品に適格かどうかを判断します。スクリプト言語を使ってビジネス・ルールを定義すると、ルールを作成しやすくなり、またプログラマーではない人 (ローンの審査担当者など) にとっ

て、おそらくルールが読みやすくなります。また Java スクリプト API を利用してルールを外部化すると、アプリケーションが実行中でもルールを変更したり新しい不動産担保ローン商品を追加したりできるようになります。

## Hello World よりも現実的に

このサンプル・アプリケーションは、Shaky Ground Financial という架空の会社に対する住宅ローンの申し込みを処理します。住宅用貸し付けの業界は常に新しいローン商品を作り出しており、またそうした商品への適格者に対するルールも頻繁に変更しています。Shaky Ground は、不動産担保ローン商品を迅速に追加、削除することの他に、各商品への適格者を判断するためのビジネス・ルールも迅速に変更したいと思っています。

そこに Java スクリプト API が救いに登場します。サンプル・アプリケーションは `ScriptMortgageQualifier` クラスで構成されており、このクラスが、ある特定の物件を購入しようとする借り手が指定の不動産担保ローン商品に適格かどうかを判断します。このクラスをリスト 1 に示します。

### リスト 1. ScriptMortgageQualifier クラス

```
// Imports and Javadoc not shown.
public class ScriptMortgageQualifier {
    private ScriptEngineManager scriptEngineManager = new ScriptEngineManager();

    public MortgageQualificationResult qualifyMortgage(
        Borrower borrower,
        Property property,
        Loan loan,
        File mortgageRulesFile
    ) throws FileNotFoundException, IllegalArgumentException, ScriptException
    {
        ScriptEngine scriptEngine = getEngineForFile(mortgageRulesFile);
        if (scriptEngine == null) {
            throw new IllegalArgumentException(
                "No script engine on classpath to handle file: " + mortgageRulesFile
            );
        }

        // Make params accessible to scripts by adding to engine's context.
        scriptEngine.put("borrower", borrower);
        scriptEngine.put("property", property);
        scriptEngine.put("loan", loan);

        // Make return-value object available to scripts.
        MortgageQualificationResult scriptResult = new MortgageQualificationResult();
        scriptEngine.put("result", scriptResult);

        // Add an object scripts can call to exit early from processing.
        scriptEngine.put("scriptExit", new ScriptEarlyExit());

        try {
            scriptEngine.eval(new FileReader(mortgageRulesFile));
        } catch (ScriptException se) {
            // Re-throw exception unless it's our early-exit exception.
            if (se.getMessage() == null ||
                !se.getMessage().contains("ScriptEarlyExitException")
            ) {
                throw se;
            }
            // Set script result message if early-exit exception embedded.
            Throwable t = se.getCause();
            while (t != null) {
```

```
                if (t instanceof ScriptEarlyExitException) {
                    scriptResult.setMessage(t.getMessage());
                    break;
                }
                t = t.getCause();
            }
        }

        return scriptResult;
    }

    /** Returns a script engine based on the extension of the given file. */
    private ScriptEngine getEngineForFile(File f) {
        String fileExtension = getFileExtension(f);
        return scriptEngineManager.getEngineByExtension(fileExtension);
    }

    /** Returns the file's extension, or "" if the file has no extension */
    private String getFileExtension(File file) {
        String scriptName = file.getName();
        int dotIndex = scriptName.lastIndexOf('.');

        if (dotIndex != -1) {
            return scriptName.substring(dotIndex + 1);
        } else {
            return "";
        }
    }

    /** Internal exception so ScriptEarlyExit.exit can exit scripts early */
    private static class ScriptEarlyExitException extends Exception {
        public ScriptEarlyExitException(String msg) {
            super(msg);
        }
    }

    /** Object passed to all scripts so they can indicate an early exit. */
    private static class ScriptEarlyExit {
        public void noMessage() throws ScriptEarlyExitException {
            throw new ScriptEarlyExitException(null);
        }
        public void withMessage(String msg) throws ScriptEarlyExitException {
            throw new ScriptEarlyExitException(msg);
        }
    }
}
```

このクラスが比較的小さい理由は、すべてのビジネス判断を外部スクリプトに委任しているためです。各スクリプトは 1 つの不動産担保ローン商品を表します。各スクリプト・ファイルのコードは、その不動産担保ローン商品に適格な借り手や物件、あるいはローンのタイプを定義するビジネス・ルールを含んでいます。こうしておけば、スクリプトによる商品ディレクトリーに新しいスクリプト・ファイルをコピーすることで、新しい不動産担保ローン商品を追加することができます。ある特定の不動産担保ローン商品の適格者に関するビジネス・ルールを変更する場合には、スクリプトを更新してそうした変更を反映させることができます。

不動産担保ローン商品のビジネス・ルールをスクリプト言語で作成すると、Java スクリプト API の機能がよくわかります。また、場合によるとプログラマーではない人にとってもスクリプト言語の方が読みやすく、修正しやすく、そして理解しやすいこともわかるはずです。

## ScriptMortgageQualifier クラスの動作

`ScriptMortgageQualifier` クラスの基本的なメソッドは `qualifyMortgage()` です。このメソッドはパラメーターとして下記を受け取ります。

- 借り手 (borrower)
- 購入対象の物件 (property)
- ローン (loan) の詳細
- 実行されるスクリプトを含む File オブジェクト (mortgageRulesFile)

このメソッドの仕事は、ビジネス・エンティティーのパラメーターを持つスクリプト・ファイルを実行し、その借り手がその不動産担保ローン商品に適格かどうかを示す結果オブジェクトを返すことです。ここでは、`Borrower` と `Property`、そして `Loan` のコードは示してありません。これらは単純なエンティティー・クラスであり、この記事のソース・コードの中に含まれています (「ダウンロード」を参照)。

`qualifyMortgage()` メソッドは、スクリプト・ファイルを実行する `ScriptEngine` を見つけるために、内部ヘルパー・メソッド `getEngineForFile()` を使います。`getEngineForFile()` メソッドはインスタンス変数 `scriptEngineManager` (ScriptEngineManager でクラスを作成するときに初期化されます) を使って、指定のファイル拡張子を持つスクリプトを処理できるスクリプト・エンジンを見つけます。`getEngineForFile()` メソッドは、リスト 1 に太字で示した `ScriptEngineManager.getEngineByExtension()` メソッドを使って `ScriptEngine` を探し、そしてそれを返します。

`qualifyMortgage()` は、スクリプト・エンジンが見つかると、受信されるエンティティー・パラメーターをエンジンのコンテキストにバインドし、そのスクリプトで利用できるようにします。最初の 3 回の `scriptEngine.put()` の呼び出し (これも太字で示してあります) では、これらのバインディングを実行します。4 回目の `scriptEngine.put()` の呼び出しでは新しい Java オブジェクト `MortgageQualificationResult` を作成し、これをスクリプト・エンジンと共有します。この共有オブジェクトは `qualifyMortgage()` から返されますが、この共有オブジェクトのプロパティーを設定することで、スクリプトは Java アプリケーションと通信を行い、スクリプトの実行結果を Java アプリケーションに返すことができます。またスクリプトは、グローバル変数 result を使ってこの Java オブジェクトにアクセスします。各スクリプトは、この共有スクリプト・オブジェクトを使って Java プログラムと通信を行い、スクリプトの実行結果を Java プログラムに返します。

`scriptEngine.put()` の最後の呼び出しによって、内部ヘルパー・クラス (`ScriptEarlyExit`、これもリスト 1 に示してあります) のインスタンスを `scriptExit` という変数名でスクリプトが利用できるようになります。`ScriptEarlyExit` は、`withMessage()` と `noMessage()` という 2 つの簡単なメソッドを定義します。これらのメソッドの役割は、例外をスローすることだけです。もしスクリプトが `scriptExit.withMessage()` あるいは `scriptExit.noMessage()` を呼び出すと、このメソッドは `ScriptEarlyExitException` をスローします。スクリプト・エンジンはこの例外をキャッチして実質的にスクリプトの処理を停止し、そしてこのスクリプトを呼び出した `eval()` メソッドに対して `ScriptException` をスローします。

スクリプトを途中で終了するという、この回りくどい方法によって、関数あるいはメソッドの外でのスクリプト処理から確実に戻ることができます。この目的のためのステートメントが、すべてのスクリプト言語に用意されているわけではありません。例えば JavaScript では、最上位レベ

ルのコードを実行している時には return 文を使うことができませんが、このサンプル・アプリケーションでの不動産担保ローン商品の処理スクリプトは最上位レベルのコードで構成されています。共有オブジェクト `scriptExit` はこの隙間を埋め、どのような言語のスクリプトであっても、借り手は不動産担保ローン商品に適格ではないとスクリプトが判断すると、そのスクリプトを即座に終了させます。

`qualifyMortgage` 内で実行されるスクリプト・エンジンの `eval` メソッドの呼び出し (太字で示してあります) では、`try/catch` ブロックを使って `ScriptException` をキャッチします。`catch` ブロックのコードは `ScriptException` のエラー・メッセージを検証することで、そのスクリプト例外が `ScriptEarlyExitException` によって起きたのか、あるいは本当のスクリプト・エラーなのかを判断します。もしエラー・メッセージが `ScriptEarlyExitException` という名前を含んでいる場合には、このコードはすべてが計画どおり行われたと見なし、スクリプト例外を無視します。

リテラル・ストリングを探すために Java スクリプト API のスクリプト例外エラー・メッセージを調べるという、この方法はその場しのぎの方法ですが、この記事の例で使用している Groovy と JavaScript、そして Ruby 言語のインタープリターには有効なのです。望ましい姿としては、呼び出された Java コードから例外スタックにスローされ、`Throwable.getCause()` メソッドを使って取得できる Java 例外を、すべてのスクリプト言語の実装に追加することです。JRuby や Groovy などのインタープリターはそうなっていますが、組み込みの JavaScript インタープリターである Rhino はそうなっていません。

## コードを実行する: ScriptMortgage QualifierRunner

`ScriptMortgageQualifier` クラスをテストするためにテスト・データを使います。このテスト・データは、借り手のサンプル 4 人と、これらの借り手が購入しようとする 1 つのサンプル物件、そしてサンプルの不動産担保ローンを表現します。借り手のサンプルと物件とローンを、3 つのスクリプトそれぞれに対して実行し、その借り手が、そのスクリプトで表現される不動産担保ローン商品のビジネス・ルールを満足するかどうかを調べます。

リスト 2 は `ScriptMortgageQualifierRunner` プログラムの一部を示しています。このプログラムを使ってこれらのテスト・オブジェクトを作成し、ディレクトリーの中にあるスクリプト・ファイルを発見し、そしてそれらをリスト 1 の `ScriptMortgageQualifier` クラスを使って実行します。このプログラムのヘルパー・メソッドのうち、createGoodBorrower() と `createAverageBorrower()`、`createInvestorBorrower()`、`createRiskyBorrower()`、`createProperty()`、そして `createLoan()` は、スペースを節約するために省略してあります。これらのヘルパー・メソッドは、単にエンティティー・オブジェクトを作成し、テスト用に適当な値を設定するだけです。すべてのメソッドを含んだ完全なソース・コードは「ダウンロード」のセクションで入手することができます。

### リスト 2. ScriptMortgageQualifierRunner プログラム

```
// Imports and some helper methods not shown.
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();
```

```
    /**
     * Main method to create a File for the directory name on the command line,
     * then call the run method if that directory exists.
     */
    public static void main(String[] args) {
        if (args.length > 0 && args[0].contains("-help")) {
            printUsageAndExit();
        }
        String dirName;
        if (args.length == 0) {
            dirName = "."; // Current directory.
        } else {
            dirName = args[0];
        }

        scriptDirectory = new File(dirName);
        if (!scriptDirectory.exists() || !scriptDirectory.isDirectory()) {
            printUsageAndExit();
        }

        run();
    }

    /**
     * Determines mortgage loan-qualification status for four test borrowers by
     * processing all script files in the given directory. Each script will determine
     * whether the given borrower is qualified for a particular mortgage type
     */
    public static void run() {
        ScriptMortgageQualifier mortgageQualifier = new ScriptMortgageQualifier();

        for(;;) { // Requires Ctrl-C to exit
            runQualifications(mortgageQualifier, goodBorrower, loan, property);
            runQualifications(mortgageQualifier, averageBorrower, loan, property);

            loan.setDownPayment(30000.0); // Reduce down payment to 10%
            runQualifications(mortgageQualifier, investorBorrower, loan, property);

            loan.setDownPayment(10000.0); // Reduce down payment to 3 1/3%
            runQualifications(mortgageQualifier, riskyBorrower, loan, property);

            waitOneMinute();
        }
    }

    /**
     * Reads all script files in the scriptDirectory and runs them with this borrower's
     * information to see if he/she qualifies for each mortgage product.
     */
    private static void runQualifications(
        ScriptMortgageQualifier mortgageQualifier,
        Borrower borrower,
        Loan loan,
        Property property
    ) {
        for (File scriptFile : getScriptFiles(scriptDirectory)) {
            // Print info about the borrower, loan and property.
            System.out.println("Processing file: " + scriptFile.getName());
            System.out.println("  Borrower: " + borrower.getName());
            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 {
                // Run the script rules for this borrower on the loan product.
```

```
                result = mortgageQualifier.qualifyMortgage(
                    borrower, property, loan, scriptFile
                );
            } 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; // Must have hit exception.

            // Print results.
            System.out.println(
                "* Mortgage product: " + result.getProductName() +
                ", Qualified? " + result.isQualified() +
                "\n* Interest rate: " + result.getInterestRate() +
                "\n* Message: " + result.getMessage()
            );
            System.out.println();
        }
    }

    /** Returns files with a '.' other than as the first or last character. */
    private static File[] getScriptFiles(File directory) {
        return directory.listFiles(new FilenameFilter() {
            public boolean accept(File dir, String name) {
                int indexOfDot = name.indexOf('.');
                // Ignore files w/o a dot, or with dot as first or last char.
                if (indexOfDot < 1 || indexOfDot == (name.length() - 1)) {
                    return false;
                } else {
                    return true;
                }
            }
        });
    }

    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);
        }
    }
}
```

`ScriptMortgageQualifierRunner` の `main()` メソッドは、スクリプト・ファイルを読み取るためのディレクトリーをコマンドライン上で見つけ、もしそのディレクトリーが存在する場合には、そのディレクトリーの `File` オブジェクトを使って静的変数を設定し、そして `run()` メソッドをコールしてさらに処理を行います。

`run()` メソッドはリスト 1の `ScriptMortgageQualifier` クラスをインスタンス化し、そして無限ループを使って、(借り手とローンに関する 4 つのシナリオで) 内部の `runQualifications()` メソッドを呼び出します。この無限ループは、不動産担保ローン商品の申し込みを継続的にライブ処理する様子をシミュレートしています。このループでは処理対象のディレクトリーにあるスクリプト・ファイル (不動産担保ローン商品) に追加や変更を行うことができ、またこれらの変更はアプリケーションを停止することなく、動的に有効になります。アプリケーションのビジネス・ロジックが外部スクリプトにあることによって、実行時にビジネス・ロジックを変更するという、こうした動的な機能を実現できるのです。

`runQualifications()` ヘルパー・メソッドは実際に `ScriptMortgageQualifer.qualifyMortgage` を呼び出し、スクリプト・ディレクトリーの中で見つかるスクリプト・ファイルそれぞれに対して 1 回の呼び出しを行います。各呼び出しの前に、スクリプト・ファイルと処理対象の借り手を記述する出力命令があり、また各呼び出しの後にはさらに、各不動産担保ローン商品に対して借り手が適格かどうかを示す出力命令があります。適格かどうかの結果は共有 Java オブジェクト `MortgageQualificationResult` の属性で判断され、スクリプト・コードはこれを使って結果を返します。

この記事のソース・コードの ZIP ファイルには、Groovy と JavaScript、そして Ruby で記述された 3 つのサンプル・スクリプトのファイルが含まれています。各ファイルは、標準的な 30 年固定金利の不動産担保ローン商品の異なるタイプを表しています。スクリプトのコードは借り手がその不動産担保ローン商品のタイプに適格かどうかを判断し、先ほど説明したスクリプト・エンジンの `put()` メソッドで提供される共有のグローバル変数 `result` に対するメソッドを呼び出すことで判断の結果を返します。グローバル変数 result は `MortgageQualificationResult` クラスのインスタンスであり、その一部をリスト 3 に示します。

### リスト 3. MortgageQualificationResult の結果クラス

```
public class MortgageQualificationResult {
    private boolean qualified;
    private double interestRate;
    private String message;
    private String productName;

    // .. Standard setters and getters not shown.
}
```

このスクリプトは、リスト 3 に示す `result` プロパティーを設定し、借り手がその不動産担保ローン商品に適格かどうか、そしてどの利率で適格かを返します。スクリプトは借り手が不動産担保ローン商品に適格ではない理由を `message` プロパティーと `productName` プロパティーによって設定でき、またこれらのプロパティーを使って関連する商品名を返すことができます。

## スクリプト・ファイル

`ScriptMortgageQualifierRunner` を実行した結果の出力を示す前に、このプログラムが実行する Groovy と JavaScript、そして Ruby のスクリプト・ファイルを見てみましょう。Groovy スクリプトによるビジネス・ロジックで定義される不動産担保ローン商品は、比較的容易に適格とされますが、金銭リスクが高いことを考慮して金利は高くなっています。JavaScript のスクリプトは政府支援による不動産担保ローンを表し、借り手は収入の上限その他の制約を満足する必要があります。Ruby スクリプトによる不動産担保ローン商品は、クレジット・ヒストリーが良好な、そして

高額の頭金を支払える人のみに借り手を制限するビジネス・ルールを含んでいます。この商品には低い金利という見返りがあります。

リスト 4 は Groovy のスクリプトを示していますが、たとえ Groovy を知らなくても、おそらく内容を読み取れるはずです。

## リスト 4. Groovy による不動産担保ローン商品のスクリプト

```
/*
   This Groovy script defines the "Groovy Mortgage" product.
   This product is relaxed in its requirements of borrowers.
   There is a higher interest rate to make up for the looser standard.
   All borrowers will be approved if their credit history is good, they can
   make a down payment of at least 5%, and they either earn more than
   $2,000/month or have a net worth (assets minus liabilities) of $25,000.
*/

// Our product name.
result.productName = 'Groovy Mortgage'

//  Check for the minimum income and net worth
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.'
}

// Everyone else qualifies. Find interest rate based on down payment percent.
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
}
```

グローバル変数、`result` と `borrower`、`loan`、そして `property` に注意してください。スクリプトはこれらを使って共有 Java オブジェクトにアクセスし、値を設定します。`ScriptEngine.put()` メソッドを呼び出すことで設定されるのは、これらの変数の名前です。

また、`result.productName = 'Groovy Mortgage'` のような Groovy のステートメントにも注意してください。これは `MortgageQualificationResult` オブジェクトのストリング・プロパティー `productName` を直接設定しているように見えますが、リスト 3 を見ると明らかにプライベート・インスタンス変数です。これは、Javaスクリプト API がカプセル化違反をそっと許しているのではありません。実は Javaスクリプト API で使用できる Groovy や他の大部分のスクリプト言語のインタープリターは、共有 Java オブジェクトと適切に協調動作するのです。Groovy のステートメント

が Java オブジェクトの private のプロパティー値を設定あるいは読み取ろうとする場合には、もし JavaBean 風で public の `setter` あるいは `getter` メソッドが定義されていれば、Groovy はそれらを探して使用します。例えば、`result.productName = 'Groovy Mortgage'` というステートメントは、想定される Java 命令 `result.setProductName("Groovy Mortgage")` に自動的に変換されます。これは Java の `setter` 命令と等価であり Groovy でも有効です。これでもスクリプトの中で適切に動作するはずですが、プロパティーの割り当て命令を直接使う方が、より Groovy らしくなります。

今度は JavaScript の不動産担保ローン商品を見てみましょう (リスト 5)。JavaScript で表現される不動産担保ローン商品は、持ち家促進のための政府支援ローンを表現しようとしています。従ってこのビジネス・ルールでは、これが借り手にとって初めての家の購入であること、そして借り手がその家を貸して収入を得るのではなく、その家に住むつもりであることが要求されます。

## リスト 5. JavaScript による不動産担保ローン商品のスクリプト

```
/**
 * This script defines the "JavaScript FirstTime Mortgage" product.
 * It is a government-sponsored mortgage intended for low-income, first-time
 * home buyers without a lot of assets who intend to live in the home.
 * Bankruptcies and bad (but not terrible!) credit are OK.
 */
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()
}

// Qualifies. Determine interest rate based on loan amount and credit score.
result.qualified = true
result.message = 'Congratulations, you qualify.'

if (loan.loanAmount > 450000) {
    result.interestRate = 0.08 // Big loans and poor credit require higher rate.
} 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 { // Good credit gets best rate.
    result.interestRate = 0.06
}
```

Groovy スクリプトでは、不適格を示すメッセージの設定とスクリプトの終了を 1 つのステートメントで行える Java の `scriptExit.withMessage()` メソッドを使いましたが、この JavaScript コード

では使えないことに注意してください。これは JavaScript インタープリターの Rhino が、スローされた Java 例外を、作成される `ScriptException` スタック・トレースの中に埋め込まれる「原因」として上げないためです。従って、Java コードからスローされるスクリプト例外メッセージをスタック・トレースの中で見つけるのは困難です。そのためリスト 5 の JavaScript コードは、結果メッセージの理由を別に設定してから `scriptExit.noMessage()` をコールし、スクリプト処理終了の原因となった例外を呼び出す必要があります。

3 番目で最後の不動産担保ローン商品 (リスト 6) は、Ruby で記述されています。このスクリプトは、クレジット・ヒストリーが良好で 20% の頭金を支払える借り手を対象にしています。

## リスト 6. Ruby によるモーゲージのスクリプト

```
# This Ruby script defines the "Ruby Mortgage" product.
# It is intended for premium borrowers with its low interest rate
# and 20% down payment requirement.

# Our product name
$result.product_name = 'Ruby Mortgage'

# Borrowers with credit unworthiness do not qualify.
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

# Check other negatives
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

# Borrower qualifies. Determine interest rate of loan
$result.message = "Qualified!"
$result.qualified = true

# Give the best interest rate to the best credit risks.
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
```

### JRuby 1.0 では $ を忘れないこと

Ruby スクリプトから共有 Java オブジェクトにアクセスする場合には、Ruby のグローバル変数の構文を覚えておくことが重要です。JRuby 1.0 と現在の JRuby 1.0.1 バイナリー・リリースでは、もしグローバル変数に $ を付け忘れると、何が誤っているのかの情報を何も付けずに RaiseException がスローされます。このバグは JRuby のソース・コード・リポジトリーでは修正されているので、今後のバイナリー・リリースでは修正されるはずです。

リスト 6 を読むとわかるように、スクリプト・エンジンのスコープに置かれた共有 Java オブジェクトに Ruby からアクセスするには、そのオブジェクトの名前の前に `$` を付ける必要があります。これは Ruby のグローバル変数の構文です。スクリプト・エンジンは Java 変数をグローバル変数としてスクリプト言語と共有するため、Ruby のグローバル変数の構文を使う必要があります。

またこのリストの中で、 JRuby が共有 Java オブジェクトを呼び出す際に、Ruby 方式を Java 方式に自動的に変換していることにも注意してください。例えば、アンダーバーで単語を区切る Ruby の規則 (例えば `$result.product_name = 'Ruby Mortgage'` など) を使って Java オブジェクトのメソッドが呼び出されているのを JRuby が見つけると、JRuby は、アンダーバーを持つ名前が見つからない時には、代わりに大文字と小文字混合のメソッド名を探します。従って、`product_name=` という Ruby 方式のメソッド名は、Java の `result.setProductName("Ruby Mortgage")` コールに適切に変換されます。

## プログラムの出力

今度は、3 つの不動産担保ローン商品のスクリプト・ファイルを使ってプログラム `ScriptMortgageQualifierRunner` を実行した場合の出力を調べてみましょう。このプログラムを実行するには、この記事からダウンロードできるソース・コードに含まれている Ant スクリプトを使います。Maven がお好みの方のために、ダウンロード用 ZIP ファイルの README.txt ファイルの中に、このプロジェクトを Maven を使ってビルドし、実行するための説明が含まれています。この Ant コマンドは `ant run` です。`run` タスクによって、スクリプト・エンジンと言語の JAR ファイルが確実にクラスパスに置かれます。リスト 7 は Ant の出力を示しています。

### リスト 7. Ant のプログラム出力

```
> 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...
```

この出力には 12 のセクションがあります。これは、このプログラムが 4 人の借り手サンプルのローンのプロファイルを 3 つのスクリプトそれぞれに送信しており、それによって借り手が 3 つの不動産担保ローン商品のどれか (あるいはすべて) に適格かどうかを調べるためです。この記事の場合デモを目的としているため、このプログラムは次に 1 分間待って不動産担保ローン商品のスクリプトの処理を繰り返します。この休止中に、任意のスクリプト・ファイルを編集してビジネス・ルールを変更したり、あるいは新しいスクリプト・ファイルをスクリプト・ディレクトリーに追加して独自の不動産担保ローン商品を表現したりすることができます。プログラムはパスごとにスクリプト・ディレクトリーをスキャンし、そして見つかった新しいスクリプト・ファイルを処理します。

例えば、ローンに適格となるために必要なクレジット・スコア (訳注: 消費者個人に与えられる信用評価点のこと。米国で個人の信用度を評価する際に近年特に用いられている。) の最低値を上げたいとします。1 分間の休止中に src/main/scripts/mortgage-products ディレクトリーの JavaScriptFirstTimeMortgage.js スクリプト (リスト 5 を参照) を編集し、23 行目のビジネス・ルールを `if (borrower.creditScore < 500) {` から `if (borrower.creditScore < 550) {` に変更することができます。次回このルールを実行すると、Risk E. Borrower がもはや JavaScript FirstTime Mortgage に適格ではなくなっていることがわかります。この借り手のクレジット・スコア (520) は、今や低すぎるのです。エラー・メッセージは相変わらず「Your credit score of 520 does not meet the 500 requirement (あなたのクレジット・スコアの 520 は 500 という条件を満足しません)」と表示されますが、この、今や古くなってしまったエラー・メッセージも即座に変更することができます。

## 動的スクリプトによる危険を回避する

実行時にプログラムを変更できるという機能は強力ですが、危険も潜んでいます。新しい機能と新しいビジネス・ルールは、アプリケーションを停止して再起動しなくても、アプリケーションの実行中に追加することができるのです。同様に、新しいバグも、そして致命的となり得るバグも、容易に追加できてしまいます。

しかし実行中のアプリケーションを動的に変更することは、停止したアプリケーションを変更する以上に危険なわけではありません。静的な方法は単に、アプリケーションを再起動しないと新しいエラーを見つけられない、という意味でしかありません。適切なソフトウェア開発の習慣としては、実動のアプリケーションを変更する場合には、それが動的な変更であれ静的な変更であれ、どのような変更であっても、実動させる前にそれをテストする必要があります。Java スクリプト API でもそのルールが変わるわけではありません。

外部スクリプト・ファイルは、開発中に通常のユニット・テストの一部としてテストすることができます。JUnit あるいは別のテスト・ハーネスを使い、またスクリプトが実行時に必要とする Java オブジェクトの模擬オブジェクトを用意し、その模擬オブジェクトを使ってスクリプトを実行します。そしてそのスクリプトがエラーを起こさずに実行され、想定される結果が作成されることを確認します。アプリケーションのロジックを Java ではないスクリプト・ファイルに外部化したからといって、これらのスクリプトをテストしなくてよいことにはなりません。

Web CGI のスクリプトをプログラミングしたことのある人、あるいは現在プログラミングしている人であれば、`ScriptEngine` の `eval()` メソッドに渡すものに注意する必要がある、という点は驚くべきことではありません。スクリプト・エンジンは、`eval` メソッドに渡されたコードを即座に実行します。従って、信頼できないソースからのテキストを含むストリングあるいは `Reader` オブジェクトは、決してスクリプト・エンジンに渡して実行させるべきではありません。

例えば、スクリプト API を使って Web アプリケーションをリモートでモニターすることができます。そのために、Web アプリケーションのステータス情報を提供する重要な Java オブジェクトにスクリプト・エンジンがアクセスできるようにしたとします。そして単純な Web ページを作成し、そのページで任意のスクリプト表現を受け付け、それをスクリプト・エンジンに提供して評価させ、出力を Web ページに戻すとします。こうすれば、実行中の Java オブジェクトのメソッドを照会して実行し、アプリケーションの状態とヘルスを判断することができます。

そうしたシナリオでは、その Web ページにアクセスできる人であれば、誰でも、そのスクリプト言語で利用できる任意の命令を実行して任意の共有 Java オブジェクトにアクセスすることができます。不注意なプログラミングや構成の誤り、あるいはセキュリティー違反がある場合には、もし `System.exit` あるいは `/bin/rm -fr /` に等価なスクリプト命令をハッカーが実行すると、権限のないユーザーに機密情報が漏れたり、あるいはアプリケーションを DoS (サービス不能) 攻撃にさらすことになったりします。Java スクリプト API も、他の強力なツールと同じように、適切な注意を払って使う必要があります。

## さらなる機能

この記事は、Java アプリケーションが実行時に外部スクリプトを動的に読み取って実行できること、また外部スクリプトが、そのスクリプトに対して明示的に提供されている Java オブジェクトにアクセスできることを説明しました。Java スクリプト API には他にも機能があります。例えば、次のようなことができます。

- スクリプト言語を使って Java インターフェースを実装し、他の Java インターフェース参照を呼び出すのと同じように Java コードからスクリプト・コードを呼び出す。
- 内部スクリプトから Java オブジェクトをインスタンス化して使用し、これらのオブジェクトを後で Java アプリケーションで使用できるようにしておく。
- 動的スクリプトがロードされたらプリコンパイルし、その後の実行を速くする。
- スクリプトが使用する入力と出力のストリームを設定することで、ファイルをスクリプトのコンソール入力ソースとして使いやすくし、またスクリプトのコンソール出力をファイルあるいは他のストリームにキャプチャーしやすくする。
- スクリプトがコマンドラインの引数として使用できる位置パラメーターを設定する。

Java スクリプト API は、これらの機能のいくつかを、スクリプト・エンジンの実装者のためのオプションとして定義します。そのため、すべてのスクリプト・エンジンがこれらの機能を提供し

ているわけではありません。これらの機能やその他の機能についての詳しい資料やオンライン資料は「参考文献」を参照してください。

---

# ダウンロード

| 内容 | ファイル名 | サイズ |
| --- | --- | --- |
| Source code and all JAR files | java-scripting-part2.zip | 4.5MB |

# 著者について

Tom McQueeney

Tom McQueeney は全米で活動するコンサルタント会社である Idea Integration の Java 開発者であり、アプリケーション・アーキテクトです。彼は開発を迅速に、効率的に、そして楽しくするために Ruby や Groovy などの動的言語を Java プロジェクトに統合する業務を楽しみながら行っています。彼は過去に O'Reilly 社主催の OSCON と ApacheCon Europe で講演したことがあり、また Denver Java Users Group の会長を務めていました。彼は、同じく認定 Java アーキテクトである妻と、ワシントン特別区に住んでいます。