What's new in Groovy 2.0?

Posted by **Guillaume Laforge** on Jun 28, 2012

Sections
Development
Topics
Groovy ,
JVM Languages ,
Dynamic Languages ,
Java ,
Languages ,
Programming

The newly released Groovy 2.0 brings key static features to the language with **static type checking** and **static compilation**, adopts JDK 7 related improvements with **Project Coin syntax enhancements** and the **support of the new "invoke dynamic"** JVM instruction, and becomes **more modular** than before. In this article, we're going to look into those new features in more detail.

A "static theme" for a dynamic language

Static type checking

Groovy, by nature, is and will always be a dynamic language. However, Groovy is often used as a "Java scripting language", or as a "better Java" (ie. a Java with less boilerplate and more power features). A lot of Java developers actually use and embed Groovy in their Java applications as an extension language, to author more expressive business rules, to further customize the application for different customers, etc. For such Java-oriented use cases, developers don't need all the dynamic capabilities offered by the language, and they usually expect the same kind of feedback from the Groovy compiler as the one given by javac. In particular, they want to get compilation errors (rather than runtime errors) for things like typos on variable or method names, incorrect type assignments and the like. That's why Groovy 2 features **static type checking support**.

Spotting obvious typos

The static type checker is built using Groovy's existing powerful AST (Abstract Syntax Tree) transformation mechanisms but for those not familiar with these mechanisms you can think of it as an optional compiler plugin triggered through an annotation. Being an optional feature, you are not forced to use it if you don't need it. To trigger static type checking, just use the `@TypeChecked` annotation on a method or on a class to turn on checking at your desired level of granularity. Let's see that in action with a first example:

```
import groovy.transform.TypeChecked

void someMethod() {}

@TypeChecked
void test() {
    // compilation error:
    // cannot find matching method sommeeMethod()
    sommeeMethod()

    def name = "Marion"

    // compilation error:
    // the variable naaammme is undeclared
    println naaammme
}
```

We annotated the `test()` method with the `@TypeChecked` annotation, which instructs the Groovy compiler to run the static type checking for that particular method at compilation time. We're trying to call `someMethod()` with some obvious typos, and to print the name variable again with another typo, and the compiler will throw two compilation errors because respectively, the method and variable are not found or declared.

Check your assignments and return values

The static type checker also verifies that the return types and values of your assignments are coherent:

```
import groovy.transform.TypeChecked

@TypeChecked
Date test() {
    // compilation error:
    // cannot assign value of Date
    // to variable of type int
    int object = new Date()

    String[] letters = ['a', 'b', 'c']
    // compilation error:
    // cannot assign value of type String
    // to variable of type Date
    Date aDateVariable = letters[0]

    // compilation error:
    // cannot return value of type String
    // on method returning type Date
    return "today"
}
```

In this example, the compiler will complain about the fact you cannot assign a `Date` in an `int` variable, nor can you return a `String` instead of a `Date` value specified in the method signature. The compilation error from the middle of the script is also interesting, as not only does it complain of the wrong assignment, but also because it shows type inference at play, because the type checker, of course, knows that `letters[0]` is of type `String`, because we're dealing with an array of `Strings`.

More on type inference

Since we're mentioning type inference, let's have a look at some other occurrences of it. We mentioned the type checker tracks the return types and values:

```
import groovy.transform.TypeChecked

@TypeChecked
int method() {
    if (true) {
        // compilation error:
        // cannot return value of type String
        // on method returning type int
        'String'
    } else {
        42
    }
}
```

Given a method returning a value of primitive type `int`, the type checker is able to also check the values returned from different constructs like `if / else` branches, `try / catch` blocks or `switch / case` blocks. Here, in our example, one branch of the `if / else` blocks tries to return a `String` value instead of a primitive `int`, and the compiler complains about it.

Common type conversions still allowed

The static type checker, however, won't complain for certain automatic type conversions that Groovy supports. For instance, for method signatures returning `String, boolean` or `Class`, Groovy converts return values to these types automatically:

```
import groovy.transform.TypeChecked

@TypeChecked
boolean booleanMethod() {
    "non empty strings are evaluated to true"
}

assert booleanMethod() == true

@TypeChecked
String stringMethod() {
    // StringBuilder converted to String calling
toString()
    new StringBuilder() << "non empty string"
}

assert stringMethod() instanceof String
```

```
@TypeChecked
Class classMethod() {
    // the java.util.List class will be returned
    "java.util.List"
}

assert classMethod() == List
```

The static type checker is also clever enough to do **type inference**:

```
import groovy.transform.TypeChecked

@TypeChecked
void method() {
    def name = " Guillaume "

    // String type inferred (even inside GString)
    println "NAME = ${name.toUpperCase()}"

    // Groovy GDK method support
    // (GDK operator overloading too)
    println name.trim()

    int[] numbers = [1, 2, 3]
    // Element n is an int
    for (int n in numbers) {
        println
    }
}
```

Although the `name` variable was defined with `def`, the type checker understands it is of type `String`. Then, when this variable is used in the interpolated string, it knows it can call `String's toUpperCase()` method, or the `trim()` method later one, which is a method added by the Groovy Development Kit decorating the `String` class. Last, when iterating over the elements of an array of primitive `ints`, it also understands that an element of that array is obviously an `int`.

Mixing dynamic features and statically typed methods

An important aspect to have in mind is that using the static type checking facility restricts what you are allowed to use in Groovy. Most runtime dynamic features are not allowed, as they can't be statically type checked at compilation time. So adding a new method at runtime through the type's metaclasses is not allowed. But when you need to use some particular dynamic feature, like Groovy's builders, you can opt out of static type checking should you wish to.

The `@TypeChecked` annotation can be put at the class level or at the method level. So if you want to have a whole class type checked, put the annotation on the class, and if you want only a few methods type checked, put the annotation on just those methods. Also, if you want to have everything type checked, except a specific method, you can annotate the latter with `@TypeChecked(TypeCheckingMode.SKIP)` - or `@TypeChecked(SKIP)` for short, if you statically import the associated enum. Let's illustrate the situation with the following script, where the `greeting()` method is type checked, whereas the `generateMarkup()` method is not:

```
import groovy.transform.TypeChecked
import groovy.xml.MarkupBuilder

// this method and its code are type checked
@TypeChecked
```

```
String greeting(String name) {
    generateMarkup(name.toUpperCase())
}

// this method isn't type checked
// and you can use dynamic features like the markup
builder
String generateMarkup(String name) {
    def sw =new StringWriter()
    new MarkupBuilder(sw).html {
        body {
            div name
        }
    }
    sw.toString()
}

assert greeting("Cédric").contains("<div>CÉDRIC</div>")
```

Type inference and instanceof checks

Current production releases of Java don't support general type inference; hence we find today many places where code is often quite verbose and cluttered with boilerplate constructs. This obscures the intent of the code and without the support of powerful IDEs is also harder to write. This is the case with `instanceof` checks: You often check the class of a value with instanceof inside an `if` condition, and afterwards in the `if` block, you must still use casts to be able to use methods of the value at hand. In plain Groovy, as well as in the new static type checking mode, you can completely get rid of those casts.

```
import groovy.transform.TypeChecked
import groovy.xml.MarkupBuilder

@TypeChecked
String test(Object val) {
    if (val instanceof String) {
        // unlike Java:
        // return ((String)val).toUpperCase()
        val.toUpperCase()
    } else if (val instanceof Number) {
        // unlike Java:
        // return ((Number)val).intValue().multiply(2)
        val.intValue() * 2
    }
}

assert test('abc') == 'ABC'
assert test(123) == '246'
```

In the above example, the static type checker knows that the val parameter is of type `String` inside the `if` block, and of type `Number` in the else if block, without requiring any cast.

Lowest Upper Bound

The static type checker goes a bit further in terms of type inference in the sense that it has a more granular understanding of the type of your objects. Consider the following code:

```
import groovy.transform.TypeChecked

// inferred return type:
```

```
// a list of numbers which are comparable and serializable
@TypeChecked test() {
    // an integer and a BigDecimal
    return [1234, 3.14]
}
```

In this example, we return, intuitively, a list of numbers: an `Integer` and a `BigDecimal`. But the static type checker computes what we call a *"lowest upper bound"*, which is actually a list of numbers which are also serializable and comparable. It's not possible to denote that type with the standard Java type notation, but if we had some kind of intersection operator like an ampersand, it could look like `List<Number & Serializable & Comparable>.`

Flow typing

Although this is not really recommended as a good practice, sometimes developers use the same untyped variable to store values of different types. Look at this method body:

```
import groovy.transform.TypeChecked

@TypeChecked test() {
    def var = 123                 // inferred type is int
    var = "123"                   // assign var with a String

    println var.toInteger()   // no problem, no need to
cast

    var = 123
    println var.toUpperCase() // error, var is int!
}
```

The `var` variable is initialized with an `int`. Then, a `String` is assigned. The *"flow typing"* algorithm follows the flow of assignment and understands that the variable now holds a `String`, so the static type checker will be happy with the `toInteger()` method added by Groovy on top of `String`. Next, a number is put back in the var variable, but then, when calling `toUpperCase()`, the type checker will throw a compilation error, as there's no `toUpperCase()` method on `Integer`.

There are some special cases for the flow typing algorithm when a variable is shared with a closure which are interesting. What happens when a local variable is referenced in a closure inside a method where that variable is defined? Let's have a look at this example:

```
import groovy.transform.TypeChecked

@TypeChecked test() {
    def var = "abc"
    def cl = {
        if (new Random().nextBoolean()) var = new Date()
    }
    cl()
    var.toUpperCase() // compilation error!
}
```

The `var` local variable is assigned a `String`, but then, `var` might be assigned a `Date` if some random value is true. Typically, it's only at runtime that we really know if the condition in the if statement of the closure is made or not. Hence, at compile-time, there's no chance the compiler can know if `var` now contains a `String` or a `Date`. That's why the compiler will actually complain about the `toUpperCase()` call, as it is not able to infer that the variable contains a `String` or not. This example is certainly a bit contrived, but there are some more interesting cases:

```
import groovy.transform.TypeChecked

class A           { void foo() {} }
class B extends A { void bar() {} }

@TypeChecked test() {
    def var = new A()
    def cl = { var = new B() }
    cl()
    // var is at least an instance of A
    // so we are allowed to call method foo()
    var.foo()
}
```

In the `test()` method above, `var` is assigned an instance of `A`, and then an instance of `B` in the closure which is call afterwards, so we can at least infer that var is of type `A`.

All those checks added to the Groovy compiler are done at compile-time, but the generated bytecode is still the same dynamic code as usual - no changes in behavior at all.

Since the compiler now knows a lot more about your program in terms of types, it opens up some interesting possibilities: what about compiling that type checked code statically? The obvious advantage will be that the generated bytecode will more closely resemble the bytecode created by the javac compiler itself, making statically compiled Groovy code as fast as plain Java, among other advantages. In the next section, we'll learn more about Groovy's static compilation.

Static compilation

As we shall see in the following chapter about the JDK 7 alignments, Groovy 2.0 supports the new *"invoke dynamic"* instruction of the JVM and its related APIs, facilitating the development of dynamic languages on the Java platform and bringing some additional performance to Groovy's dynamic calls. However, unfortunately shall I say, JDK 7 is not widely deployed in production at the time of this writing, so not everybody has the chance to run on the latest version. So developers looking for performance improvements would not see much changes in Groovy 2.0, if they aren't able to run on JDK 7. Luckily, the Groovy development team thought those developers could get interesting performance boost, among other advantages, by allowing type checked code to be compiled statically.

Without further ado, let's dive in and use the new `@CompileStatic` transform:

```
import groovy.transform.CompileStatic

@CompileStatic
int squarePlusOne(int num) {
    num * num + 1
}

assert squarePlusOne(3) == 10
```

This time, instead of using `@TypeChecked`, use `@CompileStatic`, and your code will be statically compiled, and the bytecode generated here will look like javac's bytecode, running just as fast. Like the `@TypeChecked annotation`, `@CompileStatic` can annotate classes and methods, and `@CompileStatic(SKIP)` can bypass static compilation for a specific method, when its class is marked with `@CompileStatic`.

Another advantage of the javac-like bytecode generation is that the size of the bytecode for those annotated methods will be smaller than the usual bytecode generated by Groovy for dynamic methods, since to support Groovy's dynamic features, the bytecode in the dynamic case contains additional instructions to call into Groovy's runtime system.

Last but not least, static compilation can be used by framework or library code writers to help avoid adverse interactions when dynamic metaprogramming is in use in several parts of the codebase. The dynamic features available in languages like Groovy are what give developers incredible power and flexibility but if care is not taken, different assumptions can exist in different parts of the system with regards to what metaprogramming features are in play and this can have unintended consequences. As a slightly contrived example, consider what happens if you are using two different libraries, both of which add a similarly named but differently implemented method to one of your core classes. What behaviour is expected? Experienced users of dynamic languages will have seen this problem before and probably heard it referred to as *"monkey patching"*. Being able to statically compile parts of your code base - those parts that don't need dynamic features - shields you from the effects of monkey patching, as the statically compiled code doesn't go through Groovy's dynamic runtime system. Although dynamic runtime aspects of the language are not allowed in a static compilation context, all the usual AST transformation mechanisms work just as well as before, since most AST transforms perform their magic at compilation time.

In terms of performance, Groovy's statically compiled code is usually more or less as fast as javac's. In the few micro-benchmarks the development team used, performance is identical in several cases, and sometimes it's slightly slower.

Historically, thanks to the transparent and seamless integration of Java and Groovy, we used to advise developers to optimize some hotspot routines in Java for further performance gains, but now, with this static compilation option, this is no longer the case, and people wishing to develop their projects in full Groovy can do so.

The Java 7 and JDK 7 theme

The grammar of the Groovy programming language actually derives from the Java grammar itself, but obviously, Groovy provides additional nice shortcuts to make developers more productive. This familiarity of syntax for Java developers has always been a key selling point for the project and its wide adoption, thanks to a flat learning curve. And of course, we expect Groovy users and newcomers to also want to benefit from the few syntax refinements offered by Java 7 with its *"Project Coin"* additions.

Beyond the syntax aspects, JDK 7 also brings interesting novelties to its APIs, and for a first time in a long time, even a new bytecode instruction called *"invoke dynamic"*, which is geared towards helping implementors develop their dynamic languages more easily and benefit from more performance.

Project Coin syntax enhancements

Since day 1 (that was back in 2003 already!) Groovy has had several syntax enhancements and features on top of Java. One can think of closures, for example, but also the ability to put more than just discrete values in `switch / case` statements, where Java 7 only allows `Strings` in addition. So some of the Project Coin syntax enhancements, like `Strings` in switch, were already present in Groovy. However, some of the enhancements are new, such as binary literals, underscore in number literals, or the multi catch block, and Groovy 2 supports them. The sole omission from the Project Coin enhancements is the "try with resources" construct, for which Groovy already provides various alternatives through the rich API of the Groovy Development Kit.

Binary literals

In Java 6 and before, as well as in Groovy, numbers could be represented in decimal, octal and hexadecimal bases, and with Java 7 and Groovy 2, you can use a binary notation with the "0b" prefix:

```
int x = 0b10101111
assert x == 175

byte aByte = 0b00100001
assert aByte == 33

int anInt = 0b1010000101000101
assert anInt == 41285
```

Underscore in number literals

When writing long literal numbers, it's harder on the eye to figure out how some numbers are grouped together, for example with groups of thousands, of words, etc. By allowing you to place underscore in number literals, it's easier to spot those groups:

```
long creditCardNumber = 1234_5678_9012_3456L
long socialSecurityNumbers = 999_99_9999L
double monetaryAmount = 12_345_132.12
long hexBytes = 0xFF_EC_DE_5E
long hexWords = 0xFFEC_DE5E
long maxLong = 0x7fff_ffff_ffff_ffffL
long alsoMaxLong = 9_223_372_036_854_775_807L
long bytes = 0b11010010_01101001_10010100_10010010
```

Multicatch block

When catching exceptions, we often replicate the catch block for two or more exceptions as we want to handle them in the same way. A workaround is either to factor out the commonalities in its own method, or in a more ugly fashion to have a catch-all approach by catching `Exception`, or worse, `Throwable`. With the multi catch block, we're able to define several exceptions to be catch and treated by the same catch block:

```
try {
    /* ... */
} catch(IOException | NullPointerException e) {
    /* one block to handle 2 exceptions */
}
```

Invoke Dynamic support

As we mentioned earlier in this article, JDK 7 came with a new bytecode instruction called *"invoke dynamic"*, as well as with its associated APIs. Their goal is to help dynamic language implementors in their job of crafting their languages on top of the Java platform, by simplifying the wiring of dynamic method calls, by defining *"call sites"* where dynamic method call section can be cached, *"method handles"* as method pointers, *"class values"* to store any kind of metadata along class objects, and a few other things. One caveat though, despite promising performance improvements, "invoke dynamic" hasn't yet fully been optimized inside the JVM, and doesn't yet always deliver the best performance possible, but update after update, the optimizations are coming.

Groovy brought its own implementation techniques, to speed up method selection and invocation with "call site caching", to store metaclasses (the dynamic runtime equivalent of classes) with its metaclass registry, to perform native primitive calculations as fast as Java, and much more. But with the advent of "invoke dynamic", we can rebase the implementation of Groovy on top of these APIs and this JVM bytecode instruction, to gain performance improvements and to simplify our code base.

If you're lucky to run on JDK 7, you'll be able to use a new version of the Groovy JARs which has been compiled with the "invoke dynamic" support. Those JARs are easily recognizable as they use the "-indy" classifier in their names.

Enabling invoke dynamic support

Using the "indy" JARs is not enough, however, to compile your Groovy code so that it leverages the "invoke dynamic" support. For that, you'll have to use the --indy flag when using the "groovyc" compiler or the "groovy" command. This also means that even if you're using the indy JARs, you can still target JDK 5 or 6 for compilation.

Similarly, if you're using the groovyc Ant task for compiling your projects, you can also specify the indy attribute:

```
...
<taskdef name="groovyc"
        classname="org.codehaus.groovy.ant.Groovyc"
        classpathref="cp"/>
...
<groovyc srcdir="${srcDir}" destdir="${destDir}"
indy="true">
    <classpath>
...
    </classpath>
```

```
</groovyc>
...
```

The Groovy Eclipse Maven compiler plugin hasn't yet been updated with the support of Groovy 2.0 but this will be the case shortly. For GMaven plugin users, although it's possible to configure the plugin to use Groovy 2.0 already, there's currently no flag to enable the invoke dynamic support. Again, GMaven will also be updated soon in that regard.

When integrating Groovy in your Java applications, with `GroovyShell`, for example, you can also enable the invoke dynamic support by passing a `CompilerConfiguration` instance to the `GroovyShell` constructor on which you access and set the optimization options:

```
CompilerConfiguration config = new CompilerConfigura-
tion();
config.getOptimizationOptions().put("indy", true);
config.getOptimizationOptions().put("int", false);
GroovyShell shell = new GroovyShell(config);
```

As invokedynamic is supposed to be a full replacement to dynamic method dispatch, it is also necessary to disable the primitive optimizations which generate extra bytecode that is here to optimize edge cases. Even if it is in some cases slower than with primitive optimizations activated, future versions of the JVM will feature an improved JIT which will be capable of inlining most of the calls and remove unnecessary boxings.

Promising performance improvements

In our testing, we noticed some interesting performance gains in some areas, whereas other programs could run slower than when not using the invoke dynamic support. The Groovy team has further performance improvements in the pipeline for Groovy 2.1 however, but we noticed the JVM isn't yet finely tuned and still has a long way to go to be fully optimized. But fortunately, upcoming JDK 7 updates (in particular update 8) should already contain such improvements, so the situation can only improve. Furthermore, as invoke dynamic is used for the implementation of JDK 8 Lambdas, we can be sure more improvements are forthcoming.

A more modular Groovy

We'll finish our journey through the new features of Groovy 2.0 by speaking about modularity. Just like Java, Groovy is not just a language, but it's also a set of APIs serving various purposes: templating, Swing UI building, Ant scripting, JMX integration, SQL access, servlet serving, and more. The Groovy deliverables were bundling all these features and APIs inside a single big JAR. However, not everybody needs everything at all times in their own applications: you might be interested in the template engine and the servlets if you're writing some web application, but you might only need the Swing builder when working on a rich desktop client program.

Groovy modules

So the first goal of the modularity aspect of this release is to actually split the original Groovy JAR into smaller modules, smaller JARs. The core Groovy JAR is now twice as small, and we have the following feature modules available:

- **Ant**: for scripting Ant tasks for automating administration tasks
- **BSF**: for integrating Groovy in your Java applications with the old Apache Bean Scripting Framework
- **Console**: module containing the Groovy Swing console
- **GroovyDoc**: for documenting your Groovy and Java classes
- **Groovysh**: module corresponding to the Groovysh command-line shell

- **JMX**: for exposing and consuming JMX beans
- **JSON**: for producing and consuming JSON payloads
- **JSR-223**: for integrating Groovy in your Java applications with the JDK 6+ javax.scripting APIs
- **Servlet**: for writing and serving Groovy script servlets and templates
- **SQL**: for querying relational databases
- **Swing**: for building Swing UIs
- **Templates**: for using the template engine
- **Test**: for some test support, like the GroovyTestCase, mocking, and more
- **TestNG**: for writing TestNG tests in Groovy
- **XML**: for producing and consuming XML documents

With Groovy 2, you're now able to just pick up the modules you're interested in, rather than bringing everything on your classpath. However, we still provide the "all" JAR which contains everything, if you don't want to complicate your dependencies for just a few megabytes of saved space. We also provide those JARs compiled with the "invoke dynamic" support as well, for those running on JDK 7.

Extension modules

The work on making Groovy more modular also yielded an interesting new feature: extension modules. By splitting Groovy into smaller modules, a mechanism for modules to contribute extension methods has been created. That way, extension modules can provide instance and static methods to other classes, including the ones from the JDK or third-party libraries. Groovy uses this mechanism to decorate classes from the JDK, to add new useful methods to classes like `String, File`, streams, and many more - for example, a `getText()` method on URL allows you to retrieve the content of a remote URL through an HTTP get. Notice also that those extension methods in your modules are also understood by the static type checker and compiler. But let's now have a look at how you can add new methods to existing types.

Contributing an instance method

To add new methods to an existing type, you'll have to create a helper class that will contain those methods. Inside that helper class, all the extension methods will actually be `public` (the default for Groovy but required if implementing in Java) and `static` (although they will be available on instances of that class). They will always take a first parameter which is actually the instance on which this method will be called. And then following parameters will be the parameters passed when calling the method. This is the same convention use for Groovy categories.

Say we want to add a `greets()` method on `String,` that would greet the name of the person passed in parameters, so that you could that method as follow:

```
assert "Guillaume".greets("Paul") == "Hi Paul, I'm Guil-
laume"
```

To accomplish that, you will create a helper class with an extension method like so:

```
package com.acme

class MyExtension {
    static String greets(String self, String name) {
        "Hi ${name}, I'm ${self}"
    }
}
```

Contributing a static method

For static extension methods, this is the same mechanism and convention. Let's add a new static method to Random to get a random integer between two values, you could proceed as in this class:

```
package com.acme

class MyStaticExtension {
    static String between(Random selfType, int start, int
end) {
        new Random().nextInt(end - start + 1) + start
    }
}
```

That way, you are able to use that extension method as follows:

```
Random.between(3, 4)
```

Extension module descriptor

Once you've coded your helper classes (in Groovy or even in Java) containing the extension methods, you need to create a descriptor for your module. You must create a file called `org.codehaus.groovy.runtime.ExtensionModule` in the `META-INF/services` directory of your module archive. Four essential fields can be defined, to tell the Groovy runtime about the name and version of your module, as well as to point at your helper classes for extension methods with a comma-separated list of class names. Here is what our final module descriptor looks like:

```
moduleName = MyExtension
moduleVersion = 1.0
extensionClasses = com.acme.MyExtension
staticExtensionClasses = com.acme.MyStaticExtension
```

With this extension module descriptor on the classpath, you are now able to use those extension methods in your code, without needing an import or anything else, as those extension methods are automatically registered.

Grabbing an extension

With the @Grab annotation in your scripts, you can fetch dependencies from Maven repositories like Maven Central. With the addition of the @GrabResolver annotation, you can specify your own location for your dependencies as well. If you are "grabbing" an extension module dependency through this mechanism, the extension method will also be installed automatically. Ideally, for consistency, your module name and version should be coherent with the artifact id and version of your artifact.

Summary

Groovy is very popular among Java developers and offers them a mature platform and ecosystem for their application needs. But without resting still, the Groovy development team continues to further improve the language and its APIs to help its users increase their productivity on the Java platform.

Groovy 2.0 responds to three key themes:

- **More performance**: with the **support of JDK 7 Invoke Dynamic** to speed up Groovy for those lucky to have JDK 7 already in production, but also with **static compilation** for JDK 5 and beyond for everyone, and particularly those ready to abandon some aspects of dynamicity to shield themselves from the reach of "monkey patching" and to gain the **same speed as Java**.
- **More Java friendliness**: with the support of the **Java 7 Project Coin enhancements** to keep Groovy and Java as close syntax cousins as ever, and with the **static type checker** to have the same level of feedback and type

safety as provided by the javac compiler for developers using Groovy as a Java scripting language

- **More modularity**: with a new level of modularity, Groovy opens the doors for **smaller deliverables**, for example for integration in **mobile applications** on Android, and allowing the Groovy APIs to grow and evolve with newer versions and newer extension modules, as well as allowing users to contribute extension methods to existing types.

About the Author

As Head of Groovy Development for SpringSource, a division of VMware, Guillaume Laforge is the official Groovy Project Manager, leading the Groovy dynamic language project at Codehaus.

He initiated the creation of the Grails web application framework, and founded the Gaelyk project, a lightweight toolkit for developing applications in Groovy for Google App Engine. He is also a frequent conference speaker presenting Groovy, Grails, Gaelyk, Domain-Specific Languages at JavaOne, GR8Conf, SpringOne2GX, QCon, and Devoxx, among others.

Guillaume is also one of the founding members of the French Java/OSS/IT podcast LesCastCodeurs.

JDK 7u8? by Ashwin Jayaprakash Posted 8 hours ago

1. A lot of traditional Java developers are reluctant towards Groovy because of its dynamic nature.
   The (optional) static type checking offers the best of both worlds and hopefully persuades these Java developers to have a second look at Groovy.

   Nice job guys!