# 動的言語を動的に呼び出す: 第 1 回 Java スクリプト API 入門

## 実行中のアプリケーションを javax.script API を使って変更する

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

2007年 9月 04日

動的言語を Java アプリケーションで使用するために Java™ バイトコードにコンパイルする必要はありません。Java コードからは、非常にさまざまな種類のスクリプト言語を、単純で統一された方法で実行時に呼び出すことができます。そのためには、Java SE (Java Platform, Standard Edition 6) に追加され、Java SE 5 と後方互換性がある、スクリプト・パッケージを使います。2 回シリーズの第 1 回である今回は、Java スクリプト API の機能について紹介します。この記事では単純な Hello World アプリケーションを使いながら、Java コードがスクリプト・コードをどのように実行するか、また逆にスクリプトはどのように Java コードを実行するかについて説明します。第 2 回では Java スクリプト API の実力を掘り下げて解説します。

このシリーズの他の記事を見る

Java 開発者は、Java 言語が必ずしもすべての作業に最適ではないことを知っています。今年リリースされた JRuby と Groovy の 1.0 バージョンによって、Java アプリケーションに動的言語を追加することへの関心が高まりました。Groovy や JRuby、Rhino、Jython などのオープン・ソース・プロジェクトによって、いわゆるスクリプト言語でコードを作成することができ、そしてそれを JVM の中で実行することができます (「参考文献」を参照)。しかしこれらの言語を Java コードと統合するということは、通常は各インタープリター固有の API と機能を学ぶ必要がある、ということを意味していました。

Java SE 6 に追加された `javax.script` パッケージによって、動的言語の統合が容易になりました。このパッケージを利用すると、ごくわずかのインターフェースや具象クラスのセットを使って、1 つの単純な方法で非常にさまざまな種類のスクリプト言語を呼び出すことができます。しかし Java スクリプト API は、アプリケーションの一部をスクリプト化しやすくするだけではありません。つまりスクリプト・パッケージによって、実行時に外部スクリプトを読み取り、そして呼び出すことができます。これは、これらのスクリプトを動的に変更できること、それによって実行中のアプリケーションの振る舞いを変更できることを意味します。

この記事は 2 回シリーズの第 1 回として、Hello World スタイルのアプリケーションを使いながら Java スクリプト API の機能と重要なクラスを紹介します。第 2 回では、このスクリプト API の実力をさらに顕著に示す現実的なサンプル・アプリケーションについて解説します。第 2 回のアプリケーションでは、スクリプト API を使って動的なルール・エンジンを作成します。このルール・エンジンでのルールは Groovy と JavaScript、そして Ruby で作成された外部スクリプトとしてエンコードされています。これらのルールによって、住宅ローンの申込者に対して、その人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。Java スクリプト API を使ってルールを外部化することで、ルールの変更や新しい不動産担保ローン商品の追加を実行時に行えるようになります。

## Java スクリプト API

### スクリプト言語と動的言語

スクリプト言語は通常、コンパイル・ステップを別に設けずにインタープリター・シェルから実行する言語のことを指しています。動的言語は通常、ランタイムが変数の型やオブジェクトの振る舞いを判断するまで待ち、またクロージャーや継続 (continuation) などの機能を含む言語を指します。いくつかの汎用プログラミング言語は、どちらにも該当します。この記事では Java スクリプト API に焦点を当てているため、スクリプト言語を採用しています。これは、ここで取り上げる言語に動的な機能が欠けているためではありません。

スクリプト・パッケージは、Java アプリケーションにスクリプト言語を統合するための統一的な方法を提供するために、2006年12月に Java 言語に追加されました。このパッケージによって、言語開発者は、自分たちが作成した言語を Java アプリケーションから動的に呼び出すために必要なグルー・コードを作成することができます。Java 開発者の立場から見ると、このスクリプト・パッケージはいくつかのクラスとインターフェースを提供しており、それらを利用すると、さまざまな言語で作成されたスクリプトを共通の API を使って呼び出すことができます。従ってこのスクリプト・パッケージは、さまざまな言語 (さまざまなデータベースなど) を一貫したインターフェースを使って Java プラットフォームに統合できるという意味で、JDBC (Java Database Connectivity) パッケージと似ています。

これまでは、Java コードから動的にスクリプト言語を呼び出そうとすると、各言語のディストリビューションで提供される固有のクラスを使うか、あるいは Apache の Jakarta BSF (Bean Scripting Framework) を使う必要がありました。BSF では、1 つの API の背後にいくつかのスクリプト言語が統合されています (「参考文献」を参照)。大部分を BSF をベースにしている Java SE 6 スクリプト API を利用すると、AppleScript や Groovy、JavaScript、Jelly、PHP、Python、Ruby、そして Velocity など 20 数種類を超えるスクリプト言語を Java コードに統合することができます。

このスクリプト API によって、Java アプリケーションと外部スクリプトが、お互いに相手を見られるようになります。Java コードが外部スクリプトを呼び出せるだけではなく、外部スクリプトからも、選択された Java オブジェクトにアクセスできるようになります。例えば、外部の Ruby スクリプトが Java オブジェクトのメソッドを呼び出し、それらのプロパティーにアクセスすることができます。つまりこれらのスクリプトによって、開発時には想定されていなかった振る舞いを実行中のアプリケーションに追加することができます。

外部スクリプトを呼び出す機能は、実行時におけるアプリケーションの機能強化や、構成、監視、その他のランタイム操作 (アプリケーションを停止させずにビジネス・ルールを変更するな

ど) に使用することができます。スクリプト・パッケージの使い方としては、次のようなものが考えられます。

- 本格的なルール・エンジンを使わずに、Java 言語よりも単純な言語でビジネス・ルールを作成する。
- ユーザーがアプリケーションを即座にカスタマイズできるプラグイン・アーキテクチャーを作成する。
- 既存のスクリプト (例えばテキスト・ファイルを処理する、あるいは変換するスクリプトなど) を Java アプリケーションに統合する。
- アプリケーションの実行時の振る舞いを、プロパティー・ファイルではなく本格的なプログラミング言語を使って外部で構成する。
- Java アプリケーションにドメイン固有言語を追加する。
- Java アプリケーションのプロトタイピングの際にスクリプト言語を使う。
- アプリケーションのテスト・コードをスクリプト言語で作成する。

## Hello World をスクリプト言語で作成する

`HelloScriptingWorld` クラス (この記事の他のコードと共にダウンロードすることができます (「参考文献」を参照)) は、この Java スクリプト・パッケージの重要な機能を表しています。このクラスは、ハードコーディングされた JavaScript のスニペットをスクリプト言語の例として使っています。このクラスの `main()` メソッド (リスト 1) は JavaScript スクリプト・エンジンを作成し、そしてこのスクリプト・パッケージの機能を際立たせている 5 つのメソッドを呼び出します (これらのメソッドのリストは後ほど示します)。

### HelloScriptingWorld の main メソッド

```
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);
}
```

`main()` メソッドの主な機能は、`javax.script.ScriptEngine` のインスタンスを取得することです (リスト 1の最初の 2 つのステートメント)。スクリプト・エンジンは、ある特定の言語のスクリプトをロードして実行します。スクリプト・エンジンは、Java スクリプト・

パッケージの中で最も頻繁に使われ、そして最も重要なクラスです。スクリプト・エンジンは、`javax.script.ScriptEngineManager` (最初のステートメント) から取得することができます。典型的なプログラムでは、多くのスクリプト言語が使われている場合を除き、スクリプト・エンジンの 1 つのインスタンスを取得すればよいだけです。

## ScriptEngineManager クラス

`ScriptEngineManager` は、スクリプト・パッケージの中で頻繁に使われる、おそらく唯一の具象クラスです。それ以外に使用されるものの大部分はインターフェースです。またスクリプト・パッケージの中で、直接インスタンス化される (あるいは Spring Framework などの依存性注入機構によって間接的にインスタンス化される) 唯一のクラスかもしれません。`ScriptEngineManager` がスクリプト・エンジンを返す方法は、次の 3 つのうちのいずれかです。

- エンジン名、あるいは言語名によって (例えばリスト 1 では JavaScript エンジンを要求しています)。
- その言語でスクリプト用に一般的に使われているファイル拡張子によって (例えば Ruby スクリプトの場合は .rb、など)。
- スクリプト・エンジンが処理方法を知っているものとして宣言した MIME タイプによって。

### この例ではなぜ JavaScript を使っているのか

この Hello World の例では JavaScript を使っています。これはコードが理解しやすいことも 1 つの理由ですが、Sun Microsystems と BEA Systems が提供する Java 6 のランタイム環境にオープン・ソースの JavaScript 実装である Mozilla Rhino をベースとする JavaScript インタープリターがバンドルされていることが大きな理由です。JavaScript では、クラスパスにスクリプト言語の JAR ファイルを追加する必要がありません。

`ScriptEngineManagers` はスクリプト・エンジンを間接的に発見し、また作成します。つまり、スクリプト・エンジン・マネージャーがインスタンス化されていると、これらのマネージャーは Java 6 で追加されたサービス発見の機構を使って、クラスパスに登録されたすべての `javax.script.ScriptEngineFactory` 実装を発見します。これらのファクトリー・クラスには Java スクリプト API の実装がパッケージされているため、これらのファクトリー・クラスを直接扱う必要はありません。

`ScriptEngineManager` はスクリプト・エンジンのファクトリー・クラスをすべて発見すると、それぞれを照会し、それが要求された型のスクリプト・エンジン (リスト 1 の場合は JavaScript エンジン) を作成できるかどうかを判断します。あるファクトリーが、要求された言語用のスクリプト・エンジンを作成できると答えると、スクリプト・エンジン・マネージャーはそのファクトリーにスクリプト・エンジンを作成するように依頼し、そしてそのファクトリーは作成したスクリプト・エンジンを呼び出し側に返します。もし要求された言語用のファクトリーが発見できない場合には、マネージャーは null を返します。リスト 1のコードは、これに備えて null の戻り値をチェックしています。

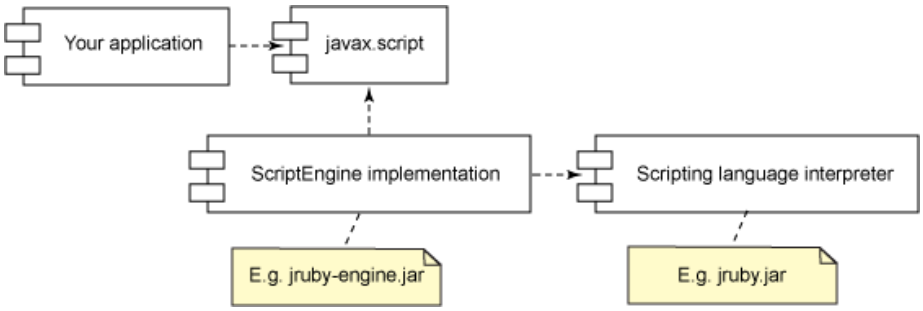
## ScriptEngine インターフェース

先ほど触れたように、このコードは `ScriptEngine` のインスタンスを使ってスクリプトを実行します。スクリプト・エンジンは、スクリプト・コードと、(最終的にコードを実行する) ベースとなる言語インタープリターまたはコンパイラーとの間のメディエーターとして動作します。そのため、各インタープリターがどのクラスを使ってコードを実行するのかを知る必要がありません。

例えば、JRuby 用のスクリプト・エンジンは、最初に JRuby の `org.jruby.Ruby` クラスのインスタンスにコードを渡してスクリプトを中間形式にコンパイルし、そしてその中間形式を再度呼び出してスクリプトを評価し、戻り値を処理することができます。スクリプト・エンジンの実装によって、インタープリターがクラス定義やアプリケーションのオブジェクト、入出力ストリームなどを Java コードと共有する方法の詳細が隠されるのです。

図 1 は、アプリケーションと Java スクリプト API、`ScriptEngine` の実装、そしてスクリプト言語のインタープリターとの間の一般的な関係を示しています。これを見ると、アプリケーションが、`ScriptEngineManager` クラスと `ScriptEngine` インターフェースを提供するスクリプト API のみに依存していることがわかります。`ScriptEngine` の実装コンポーネントは、特定のスクリプト言語用のインタープリターを使うための詳細を処理します。

## 図 1: スクリプト API のコンポーネントの関係




スクリプト・エンジンの実装と言語のインタープリターに必要な JAR ファイルをどこで取得するのか、と思う人がいるかもしれません。スクリプト・エンジンの実装を探すための最初の場所としては、java.net がホストするオープン・ソースの Scripting プロジェクト (「参考文献」を参照) が最適です。このプロジェクトでは、多くの言語のためのスクリプト・エンジンの実装と、他の場所でホストされているスクリプト・エンジンの実装へのリンクを見つけることができます。また Scripting プロジェクトは、このプロジェクトがサポートするスクリプト言語用のインタープリターをダウンロードするためのリンクも提供しています。

リスト 1 では、`main()` メソッドは、各メソッドの `JavaScript` コードを評価するために使用する `ScriptEngine` を各メソッドに渡します。この最初のメソッドをリスト 2 に示します。`invokeHelloScript()` メソッドはスクリプト・エンジンの `eval` メソッドを呼び出し、指定された `JavaScript` コードのストリングを評価して実行します。`ScriptEngine` インターフェースは、オーバーロードされた 6 つの `eval()` メソッドを定義します。これらのメソッドは、ストリング、あるいは `java.io.Reader` オブジェクトとして評価されるスクリプトを受け付けます (`java.io.Reader` はファイルなどの外部ソースからスクリプトを読み取るために一般的に使われます)。

## invokeHelloScript メソッド

```
private static void invokeHelloScript(ScriptEngine jsEngine) throws ScriptException {
    jsEngine.eval("println('Hello from JavaScript')");
}
```

### スクリプトの実行コンテキスト

`HelloScriptingWorld` アプリケーションのサンプル・スクリプトでは JavaScript の `println()` 関数を使ってコンソールに出力しますが、入力と出力のストリームは完全にコ

ントロールすることができます。スクリプト・エンジンは、スクリプト実行のコンテキストを変更するオプションを提供しています。これはつまり、標準入力や標準出力、標準エラーに使われるストリームを変更できること、また実行されるスクリプトがどのグローバル変数と Java オブジェクトを利用できるかを定義できるということを意味しています。

`invokeHelloScript()` メソッドの JavaScript は、`Hello from JavaScript` を標準出力ストリーム (この場合はコンソール・ウィンドウ) に出力します。(リスト 6 には `HelloScriptingWorldApplication` を実行したことによる完全な出力が含まれています。)

このクラスの、このメソッドや他のメソッドが、`javax.script.ScriptException` をスローすることを宣言していることに注目してください。この (スクリプト・パッケージで定義されている唯一の) チェック例外は、エンジンが、指定されたコードの構文解析あるいは実行に失敗したことを示します。スクリプト・エンジンのすべての `eval()` メソッドは ScriptException をスローすることを宣言しているため、これをコードで適切に処理できる必要があります。

リスト 3 は、これに関連する 2 つのメソッド、`defineScriptFunction()` と `invokeScriptFunctionFromEngine()` を示しています。`defineScriptFunction()` メソッドも、ハードコーディングされた JavaScript のスニペットを使ってスクリプト・エンジンの `eval()` メソッドを呼び出します。ただし、このメソッドは JavaScript の関数 `sayHello()` を定義すること以外は何もしないことに注意してください。コードは実行されていません。`sayHello()` 関数は 1 つのパラメーターをとり、これを次の `println()` 文でコンソールに出力します。スクリプト・エンジンの JavaScript インタープリターは、この関数をインタープリターのグローバル環境に追加し、その後 `eval()` メソッドを呼び出してこの関数を利用できるようにします (このメソッドの呼び出しは、(ご想像通り) `invokeScriptFunctionFromEngine()` メソッドの中で実行されています)。

## defineScriptFunction メソッドと invokeScriptFunctionFromEngine メソッド

```
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!')");
}
```

この一対のメソッドは、スクリプト・エンジンがアプリケーション・コンポーネントの状態を維持すること、そしてその状態を、後でエンジンの `eval()` メソッドを呼び出す際に利用できることを示しています。`invokeScriptFunctionFromEngine()` メソッドは、この維持された状態を利用し、前に `eval()` を呼び出す際に定義されたJavaScript 関数 `sayHello()` を呼び出します。

多くのスクリプト・エンジンは、`eval()` を呼び出してから次に呼び出すまでの間、グローバル変数と関数の状態を維持します。しかし重要な注意点として、Java スクリプト API はスクリプト・エンジンがこの機能を提供することを要求しません。しかしこの記事で使用している JavaScript と Groovy、そして JRuby のスクリプト・エンジンは、`eval()` を呼び出してから次に呼び出すまでの間、状態を維持します。

リスト 4 は前の例のバリエーションを示しています。この `invokeScriptFunctionFromJava()` メソッドは、ScriptEngine の `eval()` メソッドあるいは JavaScript コードを使わずに JavaScript 関数 `sayHello()` を呼び出すという点が異なります。このメソッドは代わりに、Java スクリプト API の javax.script.Invocable インターフェースを使って、スクリプト・エンジンが維持管理する関数を呼び出します。`invokeScriptFunctionFromJava()` メソッドはスクリプト・エンジンのオブジェクトを Invocable インターフェースにキャストし、そしてそのインターフェースの `invokeFunction()` メソッドを呼び出し、特定のパラメーターを使って JavaScript 関数 `sayHello()` を呼び出します。呼び出された関数が値を返すと、`invokeFunction()` メソッドは、その値を Java Object 型としてラップして返します。

### invokeScriptFunctionFromJava メソッド

```
private static void invokeScriptFunctionFromJava(ScriptEngine engine)
  throws ScriptException, NoSuchMethodException
{
Invocable invocableEngine = (Invocable) engine;
    invocableEngine.invokeFunction("sayHello", "from Java");
}
```

#### プロキシーを使った高度なスクリプト呼び出し

スクリプト関数またはスクリプト・メソッドが Java インターフェースを実装している場合には、Invocable を高度な方法で使うことができます。Invocable インターフェースは `getInterface()` メソッドを定義します。このメソッドはインターフェースをパラメーターとして使い、提供されたインターフェースを実装する Java プロキシー・オブジェクトを返します。スクリプト・エンジンからプロキシー・オブジェクトを取得できると、それを通常の Java オブジェクトとして扱うことができます。プロキシーに対して呼び出されたメソッドはスクリプト・エンジンに委譲され、スクリプト言語によって実行されます。

リスト 4 には JavaScript が含まれていないことに注意してください。`Invocable` インターフェースを使うことによって、Java コードはスクリプトの実装言語を知らなくてもスクリプトの関数を呼び出すことができます。`invokeFunction()` メソッドは、指定された名前またはパラメーター型を持つ関数をスクリプト・エンジンが見つけられないと、`java.lang.NoSuchMethodException` をスローします。

Java スクリプト API はスクリプト・エンジンが `Invocable` インターフェースを実装することを要求しません。しかし現実的には、リスト 4のコードは、スクリプト・エンジンが `Invocable` インターフェースを実装していることを、キャストを行う前に `instanceof` 演算子を使って確認する必要があります。

## スクリプト・コードから Java メソッドを呼び出す

リスト 3 とリスト 4 の例は、スクリプト言語で定義された関数またはメソッドを Java コードが呼び出す方法を示しています。皆さんはおそらく、スクリプト言語で作成されたコードを使って Java オブジェクトのメソッドを呼び出せるのか疑問に思っているかもしれませんが、呼び出せるのです。リスト 5 の invokeJavaFromScriptFunction() メソッドは、スクリプト・エンジンが Java オブジェクトを利用できるようにする方法と、その Java オブジェクトのメソッドをスクリプト・コードで呼び出す方法を示しています。正確に言えば、invokeJavaFromScriptFunction() メソッドは、スクリプト・エンジンの put() メソッドを使って HelloScriptingWorld クラスのインスタンス自体をエンジンに提供します。put() メソッドを呼び出す際に提供される名前を使ってエンジンが

Java オブジェクトにアクセスできると、そのオブジェクトは eval() メソッドへの呼び出しの中のスクリプト・コードで使われます。

## invokeJavaFromScriptFunction メソッドと getHelloReply メソッド

```
private static void invokeJavaFromScriptFunction(ScriptEngine engine)
  throws ScriptException
{
    engine.put("helloScriptingWorld", new HelloScriptingWorld());
    engine.eval(
        "println('Invoking getHelloReply method from JavaScript...');" +
        "var msg = helloScriptingWorld.getHelloReply(vJavaScript');" +
        "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 + "'";
}
```

リスト 5 の中で、`eval()` メソッドへの呼び出しに含まれる `JavaScript` コードは、Java オブジェクト `HelloScriptingWorld` を使います。このオブジェクトへのアクセスは、スクリプト・エンジンの `put()` メソッドを呼び出す際に指定される変数名 `helloScriptingWorld` を使って行われます。`JavaScript` コードの 2 行目は、(同じくリスト 5 に示す) Java のパブリック・メソッド getHelloReply() を呼び出します。`getHelloReply()` メソッドは、ストリング `Java method getHelloReply says, 'Hello, <parameter>'` を返します。`eval()` メソッドの `JavaScript` コードは `msg` 変数に Java の戻り値を割り当て、そしてその値をコンソールに出力します。

### Java オブジェクトの変換

あるスクリプト・エンジンの環境で実行するスクリプトで Java オブジェクトを利用できるようにする場合、そのスクリプト・エンジンは、そのスクリプト言語に適用できるオブジェクト型でそのオブジェクトをラップする必要があります。このラッピングでは、オブジェクトと値との適切な変換をする必要があるかもしれません (例えば、スクリプト言語の数学表現の中で Java の `Integer` オブジェクトを直接使用できるようにするなど)。Java オブジェクトをどのようにスクリプト・オブジェクトに変換するかは各スクリプト言語エンジンによって異なり、この記事ではこれに関する説明を省略します。しかし変換が行われていることは知っておく必要があり、使用しているスクリプト言語が想定通り変換を行うかどうかを確認するためのテストを行う必要があります。

`ScriptEngine.put` と、これに関連する `get()` メソッドは、スクリプト・エンジン内で実行する Java コードとスクリプトの間でオブジェクトとデータを共有するための基本的な方法です。(この話題の詳細については、この記事の後の方で説明する「スクリプト実行のスコープ」を参照してください。) スクリプト・エンジンの `put()` メソッドを呼び出すと、エンジンは指定されたストリング・キーと 2 番目のパラメーター (任意の Java オブジェクト) とを関連付けます。大部分のスクリプト・エンジンは、指定された変数名でこれらの Java オブジェクトをスクリプトが利用できるようにします。スクリプト・エンジンは、put() メソッドに渡す名前を自由に変更することができます。例えば JRuby スクリプト・エンジンは、グローバル変数用の Ruby の構文に合うように、`helloScriptingWorld` を `$helloScriptingWorld` というグローバル変数として Ruby コードで利用できるようにします。

スクリプト・エンジンの `get()` メソッドは、スクリプト環境の中で得られる値を取得します。一般的には、スクリプト環境の中にあるすべてのグローバル変数とグローバル関数は、Java コー

ドから `get()` メソッドを使ってアクセスすることができます。しかしスクリプトが利用できるのは、`put()` を使って明示的にスクリプト・エンジンと共有される Java オブジェクトのみです。

実行中のアプリケーションの Java オブジェクトに外部スクリプトがアクセスでき、操作できるという、この機能は、Java プログラムの機能を拡張するための強力な方法です。(第 2 回の例では、この方法の詳細を解説します。)

## HelloScriptingWorld アプリケーションを実行する

HelloScriptingWorld アプリケーションを実行するためには、ソース・コードをダウンロードしてビルドします。.zip ファイルには、サンプル・アプリケーションをコンパイルして実行するための Ant スクリプトと Maven ビルド・ファイルの両方が含まれています。下記のステップで作業を行います。

1. .zip ファイルをダウンロードします。
2. 新しいディレクトリー (例えば java-scripting など) を作成し、ステップ 1 でダウンロードしたファイルを、このディレクトリーに解凍します。
3. コマドライン・シェルを開き、このディレクトリーに変更します。
4. `ant run-hello` を実行します。

リスト 6 に示すような、Ant のコンソール出力が表示されるはずです。`defineScriptFunction()` メソッドが何も出力を生成していないことに注目してください。これは、このメソッドは JavaScript 関数を定義しても呼び出しはしないためです。

### HelloScriptingWorld を実行したことによる出力

```
Calling invokeHelloScript...
Hello from JavaScript

Calling defineScriptFunction...

Calling invokeScriptFunctionFromEngine...
Hello, World!

Calling invokeScriptFunctionFromJava...
Hello, from Java

Calling invokeJavaFromScriptFunction...
Invoking getHelloReply method from JavaScript...
Java returned: Java method getHelloReply says, 'Hello, JavaScript'
```

## Java 5 との互換性

Java SE 6 では Java スクリプト API が導入されましたが、この API を Java SE 5 で実行することもできます。そのためには、単に Java SE 5 に欠けている `javax.script` パッケージ・クラスの実装を提供すればよいだけです。幸い、Java Specification Request 223 のリファレンス実装の中に、利用できる 1 つの実装があります (「参考文献」のダウンロードへのリンクを参照してください)。JSR 223 は Java スクリプト API を定義しています。

JSR 223 リファレンス実装をダウンロードしたら、このファイルを解凍し、script-api.jar ファイルと script-js.jar ファイル、そして js.jar ファイルをクラスパスに配置します。これらのファイル

は、Java SE 6 にバンドルされているスクリプト API と JavaScript スクリプト・エンジン・インターフェース、そして JavaScript スクリプト・エンジンを提供します。

## スクリプト実行のスコープ

スクリプト・エンジンの中で実行中のスクリプトに Java オブジェクトを公開するための方法は、単にスクリプト・エンジンの `get()` メソッドと `put()` メソッドを呼び出す以外にも構成方法があります。スクリプト・エンジンの `get()` メソッドあるいは `put()` メソッドを呼び出すと、エンジンは要求された鍵を `javax.script.Bindings` インターフェースのデフォルトのインスタンスの中で取得するか、あるいはこのインスタンスの中に保存します。(`Bindings` インターフェースは、鍵を強制的にストリングにする単なる `Map` インターフェースです。)

スクリプト・エンジンの `eval()` メソッドをコードから呼び出すと、スクリプト・エンジンにデフォルトで設定されている鍵と値のバインディングが使われます。しかし `eval()` メソッドを呼び出す際に独自の `Bindings` オブジェクトを提供すれば、その特定のスクリプトからどの変数とオブジェクトが見えるかを制限することができます。この呼び出しは、`eval(String, Bindings)` あるいは `eval(Reader, Bindings)` のようになります。スクリプト・エンジンは、カスタマイズした `Bindings` を作りやすいように、空の `Bindings` オブジェクトを返す `createBindings()` メソッドを提供しています。`Bindings` オブジェクトを使って `eval()` メソッドを呼び出すことで、その前にスクリプト・エンジンのデフォルト・バインディングに保存された Java オブジェクトを一時的に隠すことができます。

さらに補足しておくと、スクリプト・エンジンには 2 つのデフォルト・バインディングが含まれています。それは、`get()` と `put()` を呼び出す際に使われる「エンジン・スコープ」バインディングと、「エンジン・スコープ」バインディングの中にオブジェクトが見つからない場合にスクリプト・エンジンがオブジェクトを参照するために使用することができる「グローバル・スコープ」バインディングです。『できる』という言葉に重要な意味があります。つまりスクリプト・エンジンは、必ずしもグローバル・バインディングをスクリプトから利用できるようにする必要はないのです。しかし大部分のスクリプト・エンジンは利用できるようにしています。

「グローバル・スコープ」バインディングが設計されている背景には、オブジェクトをさまざまなスクリプト・エンジンで共有するという目的があります。`ScriptEngineManager` インスタンスが返すすべてのスクリプト・エンジンは、同じ「グローバル・スコープ」バインディング・オブジェクトを共有しています。あるエンジンのグローバル・バインディングは `getBindings(ScriptContext.GLOBAL_SCOPE)` メソッドを使って取得することができ、また `setBindings(Bindings, ScriptContext.GLOBAL_SCOPE)` を使って設定することができます。

ScriptContext は、スクリプト・エンジンのランタイム・コンテキストを定義し、そしてコントロールするインターフェースです。スクリプト・エンジンの ScriptContext に含まれるものには、「エンジン」スコープと「グローバル」スコープのバインディング、そしてエンジンが標準の入出力操作に使用する入出力ストリームがあります。スクリプト・エンジンのコンテキストを取得し、操作するには、そのエンジンの `getContext()` メソッドを使います。

スコープやバインディング、コンテキストなど、スクリプト API の概念は、それぞれが重複する意味を持っているために、最初はわかりにくいかもしれません。この記事のソース・コードのダウンロード・ファイルには、`ScriptApiRhinoTest` と呼ばれる JUnit テスト・ファイルが src/test/

java ディレクトリーに含まれており、これらの概念を Java コードを通して理解するために役立つはずです。

## 次回は

Java スクリプト API の基本事項は理解できたので、第 2 回では、この知識をもっと現実的なサンプル・アプリケーションを使って改善し、拡張します。このアプリケーションは、Groovy と Ruby、そして JavaScript を組み合わせて作成された外部スクリプト・ファイルを使って、実行時に変更できるビジネス・ロジックを定義します。第 2 回を読めばわかるように、スクリプト言語でビジネス・ルールを定義することでルールを記述しやすくなり、プログラマーではない人 (ビジネス・アナリストやルール作成者など) にとって、おそらくルールが読みやすくなります。

---

# ダウンロード

| 内容 | ファイル名 | サイズ |
| --- | --- | --- |
| Source code and JAR file | j-javascripting1.zip | 116KB |

# 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。