Groovy 2.2 release notes

Implicit closure coercion

Java 8 will feature lambdas, which are similar to Groovy's closures. One particular aspect which is interesting with lambdas is how they are converted transparently by Java to interface types that contain one single abstract method. With Groovy closures, except for a few cases, we have to explicitly use the as operator to do the coercion. In Groovy 2.2, we are allowing the same transparent closure coercion to happen, but without the explicit as type coercion, and furthermore, we make it possible to work as well with abstract classes as well.

```
interface Predicate {
    boolean test(obj)
}

List filter(List list, Predicate pred) {
    list.findAll { pred.test(it) }
}

def input = [1, 2, 3, 4, 5]

def odd = filter(input) { it % 2 == 1 }
assert odd == [1, 3, 5]
```

Notice how the closure is coerced into a Predicate instance. Without that new capabilities, we would have had to write the following instead:

```
def odd = filter(input, { it % 2 == 1} as Predicate)
```

That way, Groovy closure coercion to SAM types is as concise as Java 8 lambda closure conversion.

Here's another example using abstract classes, which are not supported by Java 8 lambda conversion:

```
abstract class SurroundedMessage {
    abstract String message()

    String surrounded() {
        "<< ${message()} >>"
    }
}
SurroundedMessage m = { 'hello' }
assert m.surrounded() == '<< hello >>'
```

@Memoized AST transformation for methods

Similarly to our Closure memoization capability, you can now annotate your methods with the new @Memoized annotation. It will use the same underlying cache solution used for closures, and will cache the result of previous executions of the annotated method with the same entry

parameters.

```
import groovy.transform.Memoized

@Memoized int expensiveOp(int a, int b) {
    sleep 1000
    return a + b
}
// one second to return
expensiveOp(1, 2)

// immediate result returned

expensiveOp(1, 2)
```

Bintray's JCenter repository

The default Grab configuration now use Bintray's JCenter repository as the first in the chain of resolvers, as Bintray's JCenter repository is noticeably faster and more responsive than Maven Central, offers dependencies always with their checksums, and stores and caches dependencies it wouldn't have for faster delivery the next time a dependency is required. This should make your scripts relying on @Grab faster when downloading dependencies for the first time.

Define base script classes with an annotation

All scripts usually extend the groovy.lang.Script abstract class, but it's possible to set up our own base script class extending Script through CompilerConfiguration. A new AST transformation is introduced in Groovy 2.2 which allows you to define the base script class as follows:

```
import groovy.transform.BaseScript

abstract class DeclaredBaseScript extends Script {
    int meaningOfLife = 42
}

@BaseScript DeclaredBaseScript baseScript

assert meaningOfLife == 42
```

New DelegatingScript base class for scripts

With the CompilerConfiguration class that you pass to GroovyShell (as well as GroovyClassLoader and others), you can define a base script class for the scripts that will be compiled with that shell. It’s handy to share common methods to all scripts.

For DSL purposes, it’s interesting to actually delegate the method calls and unbound variable assignments to a different object than the script itself, thanks to the new DelegatingScript.

To make it more concrete, let’s have a look at the following configuration script:

```groovy
// import the CompilerConfiguration class
// to configure the base script class
import org.codehaus.groovy.control.CompilerConfiguration

// the script we want to compile
def scriptContent = '''
    name = "Guillaume"
    sayHi()
'''

// the class definition of the delegate
class Person {
    String name
    void sayHi() {
        println "Hi $name"
    }
}

// configure the base script class
def cc = new CompilerConfiguration()
cc.scriptBaseClass = DelegatingScript.class.name

// parse script with GroovyShell
// and the configuration
def sh = new GroovyShell(cc)
def script = sh.parse(scriptContent)

// set the delegate and run the script
def p = new Person()
script.setDelegate(p)
script.run()

// the name is set correctly
// and the output will display "Hi Guillaume"
assert p.name == "Guillaume"
```

New @Log variant for the Log4j2 logging framework

A new @Log variant has been added to support Log4j2, with the @Log4j2 AST transformation:

```groovy
import groovy.util.logging.Log4j2

@Log4j2
class MyClass {
    void doSomething() {
        log.info "did something groovy today!"
    }
}
```

@DelegatesTo with generics type tokens

The @DelegatesTo annotation, introduced in Groovy 2.1 that helps the type checker, IDEs, tools, to provide better support for DSLs using closure delegation strategies, works with generics token types as well. You can tell Groovy that the delegatee is of the type of the generics component:

```
import groovy.transform.*

@InheritConstructors
class MyList extends LinkedList<String> {}

public <T> Object map(
        @DelegatesTo.Target List<T> target,
        @DelegatesTo(genericTypeIndex = 0) Closure arg) {
    arg.delegate = target.join('')
    arg()
}

@TypeChecked
def test() {
    map(new MyList(['f', 'o', 'o'])) {
        assert toUpperCase() == 'FOO'
    }
}
```

Note the genericTypeIndex attribute of @DelegatesTo that points at the index of the generic component. Unfortunately, as the generic placeholders are not kept at the bytecode level, it's impossible to just reference T, and we had to use an index to point at the right type.

Precompiled type checking extensions

The static type checking extensions introduced in Groovy 2.1 were working solely with non-compiled scripts. But with this beta, you can also specify a fully-qualified name of the precompiled class implementing your extension:

```
@TypeChecked(extensions = 'com.enterprise.MyDslExtension')
```

Type checking extensions now also support two more events: ambiguousMethods and incompatibleReturnType.

Groovysh enhancements

Groovysh has been expanded with various enhancements:

- support for code completion in various places, like imports, package names, class names, variable names, parameter names, keywords, etc.
- a doc command allows you to open the relevant JavaDoc and Groovy GDK web pages to have more information for a given class, for example try in Groovysh:
  doc java.util.List
- you can complete file names inside strings, particularly handy for your scripting tasks where you want to open a file with new File("data.|") (where the pipe character is the position of your cursor), and then hit the TAB key to have the completion of the file name

OSGi manifests for the "invoke dynamic" JARs

If you're using Groovy in the context of an OSGi container, the Groovy JARs contained the right OSGi metadata information in its manifest. However, it wasn't the case for the "invoke dynamic" JARs, as the underlying library used by the Gradle OSGi plugin wasn't supporting JDK 7 bytecode. Fortunately, this deficiency has been fixed, the Gradle OSGi plugin updated, and we're now able to have our "indy" JARs work fine under OSGi has well.