Advanced

Janino as a Source Code ClassLoader

The JavaSourceClassLoader extends Java™'s java.lang.ClassLoader class with the ability to load classes directly from source code.

To be precise, if a class is loaded through this ClassLoader, it searches for a matching ".java" file in any of the directories specified by a given "source path", reads, scans, parses and compiles it and defines the resulting classes in the JVM. As necessary, more classes are loaded through the parent class loader and/or through the source path. No intermediate files are created in the file system.

Example:

srcdir/pkg1/A.java

```
package pkg1;

import pkg2.*;

public class A extends B {
}
```

srcdir/pkg2/B.java

```
package pkg2;

public class B implements Runnable {
    public void run() {
        System.out.println("HELLO");
    }
}
```

```
// Sample code that reads, scans, parses,
compiles and loads
// "A.java" and "B.java", then instantiates
an object of class
// "A" and invokes its "run()" method.
ClassLoader cl = new JavaSourceClassLoader(
    this.getClass().getClassLoader(),  //
parentClassLoader
    new File[] { new File("srcdir") }, //
optionalSourcePath
    (String) null,                     //
optionalCharacterEncoding
    DebuggingInformation.NONE          //
debuggingInformation
);

// Load class A from "srcdir/pkg1/A.java",
and also its superclass
// B from "srcdir/pkg2/B.java":
Object o =
cl.loadClass("pkg1.A").newInstance();

// Class "B" implements "Runnable", so we
can cast "o" to
// "Runnable".
((Runnable) o).run(); // Prints "HELLO" to
"System.out".
```

If the Java™ source is not available in files, but from some other storage (database, main memory, ...), you may specify a custom ResourceFinder instead of the directory-based source path.

If you have many source files and you want to reduce the compilation time, you may want to use the CachingJavaSourceClassLoader, which uses a cache provided by the application to store class files for repeated use.

A BASH shell script named "bin/janino" is provided that wraps the JavaSourceClassLoader in a JAVAC-like command line interface:

```
$ cat my/pkg/A.java
package my.pkg;

import java.util.*;

public class A {
    public static void main(String[] args) {
        B b = new B();
        b.meth1();
    }
}

class B {
    void meth1() {
        System.out.println("Hello there.");
    }
}
$ type janino
/usr/local/bin/janino
$ janino my.pkg.A
Hello there.
$
```

Janino as a Command-Line Java™ Compiler

The Compiler class mimics the behavior of SUN's javac tool. It compiles a set of "compilation units" (i.e. Java™ source files) into a set of class files.

Using the "-warn" option, Janino spits out some probably very interesting warnings which may help you to "clean up"

the source code.

The BASH script "bin/janinoc" implements a drop-in replacement for SUN's JAVAC utility:

```
$ janinoc -sourcepath src -d classes
src/com/acme/MyClass.java
$ janinoc -help
A drop-in replacement for the JAVAC
compiler, see the documentation for JAVAC
Usage:
  janinoc [ <option> ] ... <class-name> [
<argument> ] ...
Options:
  -sourcepath <dir-list>    Where to look
for source files
  -classpath <dir-list>     Where to look
for class files
  -cp <dir-list>            Synonym for
"-classpath"
  -extdirs <dir-list>       Where to look
for extension class files
  -bootclasspath <dir-list> Where to look
for boot class files
  -encoding <encoding>      Encoding of
source files, default is platform-dependent
  -verbose                  Report about
opening, parsing, compiling of files
  -g                        Generate all
debugging info
  -g:none                   Generate no
debugging info
  -g:{lines,vars,source}    Generate only
```

```
some debugging info
  -warn:<pattern-list>      Issue certain
warnings, examples:
    -warn:*                 Enables all
warnings
    -warn:IASF              Only warn
against implicit access to static fields
    -warn:*-IASF            Enables all
warnings, except those against implicit
                            access to static
fields
    -warn:*-IA*+IASF        Enables all
warnings, except those against implicit
                            accesses, but do
warn against implicit access to
                            static fields
  -rebuild                  Compile all
source files, even if the class files
                            seems up-to-date
  -n                        Print
subcommands to STDOUT instead of running
them
```

```
  (any valid command-line optipon for the
JAVA tool, see "java -help")
$
```

Janino as an ANT Compiler

You can plug JANINO into the ANT utility through the AntCompilerAdapter class. Just make sure that janino.jar is on the class path, then run ANT with the following command-line option:

```
-Dbuild.compiler=org.codehaus.janino.AntComp
ilerAdapter
```

Janino as a TOMCAT Compiler

If you want to use JANINO with TOMCAT, just copy the "janino.jar" file into TOMCAT's "common/lib" directory, and add the follwing init parameter section to the JSP servlet definition in TOMCAT's "conf/web.xml" file:

```
<init-param>
    <param-name>compiler</param-name>

<param-value>org.codehaus.janino.AntCompiler
Adapter</param-value>
</init-param>
```

Janino as a Code Analyser

Apart from compiling Java™ code, JANINO can be used for static code analysis: Based on the AST ("abstract syntax tree") produced by the parser, the Traverser walks through all nodes of the AST, and derived classes can do all kinds of analyses on them, e.g. count declarations:

```
$ java
org.codehaus.janino.samples.DeclarationCount
er DeclarationCounter.java
Class declarations:     1
Interface declarations: 0
Fields:                 4
Local variables:        4
$
```

This is the basis for all these neat code metrics and style checking.

Janino as a Code Manipulator

If, e.g., you want to read a Java™ compilation unit into memory, manipulate it, and then write it back to a file for compilation, then all you have to do is:

```
// Read the compilation unit from Reader "r"
into memory.
Java.CompilationUnit cu = new Parser(new
Scanner(fileName,
r)).parseCompilationUnit();

// Manipulate the AST in memory.
// ...

// Convert the AST back into text.
UnparseVisitor.unparse(cu, new
OutputStreamWriter(System.out));
```

The UnparseVisitor (http://dist.codehaus.org/janino/javadoc/org/codehaus/janino/UnparseVisitor.html#main%28java.lang.String\[\]%29) class demostrates how to do this.

Alternative Compiler Implementations

JANINO can be configured to use not its own Java™ compiler, but an alternative implementation. Alternative implementations must basically implement the interface ICompilerFactory. One such alternative implementation is based on the javax.tools API (available since JDK 1.6), and is shipped as part of the JANINO distribution: `commons-compiler-jdk.jar`.

Basically there are two ways to switch implementations:

- Use `org.codehaus.commons.compiler.jdk.ExpressionEvaluator` and consorts instead of `org.codehaus.janino.ExpressionEvaluator`; put `commons-compiler-jdk.jar` instead of `janino.jar` on your compile-time and runtime classpath. (`commons-compiler.jar` must **always** be on the classpath, because it contains the basic classes that **every** implementation requires.)
- Use `org.codehaus.commons.compiler.CompilerFactoryFactory.getDefaultFactory().newExpressionEvaluator()` and compile only against `commons-compiler.jar` (and **no** concrete implementation). At runtime, add **one** implementation (`janino.jar` or `commons-compiler-jdk.jar`) to the class path, and `getDefaultFactory()` will find it **at runtime**.