javax.tools を使って動的アプリケーションを作成する

javax.tools.JavaCompiler の概要と動的アプリケーションのビルドへの適用

David J. Biesack (David.Biesack@sas.com)
Principal Systems Developer
SAS Institute, Inc.

2007年 12月 11日

最近のアプリケーションの多くには、アプリケーションの静的機能を拡張した難解な計算をユーザーが提供できるといった、動的機能が必要になってきていますが、このような目標を達成するとっておきの手段となるのが、Java SE (Java™ Platform, Standard Edition) 6 に Java ソースのコンパイル用標準 API として追加されたパッケージ、`javax.tools` です。この記事では、このパッケージに含まれる主要なクラスについて概説します。さらに、これらのクラスを使用してファイルではなく Java `String` から Java ソースをコンパイルするためのファサードを作成する方法を説明し、このファサードを使って対話型のグラフ描画アプリケーションをビルドします。

はじめに

Java ソースのコンパイル用標準 API として Java SE 6 に追加された `javax.tools` パッケージにより、動的機能を追加して静的アプリケーションを拡張することが可能になります。この記事では、このパッケージに含まれる主要なクラスの概要を説明し、これらのクラスを使ってファイルの代わりに Java `String`、`StringBuffer`、あるいはその他の `CharSequence` から Java ソースをコンパイルするためのファサードを作成する方法を説明します。さらにこのファサードを使用して、ユーザーが有効なあらゆる Java 数値表現により数値関数 y = f(x) を表すことのできる対話型のグラフ描画アプリケーションをビルドします。そして最後に、ソースの動的コンパイルに伴い起こり得るセキュリティー上のリスクとその対処方法についても説明します。

Java 拡張機能をコンパイルしてロードするという手段でアプリケーションを拡張するのは目新しい考えではありません。このような機能をサポートするフレームワークはすでに存在しており、Java クラスを生成してコンパイルする動的フレームワークとしては、Java EE (Java Platform, Enterprise Edition) の JSP (JavaServer Pages) 技術が広く知れ渡っています。JSP 変換機能では .jsp ファイルを Java サーブレットに変換するために中間ソース・コード・ファイルを使用し、このファイルを JSP エンジンがコンパイルして Java EE サーブレット・コンテナーにロードします。コンパイルを実行するには、`javac` コンパイラーを直接呼び出すか (この場合、JDK (Java Development Kit) がインストールされている必要があります)、あるいは `com.sun.tools.javac.Main` (Sun の tools.jar にあります) を呼び出すという方法がよく取られま

す。Sun の使用許諾では、tools.jar を完全な JRE (Java Runtime Environment) とともに再配布することが許可されています。また、このような動的機能を実装する別の手段として、既存の動的スクリプト言語 (JavaScript や Groovy など) でアプリケーションの実装言語を統合する場合や (「参考文献」を参照)、ドメイン固有の言語とそれに対応する言語インタープリターまたはコンパイラーを作成する場合もあります。

開発者が Java 言語で直接コーディングした拡張機能を使用できるフレームワークもありますが (NetBeans、Eclipse など)、このようなシステムでは外部で静的にコンパイルし、Java コードとその成果物のソースとバイナリーを管理しなければならなくなります。一方、Apache Commons の JCI はJava クラスを実動アプリケーションにコンパイルしてロードするメカニズムを提供します。Janino と Javassist も同様の動的機能を提供しますが、Janino は Java 1.4 以前の言語構文に限られ、Javassist はソース・コード・レベルではなく Java クラスの抽象レベルで機能します (これらのプロジェクトへのリンクは「参考文献」を参照)。ただし、Java 開発者であれば既に Java 言語で作成することに熟練しているはずなので、Java ソース・コードをその場で簡単に生成してコンパイルおよびロードできるというシステムは最短の学習曲線と最大の柔軟性を約束します。

`javax.tools` を使用するメリット

`javax.tools` を使うと次の利点があります。

- javax.tools は、Java Community Process に従って (JSR 199 として) 開発された標準 API で、Java SE 拡張として承認されています。一方、com.sun.tools.javac.Main API は厳密に言うと、文書化された Java プラットフォーム API の一部ではありません。そのため、必ずしも他のベンダーの JDK で利用できるわけではなく、Sun JDK の今後のリリースで同じ API が引き継がれると保証されているわけでもありません。
- バイト・コードではなく、慣れ親しんだ Java ソースを使用できます。有効な Java ソースを生成することによって正しい Java クラスを作成できるので、有効なバイト・コードに関する難解なルールや、クラス、メソッド、ステートメント、式などに関する新しいオブジェクト・モデルを学ばなくても済みます。
- javax.tools では、コードを生成およびロードするメカニズムを単純化し、この 1 つのサポート・メカニズムを標準としているため、必ずしもファイル・ベースのソースを使用する必要はありません。
- javax.tools は現時点だけでなく、今後もさまざまなベンダーの JDK Version 6 以降の実装に移植できます。
- javax.tools では、Java コンパイラーの検証済みバージョンを使用しています。
- インタープリター・ベースのシステムとは異なり、ロードしたクラスには JRE のランタイム最適化がすべて生かされます。

Java コンパイル: 概念と実装

`javax.tools` パッケージを理解するには、Java コンパイルの概念と、`javax.tools` パッケージがJava コンパイルの概念をどのように実装しているかを見直しておくと参考になります。`javax.tools` パッケージは Java コンパイルの概念のすべてを一般的な方法で抽象化し、ファイル・システムにソースを配置することなく代替のソース・オブジェクトからソース・コードを提供できるようにします。

Java ソースをコンパイルするには、以下のコンポーネントが必要です。

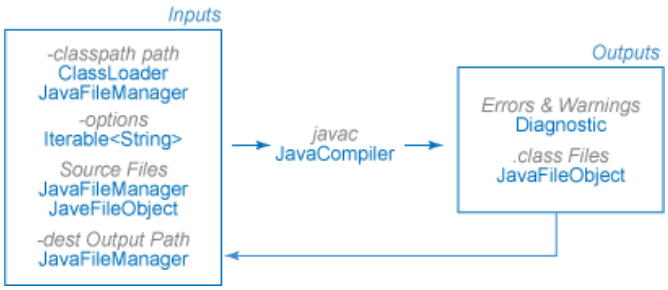
- **クラスパス。**コンパイラーはこのクラスパスからライブラリー・クラスを解決します。コンパイラーのクラスパスを構成するのは通常、ファイル・システム・ディレクトリーと以前にコンパイルされた .class ファイルが含まれるアーカイブ・ファイル (JAR または ZIP ファイル) の番号付きリストです。クラスパスを実装する `JavaFileManager` は、複数のソースと `JavaFileObject` クラスのインスタンス、そして `JavaFileManager` コンストラクターに渡される `ClassLoader` を管理します。`JavaFileObject` は、コンパイラーに役立つ以下の `JavaFileObject.Kind` 列挙型のいずれかを持つ特殊な `FileObject` です。
  - `SOURCE`
  - `CLASS`
  - `HTML`
  - `OTHER`

  各ソース・ファイルが、そのソースにアクセスするための `openInputStream()` メソッドを `InputStream` として提供します。
- **`javac` オプション。**`Iterable<String>` として渡されます。
- **ソース・ファイル。**コンパイルする 1 つ以上の .java ソース・ファイルです。ソース・ファイルと出力ファイルの名前を `JavaFileObject` インスタンスにマッピングする抽象ファイル・システムは `JavaFileManager` が提供します (この場合のファイルとは、ファイルに固有の名前とバイト・シーケンスとの関連付けのことを意味しています。クライアントは実際のファイル・システムを使用する必要はありません)。この記事のサンプルでは、`JavaFileManager` がクラス名とコンパイル対象の Java ソースが含まれる `CharSequence` インスタンスとの間のマッピングを管理します。`JavaFileManager.Location` には、ファイル名、そしてそのファイルの場所 (ソース・ファイルがある場所なのか、出力ファイルがある場所なのか) を示すフラグが含まれます。クラスパスとソース・パスが JAR とディレクトリーを結び付けるのと同じように、`ForwardingJavaFileManager` が `Chain of Responsibility` パターン (「参考文献」を参照) を実装して複数のファイル・マネージャーを結び付けられるようにします。チェーンの最初の要素に Java クラスがない場合、検索はチェーンの残りの項目に委任されます。
- **出力ディレクトリー。**コンパイラーが生成した .class ファイルを書き込むディレクトリーです。出力クラス・ファイルのコレクションのように機能する `JavaFileManager` は、コンパイル済み `CLASS` ファイルを表す `JavaFileObject` インスタンスも保存します。
- **コンパイラー本体。**`JavaCompiler` は `JavaFileManager` に含まれる `JavaFileObject SOURCE` オブジェクトからソースをコンパイルする `JavaCompiler.CompilationTask` オブジェクトを作成し、新しい出力 `JavaFileObject CLASS` ファイルと `Diagnostic` (警告およびエラー) を作成します。このコンパイラーのインスタンスは、静的 `ToolProvider.getSystemJavaCompiler()` メソッドによって返されます。
- **コンパイラーの警告およびエラー。**`Diagnostic` と `DiagnosticListener` と一緒に実装されます。Diagnostic はコンパイラーが出力する単一の警告またはコンパイル・エラーです。`Diagnostic` は以下を指定します。
  - `Kind` (`ERROR`、`WARNING`、`MANDATORY_WARNING`、`NOTE`、または `OTHER`)
  - ソースの該当する箇所 (行番号および列番号を含む)
  - メッセージ

  クライアントがコンパイラーに提供する `DiagnosticListener` によって、コンパイラーは診断結果をクライアントに返します。`DiagnosticCollector` は単純な `DiagnosticListener` 実装です。

図 1 に、`javac` の概念とそれに対応する `javax.tools` での実装を示します。

図 1. `javac` の概念と `javax.tools` インターフェースとの対応



以上の概念を基に、ここからは `CharSequence` をコンパイルするためのファサードの実装方法を説明します。

`CharSequence` インスタンスの Java ソースをコンパイルする方法

このセクションでは、`javax.tools.JavaCompiler` のファサードを構成していきます。`javaxtools.compiler.CharSequenceCompiler` クラス (「ダウンロード」を参照) は、任意の `java.lang.CharSequence` オブジェクト (`String`、`StringBuffer`、`StringBuilder` など) に保存された Java ソースをコンパイルして `Class` を返すことができます。`CharSequenceCompiler` には以下の API があります。

- **`public CharSequenceCompiler(ClassLoader loader, Iterable<String> options)`** : このコンストラクターは Java コンパイラーに渡された ClassLoader を受け入れ、Java コンパイラーが依存クラスを解決できるようにします。Iterable オプションにより、クライアントが javac のオプションに相当するコンパイラー・オプションを追加で渡すことが可能になります。
- **`public Map<String, Class<T>> compile(Map<String, CharSequence> classes, final DiagnosticCollector<JavaFileObject> diagnostics) throws CharSequenceCompilerException, ClassCastException`** : これは、複数ソースの同時コンパイルをサポートする汎用コンパイル・メソッドです。Java コンパイラーは、A.java は B.java に依存し、B.java は C.java に依存し、C.java は A.java に依存するといったクラスの循環グラフを処理しなければならないことに注意してください。このメソッドへの最初の引数は Map で、そのキーは完全修飾クラス名、そしてキーの値はそのクラスのソースが含まれる CharSequence です。以下は、その一例です。
  - `"mypackage.A"` ⇒ `"package mypackage; public class A { ... }";`
  - `"mypackage.B"` ⇒ `"package mypackage; class B extends A implements C { ... }";`
  - `"mypackage.C"` ⇒ `"package mypackage; interface C { ... }"`

  コンパイラーは `Diagnostic` を `DiagnosticCollector` に追加します。ジェネリック型パラメーター `T` は、キャスト後のクラスの型として望まれる基本型です。`compile()` は、単一のクラス名とコンパイル対象の `CharSequence` を引数として取る別のメソッドにオーバーロードされます。
- **`public ClassLoader getClassLoader()`** : このメソッドはコンパイラーが .class ファイルを生成するときに作成したクラス・ローダーを返します。このクラス・ローダーから、他のクラスやリソースをロードすることができます。

- **`public Class<T> loadClass(final String qualifiedClassName) throws ClassNotFoundException`** : このメソッドは、`compile()` メソッドによって定義可能な複数の補助クラス (ネストされた public クラスを含む) をロードできるようにします。

この `CharSequenceCompiler API` をサポートするため、2 つのクラスを持つ `javax.tools` インターフェースを実装します。一方のクラスは `CharSequence` ソースとコンパイラーの CLASS 出力を格納する `JavaFileObjectImpl`、そしてもう一方は、ソース・ファイルと出力ファイルの名前を `JavaFileObjectImpl` インスタンスにマッピングしてソース・シーケンスとコンパイラーが出力するバイト・コードの両方を管理する `JavaFileManagerImpl` です。

`JavaFileObjectImpl`

JavaFileObjectImpl (リスト 1 を参照) は JavaFileObject を実装し、CharSequence source (SOURCE の場合) または ByteArrayOutputStream byteCode (CLASS ファイルの場合) を保持します。主要なメソッドは CharSequence getCharContent(final boolean ignoreEncodingErrors) です。このメソッドによって、コンパイラーはソース・テキストを取得します。記載するすべてのサンプル・コードの完全なソースについては、「ダウンロード」を参照してください。

リスト 1. `JavaFileObjectImpl` (ソース・リストからの抜粋)

```
final class JavaFileObjectImpl extends SimpleJavaFileObject {
   private final CharSequence source;

   JavaFileObjectImpl(final String baseName, final CharSequence source) {
      super(CharSequenceCompiler.toURI(baseName + ".java"), Kind.SOURCE);
      this.source = source;
   }
   @Override
   public CharSequence getCharContent(final boolean ignoreEncodingErrors)
         throws UnsupportedOperationException {
      if (source == null)
         throw new UnsupportedOperationException("getCharContent()");
      return source;
   }
}
```

`FileManagerImpl`

`FileManagerImpl` (リスト 2 を参照) は `ForwardingJavaFileManager` を継承して、修飾クラス名を `JavaFileObjectImpl` インスタンスにマッピングします。

リスト 2. FileManagerImpl (ソース・リストからの抜粋)

```
final class FileManagerImpl extends ForwardingJavaFileManager<JavaFileManager> {
   private final ClassLoaderImpl classLoader;
   private final Map<URI, JavaFileObject> fileObjects
           = new HashMap<URI, JavaFileObject>();

   public FileManagerImpl(JavaFileManager fileManager, ClassLoaderImpl classLoader) {
      super(fileManager);
      this.classLoader = classLoader;
   }

   @Override
   public FileObject getFileForInput(Location location, String packageName,
         String relativeName) throws IOException {
      FileObject o = fileObjects.get(uri(location, packageName, relativeName));
      if (o != null)
         return o;
```

```
      return super.getFileForInput(location, packageName, relativeName);
   }

   public void putFileForInput(StandardLocation location, String packageName,
         String relativeName, JavaFileObject file) {
      fileObjects.put(uri(location, packageName, relativeName), file);
   }
}
```

CharSequenceCompiler

ToolProvider.getSystemJavaCompiler() が JavaCompiler を作成できない場合

ToolProvider.getSystemJavaCompiler() メソッドでは、tools.jar がアプリケーションのクラスパスに含まれていない場合に null を返すことが可能です。CharStringCompiler クラスは、考えられる構成上の問題を検出し、その問題についての修正案を伴った例外をスローします。Sun の使用許諾では、tools.jar を JRE とともに再配布することが許可されていることに注意してください。

以上のサポート・クラスを使って、これから早速、CharSequenceCompiler を定義します。このコンパイラーを構成するにはランタイム ClassLoader とコンパイラー·オプションを使用します。コンパイラーは ToolProvider.getSystemJavaCompiler() を使って JavaCompiler インスタンスを取得し、コンパイラーの標準ファイル・マネージャーに転送する JavaFileManagerImpl をインスタンス化します。

compile() メソッドは入力マップを繰り返し処理しながら、それぞれの名前/CharSequence から JavaFileObjectImpl を構成して JavaFileManager に追加し、JavaCompiler がファイル・マネージャーの getFileForInput() メソッドを呼び出す際にこの JavaFileObjectImpl を見つけられるようにします。続いて compile() メソッドは JavaCompiler.Task インスタンスを作成して実行します。実行が失敗した場合は CharSequenceCompilerException がスローされます。次に compile() メソッドに渡されるソースごとにコンパイルの結果、作成されるClass がロードされ、結果 Map に配置されます。

CharSequenceCompiler (リスト 3 を参照) に関連付けられるクラス・ローダーは ClassLoaderImpl インスタンスです。このインスタンスは JavaFileManagerImpl インスタンス内でクラスのバイト・コードを検索し、コンパイラーによって作成された .class ファイルを返します。

リスト 3. CharSequenceCompiler (ソース・リストからの抜粋)

```
public class CharSequenceCompiler<T> {
   private final ClassLoaderImpl classLoader;
   private final JavaCompiler compiler;
   private final List<String> options;
   private DiagnosticCollector<JavaFileObject> diagnostics;
   private final FileManagerImpl javaFileManager;

   public CharSequenceCompiler(ClassLoader loader, Iterable<String> options) {
      compiler = ToolProvider.getSystemJavaCompiler();
      if (compiler == null) {
         throw new IllegalStateException(
               "Cannot find the system Java compiler. "
               + "Check that your class path includes tools.jar");
      }
      classLoader = new ClassLoaderImpl(loader);
      diagnostics = new DiagnosticCollector<JavaFileObject>();
```

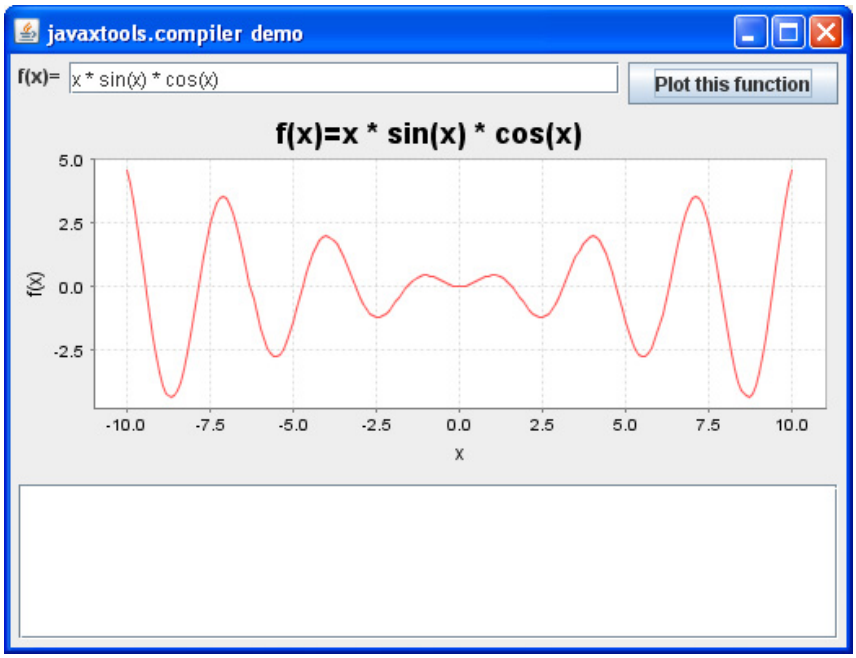
```
      final JavaFileManager fileManager = compiler.getStandardFileManager(diagnostics,
            null, null);
      javaFileManager = new FileManagerImpl(fileManager, classLoader);
      this.options = new ArrayList<String>();
      if (options != null) {
         for (String option : options) {
            this.options.add(option);
         }
      }
   }

   public synchronized Map<String, Class<T>>
       compile(final Map<String, CharSequence> classes,
                  final DiagnosticCollector<JavaFileObject> diagnosticsList)
           throws CharSequenceCompilerException, ClassCastException {
      List<JavaFileObject> sources = new ArrayList<JavaFileObject>();
      for (Entry<String, CharSequence> entry : classes.entrySet()) {
         String qualifiedClassName = entry.getKey();
         CharSequence javaSource = entry.getValue();
         if (javaSource != null) {
            final int dotPos = qualifiedClassName.lastIndexOf('.');
            final String className = dotPos == -1
               ? qualifiedClassName
                  : qualifiedClassName.substring(dotPos + 1);
            final String packageName = dotPos == -1
               ? ""
                  : qualifiedClassName .substring(0, dotPos);
            final JavaFileObjectImpl source =
               new JavaFileObjectImpl(className, javaSource);
            sources.add(source);
            javaFileManager.putFileForInput(StandardLocation.SOURCE_PATH, packageName,
                  className + ".java", source);
         }
      }
      final CompilationTask task = compiler.getTask(null, javaFileManager, diagnostics,
                                                    options, null, sources);
      final Boolean result = task.call();
      if (result == null || !result.booleanValue()) {
         throw new CharSequenceCompilerException("Compilation failed.",
                                                 classes.keySet(), diagnostics);
      }
      try {
         Map<String, Class<T>> compiled =
                     new HashMap<String, Class<T>>();
         for (Entry<String, CharSequence> entry : classes.entrySet()) {
            String qualifiedClassName = entry.getKey();
            final Class<T> newClass = loadClass(qualifiedClassName);
            compiled.put(qualifiedClassName, newClass);
         }
         return compiled;
      } catch (ClassNotFoundException e) {
         throw new CharSequenceCompilerException(classes.keySet(), e, diagnostics);
      } catch (IllegalArgumentException e) {
         throw new CharSequenceCompilerException(classes.keySet(), e, diagnostics);
      } catch (SecurityException e) {
         throw new CharSequenceCompilerException(classes.keySet(), e, diagnostics);
      }
   }
}
```

Plotter (グラフ描画) アプリケーション

これでソースをコンパイルするための単純な API が用意できたので、今度は関数からグラフを描画するアプリケーションを Swing で作成して、この API を動作させてみます。図 2 は、このアプリケーションによって描画された `x * sin(x) * cos(x)` 関数のグラフです。

図 2. javaxtools.compiler パッケージを使用した動的アプリケーション



このアプリケーションは、リスト 4 に定義する `Function` インターフェースを使用します。

リスト 4. Function インターフェース

```
package javaxtools.compiler.examples.plotter;
public interface Function {
   double f(double x);
}
```

このアプリケーションには、ユーザーが Java の式を入力できるテキスト・フィールドがあります。このフィールドに式を入力すると、暗黙的に宣言された `double x` 入力パラメーターに基づき、`double` 型の値が返されます。アプリケーションはこの式のテキストを、リスト 5 に示すコード・テンプレートの `$expression` でマークした場所に挿入します。また、アプリケーションは毎回一意に決まるクラス名を生成し、コード・テンプレート内の `$className` を置き換えます。パッケージ名も同じくテンプレート変数です。

リスト 5. Function テンプレート

```
package $packageName;
import static java.lang.Math.*;
public class $className
             implements javaxtools.compiler.examples.plotter.Function {
  public double f(double x) {
    return ($expression) ;
  }
}
```

アプリケーションは `String` オブジェクトを返す `fillTemplate(packageName, className, expr)` を使ってテンプレートに入力し、それから `CharSequenceCompiler` を使ってコンパイルを行います。例外またはコンパイラーの診断結果は `log()` メソッドに渡されるか、あるいはアプリケーションのスクロール可能な `errors` コンポーネントに直接書き込まれます。

リスト 6 に記載する `newFunction()` メソッドが、`Function` インターフェースを実装するオブジェクトを返します (リスト 5 のソース・テンプレートを参照)。

リスト 6. `Plotter` の `Function newFunction(String expr)` メソッド

```
Function newFunction(final String expr) {
   errors.setText("");
   try {
      // generate semi-secure unique package and class names
      final String packageName = PACKAGE_NAME + digits();
      final String className = "Fx_" + (classNameSuffix++) + digits();
      final String qName = packageName + '.' + className;
      // generate the source class as String
      final String source = fillTemplate(packageName, className, expr);
      // compile the generated Java source
      final DiagnosticCollector<JavaFileObject> errs =
            new DiagnosticCollector<JavaFileObject>();
      Class<Function> compiledFunction = stringCompiler.compile(qName, source, errs,
            new Class<?>[] { Function.class });
      log(errs);
      return compiledFunction.newInstance();
   } catch (CharSequenceCompilerException e) {
      log(e.getDiagnostics());
   } catch (InstantiationException e) {
      errors.setText(e.getMessage());
   } catch (IllegalAccessException e) {
      errors.setText(e.getMessage());
   } catch (IOException e) {
      errors.setText(e.getMessage());
   }
   return NULL_FUNCTION;
}
```

通常は、既知の基本クラスを継承するソース・クラスを生成するか、特定のインターフェースを実装するソース・クラスを生成することになるため、インスタンスを既知の型にキャストし、そのインスタンスのメソッドをタイプセーフな API を使って呼び出すことができます。注意する点として、`Function` クラスは、`CharSequenceCompiler<T>` をインスタンス化する際にジェネリック型パラメーター `T` として使用されます。そのため `compiledFunction` を同様に `Class<Function>` として型定義することができ、キャストを行わなくても `compiledFunction.newInstance()` によって `Function` インスタンスを返すことができます。

アプリケーションは `Function` インスタンスを動的に生成すると、そのインスタンスを使って x 値の範囲に対応する y 値を生成し、オープンソースの JFreeChart API (「参考文献」を参照) を使用して (x,y) 値からグラフを描画します。この Swing アプリケーションの完全なソースは、`javaxtools.compiler.examples.plotter` パッケージに含まれるダウンロード可能なソースとして入手することができます。

このアプリケーションではソース・コード生成の要件がかなり抑えられています。他のアプリケーションでは、Apache Velocity (「参考文献」を参照) など、これよりも高度なソース・テンプレート機能を利用することになります。

セキュリティー・リスクとその対策

ユーザーが任意の Java ソース・コードを入力できるというアプリケーションには、必然的にセキュリティー・リスクを伴います。SQL インジェクション (「参考文献」を参照) と同じように、

ユーザーやその他のエージェントがコードを生成するために直接 Java ソースを入力できるシステムは、不正に利用される可能性が考えられるからです。例えば、ここに記載した `Plotter` アプリケーションの場合、有効な Java の式には、システム・リソースにアクセスしたり、DoS 攻撃をするためのスレッドを生成したり、その他のセキュリティー上の弱点を突いた攻撃を実行したりするなどの、匿名のネストされたクラスが含まれている可能性があります。このような不正利用は、いわゆる Java インジェクションです。したがってこの種のアプリケーションは、信用できないユーザーがアクセスできるような非セキュアなロケーション (サーブレットまたはアプレットとしての Java EE サーバーなど) にデプロイしてはいけません。代わりに `javax.tools` の大半のクライアントがユーザー入力を制限し、ユーザーの要求をセキュア・ソース・コードに変換するようにしてください。

以下は、このパッケージを使用する上でのセキュリティー保護対策です。

- カスタムの `SecurityManager` または `ClassLoader` を使用して、匿名クラスや直接管理できないクラスがロードされないようにします。
- ソース・コード・スキャナーやその他のプリプロセッサーを使用して、疑わしいコード構文を使用している入力を破棄します。例えば Plotter の場合には `java.io.StreamTokenizer` を使って { (左中括弧) の文字が含まれる入力を破棄することによって、匿名クラスやネストされたクラスの宣言を効果的に防ぐことができます。
- `JavaFileManager` は `javax.tools API` を使用して、予期せず作成された CLASS ファイルを破棄することができます。例えば特定のクラスをコンパイルする際に、`JavaFileManager` は予期しないクラス・ファイルを格納するためのその他すべての呼び出しに対して `SecurityExeception` をスローし、ユーザーが推測または真似できないパッケージ名とクラス名のみが生成されるようにできます。これは、`Plotter` の `newFunction`メソッドで採用している対策です。

まとめ

この記事では、`javax.tools` パッケージの概念と重要なインターフェースについて説明し、`String` やその他の `CharSequence` に格納された Java をコンパイルするためのファサードを作成する方法を紹介した後、そのライブラリー・クラスを使って任意の f(x) 関数のグラフを描画するサンプル・アプリケーションを開発しました。この手法を極めて効果的に使用する方法は、以下をはじめ、他にも数多くあります。

- データ記述言語からバイナリー・ファイルのリーダー/ライターを生成する。
- JAXB (Java Architecture for XML Binding) やパーシスタンス・フレームワークのようなフォーマット変換機構を生成する。
- JSP の場合と同じく、ソースを Java 言語に変換してから Java ソースのコンパイルとロードを行うことにより、ドメイン固有の言語インタープリターを実装する。
- ルール・エンジンを実装する。
- その他、思い浮かんだアイデアすべて。

今度、アプリケーションを開発する際に動的な振る舞いが必要になったら、是非、javax.tools が実現する多様性と柔軟性を検討してみてください。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-jcomp.zip	166KB

著者について

David J. Biesack

David Biesack は SAS Institute, Inc. の Advanced Computing Lab で、Principal Systems Developer として高度な分析および分散コンピューティングに取り組んでいます。SAS での 19 年の経歴のうち、12 年間は Java 言語の設計およびコーディングに従事してきました。彼は Java 5 に新しい構文フィーチャーを追加した JSR 201 にも関与しました。