Groovy

Extending Java with scripting capabilities

Last updated: 10 July 2017

Contents

About Groovy

Groovy is completely optional in Java projects. It does not require rewriting of Java code. It mixes and matches perfectly with existing Java code. Groovy compiles straight to Java bytecode.

Java projects can contain anything from zero percent to one hundred percent Groovy.

Groovy seamlessly and transparently integrates and interoperates with Java and any third-party libraries.

Groovy is an agile and dynamic language that extends Java with scripting capabilities. It has a concise, readable and expressive syntax that is easy to learn for Java developers and provides powerful features, such as closures, DSL support, builders and dynamic typing. It builds upon the strengths of Java, but has additional power features inspired by languages like Python, Ruby and Smalltalk.

Groovy is great for writing concise and maintainable tests, and for all build and automation tasks.

Groovy is particularly useful for testing, as it allows faster writing of test cases with significantly less program code than traditional Java.

Install Groovy

1. Install the latest Java JDK. Don’t forget to set the “JAVA_HOME” environment variable.
2. The correct installation of Java can be verified from the command line with the command “`java -version`”.
3. Download and install Groovy from http://groovy-lang.org/download.html .
4. The correct installation of Groovy can be verified from the command line with the command “`groovy --version`”.

Execute Groovy code

Option number one: Execute directly from command line

Execute Groovy directly from the command line example (without a text file):

```
groovy -e "println 'Hello, World!'"
```

Option number two: Execute a program code (text) file

If the statement `println 'Hello, World!'` gets saved in a text file called “`HelloWorld.groovy`”, then it can be executed like this:

```
groovy HelloWorld.groovy
```

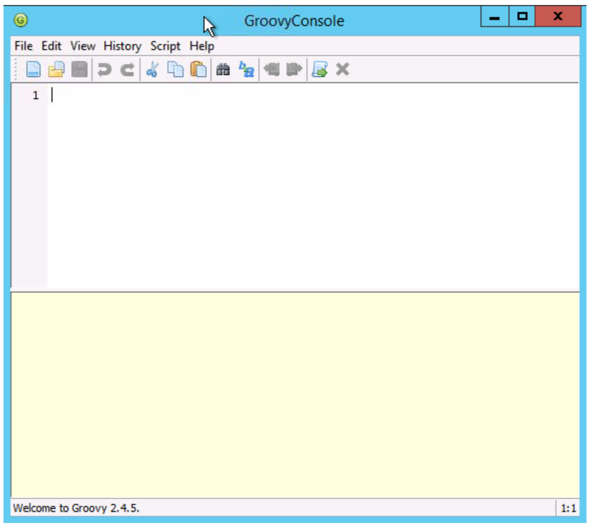
Option number three: Execute from the interactive programming environment (groovysh)

For example after entering “`groovysh`”:

```
groovy:000> helloClosure = { println "Hello $it" }
===> groovysh_evaluate$_run_closure1@7301061
groovy:000>counter = 1..5
===> [1, 2, 3, 4, 5]
groovy:000>counter.each helloClosure
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
===> [1, 2, 3, 4, 5]
```

PS: Autocomplete help is available by pressing the TAB key.

However, a much more convient GUI alternative to “`groovysh`” is the “`GroovyConsole`”:



Compiling Groovy code into Java classes

Compile a groovy script into Java classes:

`goovyc` *sample*`.groovy`

Run the compiled java classes:

`java -cp ".;%GROOVY_HOME%\lib\*"` *sample*

Dependency management with Grape

Grape stands for the “Groovy Adaptable (Advanced) Packaging Engine”, and it is a part of the Groovy installation. Grape helps you download and cache external dependencies from within a script with a set of simple annotations. It is mainly used for simple scripts that are generated without a development environment (such as Eclipse) and associated dependency management like Ant, Maven, or Gradle.

If a script requires an external dependency that is available in Maven Central Repository, then the “`@Grab`” annotation can be used to annotate an “`import`” (as in the following example), “`class`”, or “`method`” with a reference to that library. Groovy will automatically download it, cache it, and put it on the class path of the script.

The following program code is an example Groovy script that is dependent on the Apache Commons HttpClient library for making a Google search and saving a results page to a local html file:

```
@Grab('org.apache.httpcomponents:httpclient:4.5.1')
import org.apache.http.impl.client.DefaultHttpClient
import org.apache.http.client.methods.HttpGet
def httpClient = new DefaultHttpClient()
def url = 'https://www.google.com/search?q=Groovy'
def httpGet = new HttpGet(url)
def httpResponse = httpClient.execute(httpGet)
new File('result.html').text =httpResponse.entity.content.text
```

Groovy and Maven integration

This example uses the GMaven plugin to integrate Groovy and Maven. As an alternative to GMaven, there's also the Eclipse Groovy Compiler plugin for Maven that can be used instead of GMaven.

```
<dependencies>
	<dependency>
		<groupId>junit</groupId>
		<artifactId>junit</artifactId>
		<version>4.12</version>
		<scope>test</scope>
	</dependency>
	<dependency>
		<groupId>org.codehaus.gmaven.runtime</groupId>
		<artifactId>gmaven-runtime-2.0</artifactId>
		<version>1.5</version>
	</dependency>
</dependencies>
```

```
<build>
	<plugins>
		<plugin>
			<groupId>org.codehaus.gmaven</groupId>
			<artifactId>gmaven-plugin</artifactId>
			<version>1.5</version>
			<executions>
				<execution>
					<goals>
						<goal>compile</goal>
						<goal>testCompile</goal>
					</goals>
				</execution>
			</executions>
			<dependencies>
				<dependency>
					<groupId>org.codehaus.gmaven.runtime</groupId>
					<artifactId>gmaven-runtime-2.0</artifactId>
					<version>1.5</version>
```

```
                                </dependency>
                        </dependencies>
                </plugin>
        </plugins>
</build>
```

Groovy language features

Interpolation example

```
def city = 'London'
println "I am in $city"
```

Outputs: “I am in London”.

Groovy Beans (data objects)

Groovy Beans are much more compact and convenient than Java Beans. A Groovy Bean only has to declare the properties. Getters, setters and constructors are created at compilation time.

An example of a bean with 3 properties:

```
class Student {
        Long id
        String fistName
        String lastName
        List favouriteSports
}
```

In this example, data can be assigned like this:

```
def myStudent1 = new Student()
myStudent1.lastName = 'Smith'
myStudent1.favouriteSports = ['golf','tennis']
```

Alternatively, properties can also be set directly in any order. It is not required to set all the properties of the bean:

```
def myStudent2 = new Student(id:23, lastName:'Smith',
favouriteSports:['golf','tennis'])
```

Closures (defining code as data)

Closures are a lot like methods, but they act like objects (instances of the Closure class), so that they can be passed around.

Closures normally contain less code, have little or no repetition, and can be re-used easily.

A Closure always returns the value of the last statement in the body; the "`return`" keyword is not required.

The "`it`" keyword in a closure is simply a reference to the current element, for example:

```
['London','Paris','Tokyo'].each{println it}
```

Result Output: "London", "Paris", and "Tokyo".

Example:

```
def doubling = { arg1 -> println arg1 * 2 }
[1,3,5].each(doubling)
```

Result Output: "2", "6" and "10".

Testing with Groovy

Unit Testing with JUnit

Testing Java program code with Groovy makes the tests less verbose and it is easier for developers to clearly express the intent of each test method.

When Unit Tests are written in Groovy (saved in a file with the file extension "*ClassName*.groovy", not "*.java!"), the following rules and best practises apply:

1. As usual, Unit Tests should be saved in the "/src/test/…" folder structure.
2. Each test method name should start with "test" and the return type should be "void".

This example uses the Junit "@Test" annotation (JUnit version 4 and above).

"groovy.test.GroovyAssert" descends from "org.junit.Assert", which means that it inherits all JUnit assertion methods. However, with the introduction of the power assertion statement, it turned out to be good practice to rely on assertion statements, instead of using the JUnit assertion methods with the improved messages being the main reason

```
Import org.junit.Test
import static groovy.test.GroovyAssert.*

class myJavaTestClassName {

      @Test
      void testMyGroovyTestMethodName() {
            Groovy program code goes here…
            assert Result == Expectation
      }
}
```