Groovy DevCon 5

Metadata

When	• Brainstorming: May 11th-13th 2009 • Code spikes: May 13th-15th 2009
Where	Paris, France

Participants	• Brainstorming
	  • Paul King
	  • Dierk König
	  • Guillaume Laforge
	  • Peter Niederwieser
	  • Jochen Theodorou
	• Code spikes
	  • Paul King
	  • Guillaume Laforge
	  • Jochen Theodorou

Topics

The topics will range between the next coming versions of Groovy: namely Groovy 1.7 and Groovy 2.0. The main highlights will be the following:

Groovy 1.7

- Modularity of Groovy — Paul King
- Combinator parsers — Guillaume Laforge
- Anonymous Inner Classes / Nested Classes — Jochen Theodorou
- Review Hamlet's GEP 2 - AST Builder Support
- Other potential language/platform topics to be discussed
  - The native launcher
  - Groovy's web service support
  - Triple slashy string or alternative heredocs mechanism
  - Assessment of project Coin proposals that might be relevant to Groovy:
    - null-safe indexing ?[] http://mail.openjdk.java.net/pipermail/coin-dev/2009-March/000047.html
    - binary literals, e.g. 0b00100001: http://mail.openjdk.java.net/pipermail/coin-dev/2009-March/000929.html
    - curly-brace collection literals for sets (before inner class/closure usage rules them out?): http://mail.openjdk.java.net/pipermail/coin-dev/2009-March/001193.html
    - improved exception handling: http://mail.openjdk.java.net/pipermail/coin-dev/2009-February/000003.html
- Administrative/build/project issues
  - Using Gradle for the build
  - DCVS for groovy
  - The GEP system
  - ...

Groovy 2.0

- Fate of JSR-241
- Pattern matching (perhaps too much work to do before 1.7) — Guillaume Laforge
  - C# example: http://www.infoq.com/news/2009/05/Pattern-Matching
  - Scala example: http://www.scala-lang.org/node/120
  - Ruby hacks: http://blog.objectmentor.com/articles/2009/03/16/tighter-ruby-methods-with-functional-style-pattern-matching-using-the-case-gem
- Discuss the design of a new Meta-Object Protocol
  - handling of access to private methods/fields/properties
  - general missing property/method handling (order of lookup, what extension points)
  - are invokeMethod and get/setProperty on GroovyObject still needed? What was their purpose in the first place and how did they develop?
  - How does EMC allow extension to invokeMethod and get/setProperty logics? (see for example http://groovy.codehaus.org/ExpandoMetaClass+-+GroovyObject+Methods) And should that be kept, or is it intelligent to do so?
  - How to isolate meta class registries? Is that a good thing, is it not needed? How does for example Ruby handle this?
  - How easily give any created and not yet created meta class a change? How to isolate such actions?
  - Having more information on call sites
  - more uniform method missing path

- various scoping aspects
    - lexical or dynamic
    - block, class, package, OSGi
    - thread-bound or not
    - state tracing (stacked changes)
- inheritance of meta-changes
- fast
    - adaptable to invoke dynamic
    - avoid synchronization on data
    - relying on immutable data structures
    - change retrieval (transactionality?)
- non leaking abstractions for the user
- discovery of meta-changes
    - meta-inf/services
    - magic package

- Some use cases

```
/*
//add property to number
class Distance {
  def number, name
  Distance(number,name) {
    this.name = name; this.number=number
  }
  String toString(){
    "$number $name"
  }
}

// current MOP
Integer.metaClass.getKilometers = {->new Distance(delegate,"km")}
println 5.kilometers

//new MOP
Integer.metaClass.getKilometers = {->new Distance(delegate,"km")}
println 5.kilometers


//intercept method
class RemoteObject {
  def killServer() {"done."}
}

//current MOP
def mc = RemoteObject.metaClass
def old =mc.getMetaMethod("killServer",[] as Object[])
mc.killServer = {->"you are kidding me!? ok... "+old.doMethodInvoke(delegate,[]
as Object[])}
def ro = new RemoteObject()
println ro.killServer()

//new MOP
mc = RemoteObject.metaClass
def old = mc.getMetaMethod("killServer")
mc.killServer = {->"you are kidding me!? ok... "+old(delegate)}
def ro = new RemoteObject()
println ro.killServer()
```

```
*/

class RemoteObject {
  def killServer() {"done."}
  def killYourself() {"no!"}
}
class MyIn implements Interceptor {
    Object beforeInvoke(Object object, String methodName, Object[] arguments){}
    Object afterInvoke(Object object, String methodName, Object[] arguments,
Object result){
        "original $methodName: $result"
    }
    boolean doInvoke(){true}
}

def mc = RemoteObject.metaClass
mc.methodMissing = {String name, args ->
    "let us "+name
}
def proxy = ProxyMetaClass.getInstance(RemoteObject)
proxy.interceptor = new MyIn()
//RemoteObject.metaClass = proxy
def ro = new RemoteObject()
ro.metaClass = proxy
println ro.killServer()
```

```
println ro.killYourself()
println ro."kill all humans"()
```

```
/*// EMC DSL (since 1.6.0)
Object.metaClass {
   invokeMethod {}
}

// Groovy old way (since 1.0)
class A implements GroovyInterceptable {
  def invokeMethod(String name, args){}
}

// EMC closure assignment way (since ~1.5)
Object.metaClass.invokeMethod = {..}

// alternative annotation based (not yet in)
class A {
  @GroovyInterceptor
  private foo(String name, args){}
}

Object.metaClass().interceptableMethods = ["foo"]*/
//ExpandoMetaClass.enableGlobally()
/*
def a = new ArrayList()
println a.metaClass

ArrayList.metaClass.myMethod2 = {-> println "hi" }
println a.metaClass
a.myMethod2()

def b = new ArrayList()
//b.metaClass.myMethod2 = {-> println "hi b" }
b.myMethod2()
println ArrayList.metaClass
println b.metaClass
*/
```

- issues with 'private'

```
/*
class A {
  final private m() {1}
  def getM(){m()}
  def getN(other){other.m()}
}
def a = new A()
assert a.getM()==1
assert a.getN(a)==1

class B extends A {
  public m(){2}
}
def b = new B()
assert b.getM()==1
assert b.m() == 2
assert b.getN(a)==1
assert b.getN(b)==2
*/

class A {
  private String foo() {"1"}
  def bar(){foo()}
}
def a = new A()
assert a.bar()=="1"

a.metaClass.foo = {-> 2 }
assert a.bar()==2
a.foo() // ok

a.metaClass = null
assert a.bar()=="1"
a.foo() // exception

/*'
class B extends A {
  Integer foo() {2}
}
def b = new B()
assert b.bar()=="1"

B.declaredMethods.findAll{it.name=="foo"}.each {println it}
*/
println "done."



proxy#getValue(String name)

proxy.metaClass.getProperty = {proxy,name-> proxy.getValue(name)};
change(proxy)
proxy.foo.bar.else
```

Notes

- Mocking support is lacking and would need an overhaul in 1.7
- Check for coverage support results validity
- Discussion on extended annotations

```
@Validator({ name.size > 3})
String name

@InRange(18..65)
int age

@Regex(~/\d{5}/)
String zipCode

@SEF def action = { println "foo" } // OK
@SEF { println "foo" } // NO!

@SEF callSideEffectMethod(a, b, c)
```

  - Idea of encoding in the form of Strings: would create a script, which could support ranges, closures, regex. Try first with Strings to see how far we can go.
  - If such annotations can be reused from Java, what's up with parameter names, field access, etc?
  - Two aspects: adding annotations support on the various AST nodes, and how to encode that into bytecode
- Regarding loosening the omission of parens, we've currently got a good compromise on readability
- For Groovy 2.0
  - think about shipping some public stable AST interfaces on top of the concrete AST
  - allow the quoted identfiers for properties, fields, assignemnts
  - allow any character in classes fields, methods
  - fix closure naming for method names with spaces
-
  - Some ideas of what pattern matching could look like, without extending the Groovy grammar

```
class Expr {}
class Num extends Expr {
    int value
    Num(int value) { this.value = value }
}
class Plus extends Expr {
    Expr left, right
    Plus(Expr left, Expr right) {
        this.left = left
        this.right = right
    }
}
class Mult extends Expr {
    Expr left, right
    Mult(Expr left, Expr right) {
        this.left = left
        this.right = right
    }
}

@Newify([Num, Plus])
def createTerm() {
    Plus(Plus(Num(1), Num(2)), Num(3))
}
```

```
def term = createTerm()

// avoid adding a match(closure) method on Object
// poluting the available names
import static groovy.patternmatching.PatterMatcher.*

match(term) { /* ... */ }


def result = term.match {
    Num() { delegate.value } // or it.value?

    // value is a property on the Num POJO
    Num(value) { eval(value) }
    // use the left / right property
    Plus(left, right) { eval(left) + eval(right) }

    // can match a property against a specific value
    Num(value: 5) { 5 }

    // add guards examples
    Num(value > 0) { ... }

    Plus(left: Plus(left: a, right: 2), right) { a + 2 + right }

    // alternation
    Plus(left: a, right: b) | Minus(left: a, right: b) {}
    // fallthrough?
    Plus(left: a, right: b)
    Minus(left: a, right: b) {}
    // can use Object, no need for duck matching
    Object(left, right) // match any object with a left and right

    // factorization
    // factor * a + factor * b == factor * (a + b)
    Plus(left: Mult(left: factor, right: a), right: Mult(left: factor, right: b)) {
        //Mult(left: factor, right: Plus(left: a, right: b))

    }

    //

    // if nothing matches
    nothing { /* return or do something */ }
    // if anything matches
```

```
    anything {}
}
```

- Long discussions on the characteristics MOP2 should have
    - Illustrating advanced corner cases

- AST Nodes to be made proper JavaBeans
- Cleaning / review status of operator overloading
    - especially comparisons

- GEP-2
    - phase, statementsOnly, etc, should be properties on the builder itself
    - properties on the Ast Builder itself

- Modularity with Grapes
    - Status
        - Early discussion phase but leveraging existing concepts like grapes and GROOVY-2116 (James Strachan's service registration proposal but enhanced)
        - Create a branch for playing with the concept and merge back onto 1_7_X/trunk at some point if successful
        - Nice to be able to enable some small bits before 1.7 beta but will depend on speed of creation followed by speed of feedback
    - Motivation
        - How to stop groovy-all jar from getting "too fat"
        - How to evolve GDK without breaking (or at least controlled breakage of) legacy scripts
        - How to better position Groovy for use in a wide range of scenarios (e.g. mobile)
        - How to allow Groovy ecosystem to blossom by allowing many grapes to be created but without requiring all of them to be bundled
    - Proposed changes visible to users
        - leverage grape system even for Groovy's GDK (may involve "bundled grapes")
        - enhance GroovyStarter/LoaderConfiguration to support --grab on commandline and grab in conf file (see also GROOVY-3215 conf file in Groovy)
        - (slightly orthogonal but related change) enhance AST transform support to allow scripts to grab without need for a class or method definition, e.g. "@grab(...) Script" (or replace Script with "this" or something else)
        - non-all jar will be much slimmer but conf file will download other jars or at least use other jars in distribution zip
        - service registration (GROOVY-2116) but enhanced. Each grape can have a META-INF/services/groovy directory with one or more of the following files
            - groovyMethods - list of class files containing normal category methods to be registered at global level
            - groovyStaticMethods - list of class files containing static category methods to be registered at global level
            - groovyExpandoMethods - list of class files containing expando defns
            - groovyAstTransforms - ?? but only transforms that come after grape phase?
            - groovyBuilderMetadata - ?? of help to IDEs, e.g. AntBuilder/SwingBuilder
            - auto import registration
            - runner registration (e.g. easyb runner like testng runner)
        - existing grapes can be changed to better handle "service registration"
        - gdk (or pieces) will start to have version numbers (how to relate version numbers with corresponding runtime versions?)
    - Proposed changes invisible to users
        - all-jar will come bundled with (almost?) all current functionality (opportunity to remove deprecated functionality?)
        - minor changes to current source structure (but do we move to almost like separate 'modules' and move tests)
        - minor changes to build files
    - Points to consider
        - which parts of GDK to tackle All? All but core? Just some less used parts to start with? Tools?
        - granularity of grapes
            - "@grab xml" vs "@grab xml.slurper and @grab xml.parser"
            - "@grab jmx" vs "@grab jmx.builder"
            - names to use? ivy/maven/OSGi
            - versioning scheme to use
        - potential grapes:
          core?, numbers?, strings? bsf, jsr223, sql, xml, jmx, time, date, collections, cli, maps, file, streams, process, threads, templates, regex?, encoding, ant, newify?, googleCollections, jsr203/nio, jodatime, actors, combinators, BigTable?, Scala compiler hooks/Scalify
        - format of META-INF/services file: list of classes or groovy file
        - graceful degradation: jars, poms, OSGi manifest, META-INF/services
        - think through implications for various embedded usages of Groovy