

Basic

Janino as an Expression Evaluator

Say you build an e-commerce system, which computes the shipping cost for the items that the user put into his/her shopping cart. Because you don't know the merchant's shipping cost model at implementation time, you could implement a set of shipping cost models that come to mind (flat charge, by weight, by number of items, ...) and select one of those at run-time.

In practice, you will most certainly find that the shipping cost models you implemented will rarely match what the merchant wants, so you must add custom models, which are merchant-specific. If the merchant's model changes later, you must change your code, re-compile and re-distribute your software.

Because this is so unflexible, the shipping cost expression should be specified at *run-time*, not at *compile-time*. This implies that the expression must be scanned, parsed and evaluated at run-time, which is why you need an expression evaluator.

A simple expression evaluator would parse an expression and create a "syntax tree". The expression "a + b * c", for example, would compile into a "Sum" object who's first operand is parameter "a" and who's second operand is a "Product" object who's operands are parameters "b" and "c". Such a syntax tree can be evaluated relatively quickly. However, the run-time performance is about a factor of 100 worse than that of "native" Java™ code executed directly by the JVM. This limits the use of such an expression evaluator to simple applications.

Also, you may want not only do simple arithmetics like "a + b * c % d", but take the concept further and have a real "scripting" language which adds flexibility to your application. Since you know the Java™ programming language already, you may want to have a syntax that is similar to that of the Java™ programming language.

All these considerations lead to compilation of Java™ code at run-time, like some engines (e.g. JSP engines) already do. However, compiling Java™ programs with SUN's JDK is a relatively resource-intensive process (disk access, CPU time, ...). This is where Janino comes into play... a light-weight, "embedded" Java™ compiler that compiles simple programs *in memory* into Java™ bytecode which executes within the JVM of the running program.

OK, now you are curious... this is how you use the [ExpressionEvaluator](#):

```

// Compile the expression once; relatively
slow.
ExpressionEvaluator ee = new
ExpressionEvaluator(
    "c > d ? c : d",           //
expression
    int.class,                 //
expressionType
    new String[] { "c", "d" }, //
parameterNames
    new Class[] { int.class, int.class } //
parameterTypes
);

// Evaluate it with varying parameter
values; very fast.
Integer res = (Integer) ee.evaluate(
    new Object[] {           //
parameterValues
        new Integer(10),
        new Integer(11),
    }
);
System.out.println("res = " + res);

```

Notice: If you pass a string literal as the expression, be sure to escape all Java™ special characters, especially backslashes.

Compilation of the expression takes 670 microseconds on my machine (2 GHz P4), evaluation 0.35 microseconds (approx. 2000 times faster than compilation).

There is a sample program "ExpressionDemo" that you can use to play around with the `ExpressionEvaluator`, or you can study [ExpressionDemo's source code](#) to learn about `ExpressionEvaluator`'s API:

```
$ java
org.codehaus.janino.samples.ExpressionDemo
-help
Usage:
    ExpressionDemo { <option> } <expression> {
<parameter-value> }
Compiles and evaluates the given expression
and prints its value.
Valid options are
    -et <expression-type>
    (default: any)
    -pn <comma-separated-parameter-names>
    (default: none)
    -pt <comma-separated-parameter-types>
    (default: none)
    -te
    <comma-separated-thrown-exception-types>
    (default: none)
    -di <comma-separated-default-imports>
    (default: none)
    -help
The number of parameter names, types and
values must be identical.
```

```
$ java
org.codehaus.janino.samples.ExpressionDemo \
> -pn "a,b" -pt "int,int" "a + b" 11 22
Result = 33
$
```

Janino as a Script Evaluator

Analogously to the expression evaluator, a [ScriptEvaluator API](#) exists that compiles and processes a Java™ "block", i.e. the body of a method. If a return value other than "void" is defined, then the block must return a value of that type. Example:

```
System.out.println("Hello world");
return true;
```

As for the expression compiler, there is a demo program "ScriptDemo" for you to play with the `ScriptEvaluator` API:

```
$ java  
org.codehaus.janino.samples.ScriptDemo -help
```

Usage:

```
ScriptDemo { <option> } <script> {  
<parameter-value> }
```

Valid options are

```
-rt <return-type>
```

(default: void)

```
-pn <comma-separated-parameter-names>
```

(default: none)

```
-pt <comma-separated-parameter-types>
```

(default: none)

```
-te
```

```
<comma-separated-thrown-exception-types>
```

(default: none)

```
-di <comma-separated-default-imports>
```

(default: none)

```
-help
```

The number of parameter names, types and values must be identical.

```
$ java
org.codehaus.janino.samples.ScriptDemo \
> -rt boolean \
> -pn a,b \
> -pt double,double \
'System.err.println("a + b = " + (a+b));
System.err.println("a - b = " + (a - b));
return true;' 31.765 12.539
a + b = 44.304
a - b = 19.226
Result = true
```

Check the [source code of ScriptDemo](#) to learn more about the `ScriptEvaluator` API.

Janino as a Class Body Evaluator

Analogously to the expression evaluator and the script evaluator, a [ClassBodyEvaluator](#) exists that compiles and processes the body of a Java™ class, i.e. a series of method and variable declarations. If you define a contract that the class body should define a method named "main()", then your script will look almost like a "C" program:

```
import java.util.*;

// Field declaration:
private static final String hello = "World";

// Method declaration:
public static void main(String[] args) {
    System.out.println(hello + args.length);
}
```

The "ClassBodyDemo" program ([source code](#)) demonstrates this:

```
$ java
org.codehaus.janino.samples.ClassBodyDemo
-help
```

Usage:

```
ClassBodyDemo <class-body> { <argument> }
ClassBodyDemo -help
```

If <class-body> starts with a '@', then the class body is read from the named file.

The <class-body> must declare a method "public static main(String[])" to which the <argument>s are passed. If the return type of that method is not VOID, then the returned value is printed to STDOUT.

```
$ java
org.codehaus.janino.samples.ClassBodyDemo \
> 'import java.util.*;
>
> // Field declaration:
> private static final String hello =
"World";
>
> // Method declaration:
> public static void main(String[] args) {
>     System.out.println(hello +
args.length);
> }' alpha beta gamma
World3
```

Janino as a Simple Compiler

The [SimpleCompiler](#) compiles a single "compilation unit" (a.k.a. ".java" file). Opposed to normal Java™ compilation, that compilation unit may define more than one public class. Example:

```
package my.pkg;

import java.util.*;

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

public class B {
    void meth1() {
        System.out.println("Hello there.");
    }
}
```

It returns a `ClassLoader` from which you can retrieve the classes that were compiled.