実用的な Groovy: Java プログラマーのための DSL としての Groovy

Groovy を使うことで、より少ないコード量でより多くを実現する

Scott Davis

2009年 2月 17日

Groovy のエキスパートである Scott Davis が、2006年以来休止していた「実用的な Groovy」シリーズを再開します。再開第 1 回目の今回は、最初にここ数年の Groovy に関する出来事を振り返り、Groovy とそれを取り巻く環境についての現状を紹介します。そして、2009年の現在、Groovy の学習を始めることがいかに容易であるかを実感してもらいます。

このシリーズの他の記事を見る

Andrew Glover は 2004年に developerWorks で Groovy に関する記事を書き始めました。その最初の記事は alt.lang.jre に関する連載の中の「Feeling Groovy」という入門記事であり、それ以降長期にわたり、「実用的な Groovy」シリーズが続きました。それはまだ Groovy に関する本が出版される前のことであり (現在では 10 数冊の本が出版されています)、2007年1月に Groovy 1.0 がリリースされる何年も前のことでした。「実用的な Groovy」の最後の記事が 2006年の終わり頃に公開されて以降、Groovy に関するさまざまなことが変わりました。

現在、Groovy のダウンロード数は毎月平均 35,000 です。Mutual of Omaha (訳注: 米国の金融、保険関連の会社) のような保守的な会社でも 70,000 行を超える Groovy コードが実際に使われています。Groovy のメーリング・リストは (Groovy プロジェクトをホストしている) Codehaus.org で最も活発なメーリング・リストの 1 つです (「参考文献」を参照)。Groovy よりもダウンロード数が多く、より活発なメーリング・リストを持つ唯一のプロジェクトは Grails です。Grails は人気のある Web フレームワークであり、Groovy で実装されています (「参考文献」を参照)。

JVM 上で Java™ 以外の言語を実行することは当たり前のことであるだけでなく、JVM に関する Sun の中核戦略の一部でもあります。Groovy は、Sun がサポートする Java の代替言語 (JavaScript、JavaFX、JRuby、Jython など) のうちの 1 つです。2004年には実験的であったものが、今や最先端なのです。

2009年に Groovy に関する記事を書くことは、Andy (訳注: 前出の Andrew Glover の愛称) が Groovy に関する記事を書き始めた頃と多くの点でよく似ています。Groovy の構文は 2005年に安定し、それ以来変っていません。魅力的な新しい機能がリリースごとに追加されていますが、プロジェクト・リーダーにとって最も優先されるのは後方互換性が維持されることです。Groovy にはこのよ

うに強固な基礎があるため、Java 開発の現場にとって Groovy は採用しやすい技術なのです。というのも、Java 開発の現場では、開発したアプリケーションが稼働している間は存続しているであろう技術を使用することになるからです。

この記事の目的は、経験豊富な Java 開発者がすぐに Groovy 開発者として活躍できるように Groovy を解説することです。一見、入門記事のように思えるかもしれませんが、そんなことはありません。この連載では名前のとおり、Groovy の実用的な使い方のノウハウを説明します。最初は易しい「Hello, World」の例を取り上げますが、その後はすぐに実用的な説明に入ります。

このシリーズについて

Groovy は Java プラットフォーム上で実行される最新のプログラミング言語の 1 つです。Groovy は既存の Java コードとシームレスに統合できる一方、クロージャーやメタプログラミングなどの強力な新機能も導入することができます。簡単に言えば Groovy とは、21 世紀に Java 言語が作成されていたら Groovy のようになっていたであろう、そういった言語なのです。

開発ツールキットの一部として新しいツールを採用する際に重要なことは、どういう場合にそのツールを使い、どういう場合には使わずにおくかを理解することです。Groovy は非常に強力ですが、適切な方法で、適切な状況の中で使用した場合にのみ強力なツールとなるのです。そのため「実用的な Groovy」シリーズでは、どういう状況で、どのようにして Groovy を使えば効果的であるかを学べるように、Groovy の実用的な使い方を解説します。

Groovy をインストールする

これまで Groovy を扱ったことがない人は、まず Groovy をインストールする必要があります。インストールの手順は非常に単純であり、Ant や Tomcat などの一般的な Java アプリケーション、さらには Java プラットフォーム自体をインストールする場合の手順と同じです。

1. Groovy の最新の ZIP ファイルまたは tarball をダウンロードします。
2. 適当なディレクトリーにアーカイブを解凍します。(名前に空白を使っているディレクトリーは避けてください。)
3. `GROOVY_HOME` 環境変数を作成します。
4. `PATH` に GROOVY_HOME/bin を追加します。

Groovy は Java 5 または Java 6 上で実行するのが最適です。コマンド・プロンプトで `java -version` と入力して Java のバージョンがこのいずれかの新しいバージョンであることを確認します。次に `groovy -version` と入力して Groovy が適切にインストールされていることを確認します。

主要な IDE (Eclipse、IntelliJ、NetBeans など) には、自動補完やステップ・デバッグなどの機能をサポートする Groovy 用のプラグインがあります。最近では Javaコードを作成する際には、優れた IDE を使用するのがほとんど必須条件となっていますが、Groovy に関しては必ずしもそうではありません。Groovy 言語はコンパクトなため、多くの人は IDE ではなく単純なテキスト・エディターを使っています。よく使われるオープンソースのエディター (vi や Emacs) は Groovy をサポートしており、また商用の安価なテキスト・エディター (Windows® 用の Textpad や Mac OS X 用の TextMate など) も Groovy をサポートしています (詳細は「参考文献」を参照)。

この記事で後ほど説明するように、既存の Java プロジェクトと Groovy とを組み合わせることは容易です。必要なことは、GROOVY_HOME/embeddable にある 1 つの Groovy JAR をクラスパスに

追加し、既存の javac Ant タスクを groovyc タスクの中にラップすることだけです。(Maven の場合も方法は似ています。)

先を急ぐ前に、まずはお決まりの「Hello World」の例から始めることにしましょう。

Groovy による Hello World

皆さんは、「Hello World」サンプルが何を実証するためのものなのかをご存知かと思います。「Hello World」は指定された言語で作成できる最も単純なプログラムです。Java コードで作成された「Hello World」(リスト 1) を見ると、ここで何が行われているかを完全に理解するには、中間言語の知識がどれほど必要か、という興味深い点に気付きます。

リスト 1. Java コードで作成した「Hello World」の例

```
public class HelloJavaWorld{
  public static void main(String[] args){
    System.out.println("Hello Java World");
  }
}
```

まず HelloJavaWorld.java という名前のファイルを作成し、次に public class HelloJavaWorld と入力します。多くの Java 初心者が最初に学ぶ難しいレッスンは、クラス名とファイル名が (大文字小文字の区別を含めて) 正確に一致しないとクラスをコンパイルできない、という事実です。また好奇心の強い生徒であれば、この時点で public や private などのアクセス修飾子について質問し始めます。

その次の行 (public static void main(String[] args)) になると、実装に関する細かい質問が次から次へと出始めるのが普通です。つまり static とは何か、void とは何か、なぜメソッドの名前が main でなければならないのか、String 配列とは何か、といった質問です。そして最後に、初めて Java を扱う開発者に対して、out が System クラスの PrintStream オブジェクトの、public で static かつ final のインスタンスであることを説明してみてください。私は「ちぇ、僕は「Hello」と表示したかっただけなのに」と言った生徒を忘れることができません。

これを Groovy で作成した「Hello World」と比べてみてください。Groovy の場合には、HelloGroovyWorld.groovy という名前のファイルを作成し、リスト 2 に示す行を入力すればよいだけです。

リスト 2. Groovy で作成した「Hello World」の例

```
println "Hello Groovy World"
```

そうです。これがリスト 1 の Java の例と等価なものを Groovy で作成した例なのです。この場合、実装に関するすべての詳細 (つまり当面の問題解決には直接関係しない「荷物」) は背後に隠され、単純に「Hello」と出力するコードのみが残っています。groovy HelloGroovyWorld と入力すると、このコードが実際に動作することを確認することができます。

この非常に簡単な例から、Groovy の持つ 2 つの価値命題がわかります。つまり Groovy を利用することによって、作成するコードの行数を大幅に削減することができる一方、Java と等価なセマンティクスを維持することができるのです。次のセクションでは、この概念をさらに詳しく調べることにします。

Hello World を深く掘り下げる

経験豊富な Java 開発者であれば、JVM 上で実行する前にコードをコンパイルする必要があることを知っています。ところが、Groovy スクリプトのためのクラス・ファイルはどこにも見当たらないようです。これは Groovy のソース・コードを直接実行できるということなのでしょうか。答えは、「決してそんなことはありませんが、実際そのように見えることは確かです。」

Groovy インタープリターはソース・コードをメモリー内でコンパイルしてから JVM に渡します。このステップを手動で行うには `groovyc HelloGroovyWorld.groovy` と入力します。しかし、その結果作成されるクラスを `java` を使って実行しようとすると、リスト 3 に示す例外が表示されます。

リスト 3. コンパイルされた Groovy クラスを **CLASSPATH** に Groovy JAR がない状態で実行しようとする

```
$ java HelloGroovyWorld
Exception in thread "main" java.lang.NoClassDefFoundError: groovy/lang/Script
```

先ほど触れたように、Groovy JAR を `CLASSPATH` に含める必要があります。もう一度、今度は `-classpath` 引数を `java` に渡して試してみます (リスト 4)。

リスト 4. コンパイルされた Groovy クラスを **java** コマンドを使って実行すると成功する

```
//For UNIX, Linux, and Mac OS X
$ java -classpath $GROOVY_HOME/embeddable/groovy-all-x.y.z.jar:. HelloGroovyWorld
Hello Groovy World

//For Windows
$ java -classpath %GROOVY_HOME%/embeddable/groovy-all-x.y.z.jar;. HelloGroovyWorld
Hello Groovy World
```

これで、何となくわかってきました。しかし Groovy スクリプトが Java の例のセマンティクスを実際に保持していることを証明するためには、さらに詳しくバイトコードを調べる必要があります。まず、`javap HelloJavaWorld` と入力します (リスト 5)。

リスト 5. Java バイトコードを検証する

```
$ javap HelloJavaWorld
Compiled from "HelloJavaWorld.java"
public class HelloJavaWorld extends java.lang.Object{
  public HelloJavaWorld();
  public static void main(java.lang.String[]);
}
```

ここには驚くようなものは大してないはずですが、`javac` コンパイラーが追加してくれた便利なものがいくつかあります。javac コンパイラーのおかげで `extends java.lang.Object` と明示的に入力する必要がなく、またクラスに対してデフォルト・コンストラクターを追加する必要もありませんでした。

さて、ここで `javap HelloGroovyWorld` と入力してみます (リスト 6)。

リスト 6. Groovy のバイトコードを検証する

```
$ javap HelloGroovyWorld
Compiled from "HelloGroovyWorld.groovy"
public class HelloGroovyWorld extends groovy.lang.Script{
  ...
  public static void main(java.lang.String[]);
  ...
}
```

DSL とは何か

Martin Fowler はドメイン特化言語 (DSL: Domain-Specific Language) の概念を普及させました (「参考文献」を参照)。彼は DSL を「ある特定のドメインに焦点を絞り、表現力を限定したコンピューター・プログラミング言語」と定義しています。「表現力を限定」とは実用性を制限するという意味ではなく、その言語は「特定のドメイン」に適したボキャブラリーしか持たないという意味です。DSL は、Java 言語のように大規模で汎用の言語とは対照的で、特定の目的のための簡単な言語なのです。

SQL は DSL の好例です。SQL でオペレーティング・システムを作成することはできませんが、リレーショナル・データベースの処理という限られたドメインでは SQL は理想的です。それと同じように Groovy は、Java 開発という限られたドメインにとって理想的なものという意味で、Java プラットフォームにとっての DSL です。ここで私は、厳密な意味での DSL であると言っているのではなく、DSL を連想させるような役割をするものという意味で、DSL という言葉を使っています。Groovy のことを、一般的な Java イディオムのための内部 DSL、と言えば、DSL を純粋に考える人達の怒りを静められるかもしれません。

Dave Thomas は、DSL の概念をさらに明確にしています (「参考文献」を参照)。彼は、「ドメインのエキスパート同士が会話する際には、・・・彼らは必ず業界用語で会話します。業界用語とは、仲間と効率的に会話するための省略形として彼らが考え出した特別な言語です」と書いています。おそらく、Groovy を「Java の省略形」だと言った方が Groovy と Java 言語との関係をより正確に説明したことになるかもしれません。この記事の次のセクションでは、これについてのもう 1 つの例を説明します。

これを見るとわかるように、`groovyc` コンパイラーがソース・ファイルの名前を引数に取り、それと同じ名前のクラスを作成しています。(このクラスが `java.lang.Object` ではなく `groovy.lang.Script` を継承しているという事実を見ると、なぜ `CLASSPATH` に Groovy JAR がない状態でファイルを実行しようとすると `NoClassDefFoundError` 例外がスローされるのかを理解しやすくなるはずです)。それ以外の、コンパイラーによって提供されるメソッド群を見ると、おなじみの `public static void main(String[] args)` メソッドがあることに気付くはずです。`groovyc` コンパイラーはこのメソッドの中にスクリプトのコード行をラップして Java のセマンティクスを保持します。これはつまり、Groovy の場合には Java に関する既存の知識をすべて活用できるということです。

例えば、Groovy スクリプトの中でコマンドライン入力を受け付けるためには次のようにします。Hello.groovy という名前の新しいファイルを作成し、リスト 7 のコード行を追加します。

リスト 7. コマンドライン入力を受け付ける Groovy スクリプト

```
println "Hello, " + args[0]
```

今度はコマンドラインから `groovy Hello Jane` と入力します。Java 開発者の誰もが期待するように、リスト 7 には `argsString` 配列があります。ここで `args` を使うことは初心者には理解できないかもしれませんが、経験豊富な Java 開発者には完璧に理解できるはずです。

Groovy によって、Java コードは必要最低限の本質的なもののみになります。今、上で作成した Groovy スクリプトは、実行可能な疑似コードとほとんど同じように見えます。表面的には初心者にとって非常に理解しやすくなっていますが、経験豊富な開発者にとってはベースにある Java 言語の強力さが失われることはありません。このことから、私は Groovy を Java プラットフォームのための DSL (Domain-Specific Language: ドメイン特化言語) と呼ぶことにしました。(囲み記事「DSL とは何か」を参照してください。)

昔ながらの普通の Groovy オブジェクト

JavaBeans (もっとくだけた呼び方をすれば POJO: Plain Old Java Object) は Java 開発の中心となるものです。POJO を作成してドメイン・オブジェクトを表現する際には、明確な想定事項に従う必要があります。例えば、クラスは `public` でなければならず、各フィールドはそれぞれに対応する一連の `public` なゲッター・メソッドとセッター・メソッドを持ち、`private` でなければなりません。リスト 8 は典型的な Java の POJO を表しています。

リスト 8. Java の POJO

```
public class JavaPerson{
  private String firstName;
  private String lastName;

  public String getFirstName(){ return firstName; }
  public void setFirstName(String firstName){ this.firstName = firstName; }

  public String getLastName(){ return lastName; }
  public void setLastName(String lastName){ this.lastName = lastName; }
}
```

POGO (Plain Old Groovy Object) はそのまま POJO と置き換えることができます。POGO によって POJO のセマンティクスを完全に維持できる一方、作成が必要なコード量を大幅に削減することができます。リスト 9 は Groovy で作成した「省略版」の person クラスを示しています。

リスト 9. Groovy の POGO

```
class GroovyPerson{
  String firstName
  String lastName
}
```

Groovy のクラスは特に指定しない限り、すべて `public` です。すべてのプロパティーは `private` であり、すべてのメソッドは `public` です。コンパイラーは各プロパティーに対して一連の `public` なゲッター・メソッドとセッター・メソッドを自動的に提供します。`JavaPerson` を `javac` でコンパイルし、`GroovyPerson` を `groovyc` でコンパイルし、両方とも `javap` を使って実行すると、Groovy 版の中には (`java.lang.Object` を継承することに至るまで) Java 版の中にあるものがすべて入っていることを確認することができます。(先ほどの `HelloGroovyWorld` の例ではクラスを指定しませんでした。そのため `groovy.lang.Script` を継承するクラスを Groovy が作成してくれたのです。)

これらはすべて、POJO を POGO で置き換えてそのまま POGO を使い始められることを意味しています。つまり Groovy クラスは Java クラスの必要最低限の本質を抽出したものなのです。いっ

たん Groovy クラスがコンパイルされると、その Groovy クラスがあたかも Java で作成されたものであるかのように、他の Java クラスはその Groovy クラスを手軽に使うことができます。それを証明するために、JavaTest.java という名前のファイルを作成し、リスト 10 のコードを追加します。

リスト 10. Java コードから Groovy クラスを呼び出す

```
public class JavaTest{
  public static void main(String[] args){
    JavaPerson jp = new JavaPerson();
    jp.setFirstName("John");
    jp.setLastName("Doe");
    System.out.println("Hello " + jp.getFirstName());

    GroovyPerson gp = new GroovyPerson();
    gp.setFirstName("Jane");
    gp.setLastName("Smith");
    System.out.println("Hello " + gp.getFirstName());
  }
}
```

Groovy のソース・コードにはゲッターもセッターも現れませんが、このテストから、コンパイルされた Groovy クラスの中にはゲッターとセッターがあり、それらが完全に機能することを証明することができます。しかし、これに対応する Groovy で作成したテストを示さないと、この例が完結しません。そこで TestGroovy.groovy という名前のファイルを作成し、リスト 11 のコードを追加します。

リスト 11. Groovy から Java クラスを呼び出す

```
JavaPerson jp = new JavaPerson(firstName:"John", lastName:"Doe")
println "Greetings, " + jp.getFirstName() + ".
   It is a pleasure to make your acquaintance."

GroovyPerson gp = new GroovyPerson(lastName:"Smith", firstName:"Jane")
println "Howdy, ${gp.firstName}. How the heck are you?"
```

これを見ておそらく最初に気付くことは、Groovy によって新しいコンストラクターが提供されていることでしょう。このコンストラクターを使うことで各フィールドに名前を付けることができ、それらのフィールドを任意の順序で指定することができます。もっと興味深いことは、このコンストラクターを Java クラスでも Groovy クラスでも使用できるという事実です。なぜそんなことができるのでしょう。実際には、Groovy は引数を持たないデフォルト・コンストラクターを最初に呼び出し、次に各フィールドに対する適切なセッターを呼び出します。同様の動作を Java 言語でも真似ることはできますが、Java 言語には名前付き引数がなく、またどちらのフィールドも `String` なので、どのような順序でも firstName フィールドと lastName フィールドに引数を渡すことはできません。

次に、Groovy では従来からの Java 流の方法で `String` を連結することもできれば、Groovy 流の方法 (`String` の中で直接 `${}` で囲むことでコードを埋め込む) も可能なことに注目してください。(これらのストリングは、Groovy Strings を略して `GStrings` と呼ばれます。)

最後にもう 1 つ、クラスのゲッターを呼び出す際に使用できる Groovy のシンタックス・シュガーがあることがわかります。冗長な `gp.getFirstName()` を使う代わりに、単純に `gp.firstName` を呼

び出すことができます。これはフィールドに直接アクセスしているように思えるかもしれませんが、実際には対応するゲッター・メソッドを背後で呼び出しています。セッターの動作も同様です。つまり `gp.setLastName("Jones")` と `gp.lastName = "Jones"` は等価であり、後者は見えないところで前者を呼び出しています。

どの場合も、Groovy が Java 言語の省略版のように感じられること、つまり「ドメインのエキスパート」が「仲間と効率的に会話するため」に使用するような、あるいは古い友人達の間での気さくな冗談に交わされるような省略語に感じられることに、皆さんも同意するのではないかと思います。

Groovy は結局のところ Java コードです

Groovy の持つさまざまな側面の中で最も過小評価されている側面の 1 つが、Groovy は Java の構文を完全にサポートしているという事実です。先ほど触れたように、Groovy を扱うために Java の知識を捨て去る必要はまったくありません。Groovy を使い始めると、最初は皆さんが作成するコードは従来の Java コードと非常に似たものになってしまうでしょう。しかし新しい構文に慣れてくると、コードは徐々に進化し、より簡潔で表現力に富んだ Groovy スタイルを活用するようになります。

Groovy のコードが Java のコードとまったく同じようであることを証明するためには、JavaTest.java を JavaTestInGroovy.groovy にコピーし、`groovy JavaTestInGroovy` と入力してみます。すると、まったく同じ出力が表示されるはずです。ただし、Groovy クラスをコンパイルしてから実行する必要がなかったことに注目してください。

このことから、経験豊富な Java 開発者にとって Groovy がほとんど頭を使わなくてよい手軽な選択肢であることがわかると思います。Java の構文は Groovy の構文としても有効なので、初期の学習期間はほとんど必要ありません。既存の Java バージョンを、Groovy と既存の IDE、そして既存の実動環境と組み合わせて使用することができます。これはつまり、毎日の定常的な作業への影響が最小限になるということです。必要なことは、`CLASSPATH` のどこかに必ず Groovy JAR を置き、ビルド・スクリプトを若干変更して Groovy クラスが Java クラスと一緒にコンパイルされるようにすることだけです。次のセクションでは Ant の build.xml ファイルに `groovyc` タスクを追加する方法を説明します。

Ant を使って Groovy コードをコンパイルする

もし `javac` がプラグ可能なコンパイラーであったとしたら、Groovy ファイルと Java ファイルの両方を同時にコンパイルするように javac に命令することができます。しかし `javac` はプラグ可能ではないため、単純に Ant の中で `javac` タスクを `groovyc` タスクでラップします。そうすることで `groovyc` は Groovy のソース・コードをコンパイルできるようになり、また `javac` はそれまでと同様に Java のソース・コードをコンパイルすることができます。

もちろん `groovyc` は Java ファイルと Groovy ファイルの両方をコンパイルすることができますが、`groovyc` によって `HelloGroovyWorld` と `GroovyPerson` にコンビニエンス・メソッドが追加されたことを思い出してください。これらの追加メソッドを Java クラスにも追加することができます。おそらく、`groovyc` で Groovy ファイルをコンパイルし、`javac` で Java ファイルをコンパイルすることが最善でしょう。

Ant から `groovyc` を呼び出すためには、`taskdef` を使って `groovyc` タスクを定義し、通常 `javac` タスクを使う場合と同じように `groovyc` タスクを使います (詳細は「参考文献」を参照)。リスト 12 は Ant のビルド・スクリプトを示しています。

リスト 12. Ant を使って Groovy と Java のコードを コンパイルする

```
<taskdef name="groovyc"
         classname="org.codehaus.groovy.ant.Groovyc"
         classpathref="my.classpath"/>

<groovyc srcdir="${testSourceDirectory}" destdir="${testClassesDirectory}">
 <classpath>
   <pathelement path="${mainClassesDirectory}"/>
   <pathelement path="${testClassesDirectory}"/>
   <path refid="testPath"/>
 </classpath>
 <javac debug="on" />
</groovyc>
```

ところで、内部に `${}` を持つ `String` は驚くほど `GString` に似ていないでしょうか。Groovy は最善のものを組み合わせた言語であり、他のさまざまな言語やライブラリーから構文や機能を遠慮なく借用しています。Groovy の何かを見て、「以前どこかで見たことがある」と思うことはこれからもあるはずです。

まとめ

この記事では Groovy を大急ぎで紹介しました。最初に Groovy のこれまでの状況と、現在 Groovy がどのような状況にあるかを少し学びました。システムに Groovy をインストールし、いくつかの簡単な例をとおして、Java 開発者が活用できる Groovy の強力さの一端を調べました。

Groovy は JVM 上で実行される唯一の Java 代替言語ではありません。既に Ruby を知っている Java 開発者にとっては JRuby が優れたソリューションであり、既に Python を知っている Java 開発者にとっては Jython が優れたソリューションです。しかしここで見てきたように、Groovy はJRuby や Jython とは違い、既に Java 言語を知っている Java 開発者にとって優れたソリューションなのです。Groovy の構文が Java 風の簡潔な構文であり、そうした構文が Java のセマンティクスも保持するという事実は非常に魅力的です。しかも、新しい言語を採用するために `del *.*` や `rm -Rf *` を行う必要がないとしたら、その言語に変更してみるのも悪くないと思えるのではないでしょうか。

次回の記事では Groovy での繰り返しについて学びます。リストであれファイルであれ、あるいは XML 文書であれ、項目ごとにウォークスルーしなければならない場合がよくあります。そこで、よく使われる `each` クロージャーを詳細に調べます。では次回まで、皆さんが Groovy の実用的な使い方をたくさん見つけられることを祈っています。

関連トピック

- Scott Davis による最新の本、『Groovy Recipes』(Pragmatic Programmers、2008年刊) を読み、Groovy と Grails について学んでください。
- Groovy と Grails のダウンロードの統計情報を調べてください。
- Groovy のメーリング・リストをブラウズ、検索、または購読してください。
- Martin Fowler による Using Domain Specific Languages を読み、DSL についての彼の意見を学んでください。
- 「The 'Language' in Domain-Specific Language Doesn't Mean English (or French, or Japanese, or ...)」(Dave Thomas 著、PragDave、2008年3月) では、DSL のファンである Thomas が DSL についてのコメントを綴っています。
- developerWorks の Java technology ゾーンには Java プログラミングのあらゆる側面を網羅した記事が豊富に用意されています。