# Java プログラミングのダイナミックス: 第 1 回 クラスとクラスのロード処理

## クラスおよび JVM がクラスをロードするときの動作についての研究

Dennis Sosnoski
President
Sosnoski Software Solutions, Inc.

2003年 4月 29日

Java プログラミングのダイナミックな側面に光を当てるこの新しいシリーズで、Java アプリケーションを実行する際に舞台裏でどんなことが行われているのかを学んでみましょう。エンタープライズ Java の専門家である DennisSosnoski 氏が、Java のバイナリー・クラス・フォーマットや JVM 内部でのクラスの扱いについて未聞の情報を紹介してくれます。それと同時に、簡単な Java アプリケーションを実行するのに必要なクラスの数から始まり、J2EE などの複雑なアーキテクチャーで問題を起こしたりするクラス・ローダーの衝突に至るまで、クラスのロード処理の問題についても説明してくれます。

このシリーズの他の記事を見る

### このシリーズのその他の記事


- 第 1 回 クラスとクラスのロード処理
- 第 2 回 リフレクション入門
- 第 3 回 実用的なリフレクション
- 第 4 回 Javassist でのクラス変換
- 第 5 回 オンザフライでクラスを変換する
- 第 6 回 Javassist を使用したアスペクト指向の変更
- 第 7 回 Bytecode engineering with BCEL (英語)
- 第 8 回 リフレクションに取って代わるコード生成


本稿は、私が Java プログラミングのダイナミックス と呼んでいる一連のテーマを紹介していく新しいシリーズの第 1 回目です。これらのテーマには、Java バイナリー・クラス・ファイル・フォーマットの基本的な構造に始まり、リフレクションを使用しての実行時メタデータ・アクセス、さらには、実行時に新しいクラスを修正したり構築する方法に至るまでの内容が含まれます。これらの題材すべてに共通するのは、Java プラットフォームのプログラミングが、素直にネイティブ・コードにコンパイルされる言語を扱う場合よりも、はるかにダイナミックであるという考えです。こうしたダイナミックな側面を理解できれば、主流となっている他のどんなプログラミング言語を使ってもかなわないようなことを Java プログラミングで行えるようになります。

今回は、Java プラットフォームのこうしたダイナミックな性質の基盤となっている基本的な概念をいくつか紹介したいと思います。これらの概念を、クラスが JVM にロードされる際にどんなことが行われるのかなど、Java クラスの表現に用いられるバイナリー・フォーマットの観点から展開していきます。このテーマは、本シリーズの次回以降の記事の基礎となるだけでなく、Java プラットフォームで開発を行う技術者にとって非常に実際的な問題もいくつか明らかにします。

## クラスのバイナリー形式

Java 言語で開発を行っている技術者は、通常、ソース・コードがコンパイラーにかけられるときどんな処理を受けるのか細かいことまで関知する必要はありませんが、私はこのシリーズで、ソース・コードからプログラムの実行に至るまでの舞台裏の詳細をいろいろと紹介したいと考えています。そこでまず、コンパイラーによって生成されるバイナリー・クラスについて研究してみたいと思います。

バイナリー・クラス・フォーマットを実際に定義しているのは、JVM の仕様です。通常これらのクラス表現は、Java 言語のソース・コードを基にコンパイラーによって生成され、通常 `.class` という拡張子のファイルに保存されます。しかし、これらの性質はいずれも必須だというわけではありません。Java バイナリー・クラス・フォーマットを利用するプログラミング言語が他にも開発されてきていますし、いろいろな目的から、新しいクラス表現が構築され、実行中の JVM に直接ロードされたりもしています。JVM にとって重要なのは、ソースやソースがどのように保存されているかではなく、フォーマットそのものです。

では、そのクラス・フォーマットは、実際どんなものなのでしょうか。リスト 1 は、(非常に) 短かなクラスのソース・コードと、コンパイラーが出力したクラス・ファイルの 16進表示の一部を示したものです。

### リスト 1. Hello.java のソースとバイナリー (の一部)

```
public class Hello
{
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
0000: cafe babe 0000 002e 001a 0a00 0600 0c09  ................
0010: 000d 000e 0800 0f0a 0010 0011 0700 1207  ................
0020: 0013 0100 063c 696e 6974 3e01 0003 2829  .....<init>...()
0030: 5601 0004 436f 6465 0100 046d 6169 6e01  V...Code...main.
0040: 0016 285b 4c6a 6176 612f 6c61 6e67 2f53  ..([Ljava/lang/S
0050: 7472 696e 673b 2956 0c00 0700 0807 0014  tring;)V........
0060: 0c00 1500 1601 000d 4865 6c6c 6f2c 2057  ........Hello, W
0070: 6f72 6c64 2107 0017 0c00 1800 1901 0005  orld!...........
0080: 4865 6c6c 6f01 0010 6a61 7661 2f6c 616e  Hello...java/lan
0090: 672f 4f62 6a65 6374 0100 106a 6176 612f  g/Object...java/
00a0: 6c61 6e67 2f53 7973 7465 6d01 0003 6f75  lang/System...ou
...
```

### バイナリーの内部

リスト 1 に示したバイナリー・クラス表現で真っ先にきているのが、Java バイナリー・クラス・フォーマットであることを表す cafebabe というシグニチャーです (ついでに言うと、これは、Java プラットフォームを構築しようという開発者魂を保ちながら一生懸命がんばったバリスタたち への、いつまでも残るが、広く知られることのない感謝の印にもなっています)。このシグニチャー

により、データ・ブロックが Java クラス・フォーマットのインスタンスであることを表明している のかどうかを簡単に確認できます。Java バイナリー・クラスは、ファイル・システム上にないものも含め、すべて、この 4バイトで始まっている必要があります。

それ以降のデータは、それほど面白くはありません。シグニチャーの後には、クラス・フォーマットの 1対のバージョン番号 (上の例では、1.4.1 の javac が生成したコードとして、マイナー・バージョンの 0 とメジャー・バージョンの 46、すなわち 16進数の 0x2e になっている) が続き、さらに定数プールのエントリー数がきます。エントリー数 (上の例では 26、すなわち 0x001a) の後には、実際の定数プールのデータがきます。ここには、クラス定義で使用されるすべての定数が格納されます。この中には、各種のバイナリー値の他に、クラスやメソッドの名前、シグニチャー、文字列 (文字列は、16進ダンプの右側にテキストとして読むことができます) が含まれます。

定数プールの項目は可変長であり、各項目の第 1バイト目は、項目の型およびそのデコード方法を示しています。ここでは、これについて深入りしないことにします。興味のある方は、JVM の仕様を始めとして参考文献がたくさんありますので、そういったものを参照してください。重要なのは、このクラスおよびそのメソッドの実際の定義だけでなく、このクラスで使用される他のクラスやメソッドへの参照もすべて、定数プールに含まれるという点です。定数プールが、バイナリー・クラスのサイズの半分以上を占めることもよく起こります。もっとも、平均的には半分以下なのでしょうが。

定数プールの次には、そのクラス自体、スーパー・クラス、およびインターフェースの定数プールのエントリーを参照する項目がいくつかきます。これらの項目の次には、フィールドやメソッドについての情報が格納されます。フィールドやメソッドは、それ自身、複雑な構造として表現されます。メソッドの実行コードは、メソッド定義の中にコード属性 の形で含まれます。このコードは、JVM に対する命令の形式をしており、一般にバイトコード と呼ばれています。これが、以下の節でとりあげるテーマの 1つです。

属性 は、Java クラス・フォーマットの中で、上で触れたバイトコードやフィールドの定数値、例外処理、デバッグ情報など、定義されたいくつかの目的で使用されます。ただし、これらの目的だけが属性の用途だというわけではありません。JVM は未知の型の属性を無視しなければならないというのが、当初からの JVM の仕様となっています。この要件によって、属性の用途は、フレームワークがユーザー・クラスと連携する際に必要となるメタ情報を提供するなど (Java から派生した C# 言語が随所で使用してきている手法)、柔軟に拡張することができ、将来他の目的に利用できるようになっています。ただ、残念ながら、ユーザー・レベルでこの柔軟性を利用するためのフックは、まだ実現されていません。

## バイトコードとスタック

クラス・ファイルの実行コードの部分を構成しているバイトコードは、JVM という特殊なコンピューターのためのマシン・コードとなっています。これは、もともとハードウェアではなくソフトウェアで実装するために設計されたコンピューターであり、仮想 マシンと呼ばれています。Java プラットフォーム・アプリケーションを実行するための JVM は、すべて、このマシンの実装を中心に構築されています。

といっても、この仮想マシンは、非常に簡単なもので、スタック構造を採用しています。すなわち、命令オペランドを使用する場合、まず、それを内部スタックにロードしておきます。命令セットには、通常の算術命令や論理命令がすべて用意されている他、条件分岐、無条件分岐、ロード/ストア、コール/リターン、スタック操作、および特殊な命令がいくつか用意されています。命令の中に即値オペランド (immediate operand values) が直接コード化される命令もあります。その他の命令は、定数プールの値を直接参照します。

仮想マシンの構造が簡単だからといって、必ずしもその実装が簡単だというわけではありません。初期の (第 1 世代の) JVM は、基本的に、仮想マシンのバイトコードのインタープリターでした。これらのインタープリターは、確かに、比較的単純な ものでしたが、性能に関しては大きな問題がありました。ネイティブ・コードの実行に比べ、コードの解釈実行 (interpret) には時間がかかるのが常です。こうした性能上の問題を軽減するために、第 2 世代の JVM には、ジャストインタイム (JiT) 変換機能が追加されました。JiT とは、Java のバイトコードを一番最初に実行するときに、それをまずネイティブ・コードにコンパイルしておくことで、繰り返し実行されるときの性能を格段に向上させる技法のことです。現在の世代の JVM は、さらに進歩しており、プログラムの実行を監視し、頻繁に使用されるコードを選択的に最適化するという適応的手法を採用しています。

## クラスのロード

コンパイルでネイティブ・コードを作成する C や C++ などの言語は、通常、ソース・コードをコンパイルした後、リンクを行う必要があります。このリンク処理では、ソース・ファイルを別々にコンパイルして得られたコードおよび共有のライブラリー・コードをマージして、1個の実行プログラムが作成されます。しかし、Java 言語は違います。Java 言語の場合、コンパイラーで生成されたクラスは、通常、JVM にロードされるまでは、そのままにされます。クラス・ファイルから JAR ファイルを構築する場合でも、これに変わりはありません。JAR は、クラス・ファイルのコンテナーにすぎません。

クラスのリンクは、別個の手順としてではなく、JVM がクラスをメモリーにロードする際の処理の一部として行われます。この場合、クラスを最初にロードする際に、ある程度のオーバーヘッドが生じますが、Java アプリケーションには高度な柔軟性をもたらします。たとえば、実装コードが実行時まで特定されなくても、そのコードへのインターフェースを使ってアプリケーションを記述することができます。アプリケーションを組み立てるときの、この実行時バインディング (late binding) という手法は、Java プラットフォームでは随所で使用されており、サーブレットがそのよい例です。

クラスをロードする際の規則は、JVM の仕様に詳しく明記されています。基本的な原則として、クラスは、必要になったときに初めてロードされます(あるいは、少なくともそのような方法でロードされるように見えます。実際のロード方法に関しては、JVM がある程度柔軟に処理できますが、クラスの初期化に関しては、決まった手順に従う必要があります)。ロードされるクラスが別のクラスに依存していたりしますので、ロード処理は再帰的に行われます。リスト 2 のいくつかのクラスは、この再帰的なロードの動作原理を示しています。`Demo` クラスは、`Greeter` のインスタンスを作成し、`greet` メソッドを呼び出す簡単な `main` メソッドを含んでいます。`Greeter` のコンストラクターは、`Message` のインスタンスを作成し、それを `greet` メソッド呼び出しの中で使用しています。

## リスト 2. クラスのロード処理を説明するためのソース・コード

```
public class Demo
{
    public static void main(String[] args) {
        System.out.println("**beginning execution**");
        Greeter greeter = new Greeter();
        System.out.println("**created Greeter**");
        greeter.greet();
    }
}
public class Greeter
{
    private static Message s_message = new Message("Hello, World!");
    public void greet() {
        s_message.print(System.out);
    }
}
public class Message
{
    private String m_text;
    public Message(String text) {
        m_text = text;
    }
    public void print(java.io.PrintStream ps) {
        ps.println(m_text);
    }
}
```

`java` のコマンド・ラインに `-verbose:class` というパラメーターを指定すると、クラスのロード処理のトレースを表示することができます。リスト 3 は、このパラメーターを指定してリスト 2 のプログラムを実行したときの出力の一部です。

## リスト 3. -verbose:class による出力の一部

```
[Opened /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Opened /usr/java/j2sdk1.4.1/jre/lib/sunrsasign.jar]
[Opened /usr/java/j2sdk1.4.1/jre/lib/jsse.jar]
[Opened /usr/java/j2sdk1.4.1/jre/lib/jce.jar]
[Opened /usr/java/j2sdk1.4.1/jre/lib/charsets.jar]
[Loaded java.lang.Object from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.io.Serializable from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.lang.Comparable from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.lang.CharSequence from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.lang.String from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
...
[Loaded java.security.Principal from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.security.cert.Certificate from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded Demo]
**beginning execution**
[Loaded Greeter]
[Loaded Message]
**created Greeter**
Hello, World!
[Loaded java.util.HashMap$KeySet from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.util.HashMap$Keyiterator from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
```

ここには、最も重要な部分だけを示してあります。トレース全体では 294 行になりますので、このリストは、そのほとんどを削除したものになっています。クラスのロードの最初の方 (上の例では 279回) は、すべて、`Demo` クラスをロードしようとすることで起動されています。これらのクラスは、どんなに小さな Java プログラムであれ、すべてのプログラムが使用するコアのクラス群です。`Demo # main` メソッドからすべてのコードを削除したとしても、このロードの初期の手順は

変わりません。ただし、ロードされるクラスの数と名前は、クラス・ライブラリーのバージョンによって違ってきます。

`Demo` クラスをロードした後の部分のリストからは、もっと面白いことがわかります。ここに示されている手順から、`Greeter` クラスは、そのインスタンスが作成される段になって初めてロードされていることがわかります。一方、`Greeter` クラスは、`Message` クラスの静的インスタンスを使用していますので、前者のインスタンスを作成するためには、後者のクラスもロードする必要があります。

クラスがロードされ初期化されるとき、JVM 内部ではいろいろなことが行われます。バイナリー・クラス・フォーマットのデコード、他のクラスとの互換性のチェック、バイトコード命令のシーケンスの検査などが行われ、最後には、新しいクラスを表現するための `java.lang.Class` インスタンスが構築されます。この `Class` オブジェクトは、JVM で新たに作成されるクラスのすべてのインスタンスの基礎となります。また、`Class` オブジェクトは、ロードされたクラスそのものの iD でもあります (JVM にロードされる同じバイナリー・クラスから複数のコピーを設けることができるわけですが、それぞれのコピーに固有の `Class` インスタンスが生じます)。これらのコピーは、すべて同じクラス名を共有するにもかかわらず、JVM にとっては別々のクラスとして扱われます。

## 変わり種の (クラス) パス

JVM でのクラス・ロード処理は、クラス・ローダー によって制御されます。JVM には、Java クラス・ライブラリーの基本的なクラス群をロードするためのブートストラップ・クラス・ローダーが内蔵されています。このクラス・ローダーは、特別な機能を備えています。1つは、ブート・クラス・パスに存在するクラスだけをロードするということです。これらのクラスは信頼できるシステム・クラスですので、ブートストラップ・ローダーは、通常の (信頼できない) クラスに対して行われる妥当性チェックをほとんど省略します。

クラス・ローダーは、ブートストラップだけではありません。たとえば、JVM は、標準的な Java 拡張 APi によってクラスをロードするための拡張 クラス・ローダーや、一般的なクラスパスから (アプリケーション・クラスなどの) クラスをロードするためのシステム・クラス・ローダーを定義しています。アプリケーションも、(クラスを実行時に再ロードするなどの) 特別な用途に、独自のクラス・ローダーを定義することができます。そうしたクラス・ローダーを追加する場合、`java.lang.ClassLoader` クラスを派生させたもの (間接的な派生も含めて) が使われます。これは、バイト配列から内部的なクラス表現 (`java.lang.Class` インスタンス) を構築するときの中核的なサポート機能を提供するクラスです。構築されたクラスは、ある意味では、それをロードしたクラス・ローダーが「所有」しています。クラス・ローダーは、通常、ロードしたクラスのマップを作成し、次に要求があった場合には名前でクラスを検索できるようにします。

また、各クラス・ローダーは、親のクラス・ローダーへの参照も記録し、ブートストラップ・ローダーをルートとするクラス・ローダーのツリーを定義していきます。あるクラス (名前で識別される) のインスタンスが要求された場合、その要求を最初に処理するクラス・ローダーは、通常、直接そのクラスをロードせずに、まずその親のクラス・ローダーを最初にチェックします。クラス・ローダーが複数の階層になっている場合、これが再帰的に適用されていきます。したがって、クラスは通常、それをロードしたクラス・ローダーだけから見える のではなく、それより下位のすべてのクラス・ローダーにも見えることになります。また、クラスがチェーンの中

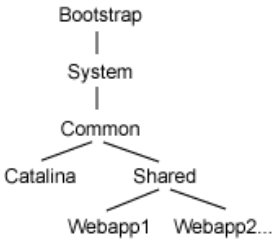
の複数のクラス・ローダーによってロード可能な場合、ツリーの中の一番上に位置するクラス・ローダーが、そのクラスを実際にロードすることになります。

Java プログラムによって複数のアプリケーション・クラス・ローダーが使用される状況には、いろいろな場合があります。たとえば、J2EE フレームワークがその例です。フレームワークによってロードされる J2EE アプリケーションには、それぞれ別個のクラス・ローダーを用意してやり、あるアプリケーションのクラス群が別のアプリケーションに干渉しないようにする必要があります。フレームワークのコード自体も、独自のクラス・ローダーを何個か使用し、やはりアプリケーションとの間で干渉を起こさないようにします。このようにして、クラス・ローダーの全体は、各レベルでロードされるクラスの種類が異なるような、ツリー構造の階層を構成します。

## ローダーのツリー

クラス・ローダー階層の実例として、図 1 に、Tomcat サーブレット・エンジンで定義されているクラス・ローダー階層を示します。この例では、Tomcat インストールの中のあるディレクトリーに入っている JAR ファイルから、サーバーとすべての Web アプリケーションの間でコードを共有するための Common というクラス・ローダーがロードされます。Catalina は Tomcat 自身のクラスをロードするためのローダーで、Shared は Web アプリケーション同士で共有されるクラスをロードするためのローダーです。そして最後に、Web アプリケーションは、それぞれのプライベートなクラスをロードするための独自のローダーを獲得します。

## 図 1. Tomcat のクラス・ローダー群




このような環境では、新しいクラスを要求する際にどのローダーを使用すべきなのかを管理しておくのが繁雑になったりします。そのため、Java2 プラットフォームでは、`java.lang.Thread` クラスに `setContextClassLoader` メソッドと `getContextClassLoader` メソッドが追加されました。これらのメソッドを利用することで、フレームワークは、アプリケーションのコードを実行させながら、それぞれのアプリケーションに使用すべきクラス・ローダーを設定することができます。

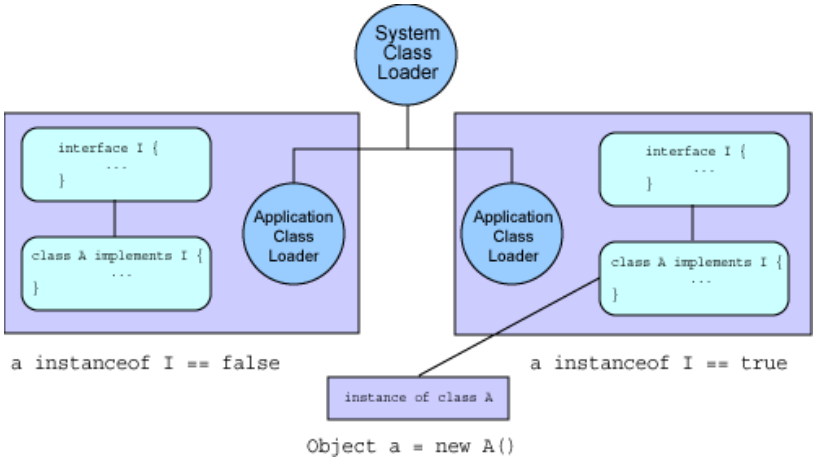
独立なクラス集合をロードできるという柔軟性は、Java プラットフォームの重要な機能です。ただし、便利な機能ではあるのですが、場合によっては、混乱を招くこともあります。混乱を招きやすいことの 1つに、JVM のクラスパスの扱いというなかなか決着の付かない問題があります。たとえば、図 1 に示した Tomcat のクラス・ローダー階層の場合、クラス・ローダー Common によってロードされたクラスは、Web アプリケーションでロードされたクラスを直接 (名前で) アクセスすることはできません。これらのクラスを結び付けるには、両方のクラス集合から見えるインターフェースを使用するしかありません。この場合、Java サーブレットによって実装された `javax.servlet.Servlet` もそれに含まれます。

また、何らかの理由でクラス・ローダー間でコードが移動される場合にも、問題が発生する場合があります。たとえば、J2SE1.4 が XML 処理用の JAXP APi を標準ディストリビューションに移したときには、それまでアプリケーションが独自に XMLAPi の実装を選択してロードするようにし

ていた数多くの環境で問題が発生しました。そうしたロードは、J2SE1.3 では、適切な JAR ファイルをユーザー・クラス・パスにインクルードすることで可能になっています。J2SE1.4 では、これらの APi の標準バージョンが拡張クラス・パスに配置されるようになりましたので、通常、これらの APi が、ユーザー・クラス・パスに存在する実装をオーバーライドすることになります。

他にも、複数のクラス・ローダーを使用する場合、混乱を起こす場合があります。図 2 は、インターフェースとそれに結合される実装が、それぞれ、別個の 2つのクラス・ローダーによってロードされる場合に起こるクラス識別の危機 の例を示したものです。インターフェースやクラスの名前やバイナリー実装は同じでも、あるローダーでロードされたクラスのインスタンスが別のローダーでロードされたインターフェースを実装しているものとは認識されません。こうした混乱は、図 2 のように、インターフェース・クラス `i` を System クラス・ローダーの空間に移してやることで解決できます。このようにしても、依然としてクラス `A` のインスタンスは別々に 2つ存在するわけですが、両方が同じインターフェース `i` を実装することになります。

## 図 2. クラス識別の危機




# まとめ

Java のクラス定義と JVM の仕様の組み合わせは、コードの実行時組み立てのための極めて強力なフレームワークを定義しています。クラス・ローダーを使用することで、Java アプリケーションは、いろいろなバージョンのクラスを衝突させないようにして扱うことができます。クラス・ローダーの柔軟性のおかげで、アプリケーションの実行中に、修正したコードを動的にロードし直すことすら可能です。

Java プラットフォームのこの面での柔軟性に対するコストは、アプリケーションを起動する際に、いくぶんオーバーヘッドが大きくなることです。ごく簡単なアプリケーション・コードの実行を開始する場合でも、JVM は、何百種類ものクラスをロードする必要があります。このような起動時のコストから、Java プラットフォームは、一般に、頻繁に使用される小さなプログラムよりも、長い間実行されるサーバー・タイプのアプリケーションに適していることになります。また、コードの実行時組み立てという柔軟性を最もよく活用できるのがサーバー・アプリケーションですので、この種の開発で Java プラットフォームの利用が増加しているのも不思議ではありません。

このシリーズの第 2 回目では、Java プラットフォームの動的な性質の土台となっているもう一つの側面であるリフレクション APi を紹介したいと思います。リフレクションを利用することで、

実行コードから、内部的なクラス情報をアクセスできるようになります。クラス同士をソース・コードでリンクすることなく、実行時にまとめてフックすることのできる柔軟なコードを構築したい場合、リフレクションが優れた働きをします。しかしながら、ほとんどのツールと同様、どんなときに、どんな方法でそれを最も上手に活用できるのかを知っておく必要があります。Java プログラミングのダイナミックス の第 2 回では効果的なリフレクションの技とそのトレードオフを紹介します。またお会いしましょう。

---

# 著者について

Dennis Sosnoski

Dennis Sosnoski はシアトル地域にある Java 技術のコンサルティング会社、Sosnoski Software Solutions, Inc. の創立者で、主席コンサルタントでもあり、また XML や Web サービスに関するトレーニングやコンサルティングの専門家でもあります。彼のプロとしてのソフトウェア開発経験は 30年以上に渡り、ここ数年はサーバー側の XML 技術や Java 技術に注力しています。Dennis は、全米各地で行われる会議で頻繁に講演を行っています。また、Java クラスワーキング技術を基に構築された、オープンソースの JiBX XML Data Binding フレームワークの中心開発者でもあります。