Looking into Groovy 1.8

JFall 2011 - http://bit.ly/jfall2011-groovy18

Hubert Klein Ikkink (mrhaki)

Take and Drop Items from a List

When working with List object we get a lot of nice and useful methods we can use in Groovy. Since Groovy 1.8.1 we can use the methods take() and drop(). With the take() method we get items from the beginning of the List. We pass the number of items we want as an argument to the method.
To remove items from the beginning of the List we can use the drop() method. Here we pass the number of items we want to drop as an argument to the method. Please keep in mind the original list is not changed, the result of the drop() method is a new list.

```
def list = ['Simple', 'list', 'with', 5, 'items']

assert list.take(1) == ['Simple']
assert list.take(2) == ['Simple', 'list']
assert list.take(0) == []
// Whole list, because we take more items then the size of list
assert list.take(6) == ['Simple', 'list', 'with', 5, 'items']

assert list.drop(1) == ['list', 'with', 5, 'items']
assert list.drop(3) == [5, 'items']
assert list.drop(5) == []
assert list.drop(0) == ['Simple', 'list', 'with', 5, 'items']
assert list == ['Simple', 'list', 'with', 5, 'items']

// After reading Tim Yates' comment I have added
// more samples showing drop() and take() also work on
// Maps, Iterators, CharSequences and arrays.
def array = ['Rock on!', 'Groovy baby!'] as String[]
assert array.take(1) == ['Rock on!'] as String[]
assert array.drop(1) == ['Groovy baby!'] as String[]

def range = 0..10
assert range.take(2) == [0,1]
assert range.take(4) == 0..3
assert range.drop(5) == 5..10

def map = [1: 'one', 2: 'two', 3: 'three']
assert map.take(2) == [1: 'one', 2: 'two']
assert map.drop(2) == [3: 'three']
assert map.drop(3) == [:]

def s = 'Hello Groovy world!'
```

```
assert s.take(5) == 'Hello'
assert s.drop(6) == 'Groovy world!'
```

Use inject Method on a Map

The inject() method is since Groovy 1.8.1 also available for Map objects. The closure arguments accepts two or three arguments. With the three-argument variant we get the key and value separately as arguments. Otherwise we get a map entry as closure argument.

```
// 3-argument closure with key, value.
def m = [user: 'mrhaki', likes: 'Groovy']
def sentence = m.inject('Message: ') { s, k, v ->
    s += "${k == 'likes' ? 'loves' : k} $v " }

assert sentence.trim() == 'Message: user mrhaki loves Groovy'

// 2-argument closure with entry.
def map = [sort: 'name', order: 'desc']
def equalSizeKeyValue = map.inject([]) { list, entry ->
    list << (entry.key.size() == entry.value.size()) }

assert equalSizeKeyValue == [true, false]
```

Use Connection Parameters to Get Text From URL

For a long time we can simply get the text from an URL in Groovy. Since Groovy 1.8.1 we can set parameters to the underlying URLConnectionthat is used to get the content. The parameters are passed as a Map to the getText() method or to the newReader() ornewInputStream() methods for an URL.

We can set the following parameters:

- connectTimeout in milliseconds
- readTimeout in milliseconds
- useCaches
- allowUserInteraction
- requestProperties is a Map with general request properties

```
// Contents of http://www.mrhaki.com/url.html:
// Simple test document
// for testing URL extensions
// in Groovy.

def url = "http://www.mrhaki.com/url.html".toURL()

// Simple Integer enhancement to make
// 10.seconds be 10 * 1000 ms.
Integer.metaClass.getSeconds = { ->
    delegate * 1000 }

// Get content of URL with parameters.
```

```
def content = url.getText(connectTimeout: 10.seconds, readTimeout: 10.seconds,
                          useCaches: true, allowUserInteraction: false,
                          requestProperties: ['User-Agent': 'Groovy Sample Script'])

assert content == '''\
Simple test document
for testing URL extensions
in Groovy. '''

url.newReader(connectTimeout: 10.seconds, useCaches: true).withReader { reader ->
    assert reader.readLine() == 'Simple test document' }
```

Streaming JSON with StreamingJsonBuilder

Since Groovy 1.8 we can use JSONBuilder to create JSON data structures. With Groovy 1.8.1 we have a variant of JsonBuilder that will not create a data structure in memory, but will stream directly to a writer the JSON structure: StreamingJsonBuilder. This is useful in situations where we don't have to change the structure and need a memory efficient way to write JSON.

```
import groovy.json.*

def jsonWriter = new StringWriter()
def jsonBuilder = new StreamingJsonBuilder(jsonWriter)
jsonBuilder.message {
    header {
        from(author: 'mrhaki')
        to 'Groovy Users', 'Java Users'
    }
    body "Check out Groovy's gr8 JSON support." }
def json = jsonWriter.toString()
assert json == '{"message":{"header":{"from":{"author":"mrhaki"},"to":["Groovy
Users","Java Users"]},"body":"Check out Groovy\'s gr8 JSON support."}}'

def prettyJson = JsonOutput.prettyPrint(json)
assert prettyJson == '''{
    "message": {
        "header": {
            "from": {
                "author": "mrhaki"
            },
            "to": [
                "Groovy Users",
                "Java Users"
            ]
        },
        "body": "Check out Groovy's gr8 JSON support."
    } }'''


new StringWriter().withWriter { sw ->
    def builder = new StreamingJsonBuilder(sw)
```

```
    // Without root element.
    builder name: 'Groovy', supports: 'JSON'

    assert sw.toString() == '{"name":"Groovy","supports":"JSON"}' }

new StringWriter().with { sw ->
    def builder = new StreamingJsonBuilder(sw)

    // Combine named parameters and closures.
    builder.user(name: 'mrhaki') {
        active true
    }

    assert sw.toString() == '{"user":{"name":"mrhaki","active":true}}'     }
```

Collect on Nested Collections

The collect() method has been around in Groovy for a long time and it is very useful. With the collect() method we can iterate through a collection and transform each element with a Closure to another value. To apply a transformation to collections in collections we can use thecollectAll() method. Since Groovy 1.8.1 the collectAll() method is deprecated in favor of the new collectNested() method. So with collectNested() we can transform elements in a collection and even in nested collections and the result will be a collection (with nested collections) with transformed elements. We can pass an initial collection to the method to which the transformed elements are added.

```
def list = [10, 20, [1, 2, [25, 50]], ['Groovy']]

assert list.collectNested { it * 2 } == [20, 40, [2, 4, [50, 100]], ['GroovyGroovy']]
assert list.collectNested(['1.8.1', [0]]) { it * 2 } == ['1.8.1', [0], 20, 40, [2, 4,
[50, 100]], ['GroovyGroovy']]
assert list.collectNested([]) { it * 2 } == [20, 40, [2, 4, [50, 100]],
['GroovyGroovy']]

// Simple collect will duplicate the nested collection instead
// of elements in the nested collection.
assert list.collect { it * 2 } == [20, 40, [1, 2, [25, 50], 1, 2, [25, 50]],
['Groovy', 'Groovy']]
```

GroupBy with Multiple Closures

We can group elements in a List or Map with the groupBy() method for a long time in Groovy. We pass a closure with the grouping condition to get a Map with the items grouped. And since Groovy 1.8.1 we can use more than closure to do the grouping. We can use it for both List and Map objects.

```
import static java.util.Calendar.*

class User {
    String name
```

```
    String city
    Date birthDate
    public String toString() { name } }

def users = [
    new User(name:'mrhaki', city: 'Tilburg',   birthDate: Date.parse('yyyy-MM-
dd', '1973-9-7')),
    new User(name:'bob',    city: 'New York',  birthDate: Date.parse('yyyy-MM-
dd', '1963-3-30')),
    new User(name:'britt',  city: 'Amsterdam', birthDate: Date.parse('yyyy-MM-
dd', '1980-5-12')),
    new User(name:'kim',    city: 'Amsterdam', birthDate: Date.parse('yyyy-MM-
dd', '1983-3-30')),
    new User(name:'liam',   city: 'Tilburg',   birthDate: Date.parse('yyyy-MM-
dd', '2009-3-6'))
]

def result = users.groupBy({it.city}, {it.birthDate.format('MMM')})

assert result.toMapString() ==
    '[Tilburg:[Sep:[mrhaki], Mar:[liam]], New York:[Mar:[bob]],
Amsterdam:[May:[britt], Mar:[kim]]]'

assert result.Amsterdam.size() == 2
assert result.Tilburg.Mar.name == ['liam']

result = users.groupBy({it.name[0]}, {it.city})
assert result.toMapString() ==
    '[m:[Tilburg:[mrhaki]], b:[New York:[bob], Amsterdam:[britt]], k:[Amsterdam:[kim]
], l:[Tilburg:[liam]]]'
assert result.k.Amsterdam.name == ['kim']

// groupBy with multiple closues also works on Map
def usersByCityMap = users.groupBy({it.city})
def resultMap = usersByCityMap.groupBy({it.value.size()}, { k,v -> k.contains('i') })
assert resultMap.toMapString() ==
    '[2:[true:[Tilburg:[mrhaki, liam]], false:[Amsterdam:[britt, kim]]],
1:[false:[New York:[bob]]]]'
assert resultMap[1].size() == 1
assert resultMap[2].size() == 2
assert resultMap[2][true].Tilburg.name.join(',') == 'mrhaki,liam'
```

Sort or Remove Duplicates without Changing the Original Collection

We can sort and remove duplicates from collections or arrays in Groovy for a long time. The sort() and unique() methods alter the original collection, so there is a big side effect. Since Groovy 1.8.1 we can use a boolean argument to indicate if we want to mutate the original collection or not. So now we can remove the side effect and leave the original collection or array unchanged.

The reverse() method also accepts this boolean argument since Groovy 1.8.1, but the

default behavior was already to leave the original collection unchanged. Now we can explicitly tell the reverse() method to change the original collection.

```
class User implements Comparable {
    String username, email
    boolean active

    int compareTo(Object other) {
        username <=> other.username
    }

    String toString() { "[User:$username,$email,$active]" } }

def mrhaki1 = new User(username: 'mrhaki', email: 'mrhaki@localhost', active: false)
def mrhaki2 = new User(username: 'mrhaki', email: 'user@localhost', active: true)

def hubert1 = new User(username: 'hubert', email: 'user@localhost', active: false)
def hubert2 = new User(username: 'hubert', email: 'hubert@localhost', active: true)

def users = [mrhaki1, mrhaki2, hubert1, hubert2]


/* Sort */
def sortedUsers = users.sort(mutate = false)  // Don't mutate original List
assert sortedUsers == [hubert1, hubert2, mrhaki1, mrhaki2]
assert users == [mrhaki1, mrhaki2, hubert1, hubert2]

// Default behaviour is to change original collection
sortedUsers = users.sort()
assert sortedUsers == [hubert1, hubert2, mrhaki1, mrhaki2]
assert users == [hubert1, hubert2, mrhaki1, mrhaki2]


/* Unique */
def uniqueUsers = users.unique(mutate = false)  // Don't mutate original List
assert uniqueUsers == [hubert1, mrhaki1]
assert users == [hubert1, hubert2, mrhaki1, mrhaki2]

// Default behaviour is to change original collection
uniqueUsers = users.unique()
assert uniqueUsers == [hubert1, mrhaki1]
assert users == [hubert1, mrhaki1]


/* Reverse */
def reverseUsers = users.reverse()  // mutate false is default
assert reverseUsers == [mrhaki1, hubert1]
assert users == [hubert1, mrhaki1]

// Default behaviour is to leave original collection alone
reverseUsers = users.reverse(mutate = true)  // Mutate original list
assert reverseUsers == [mrhaki1, hubert1]
assert users == [mrhaki1, hubert1]
```

Add AST Transformations Transparently to Scripts

With Groovy 1.8 we can add compilation customizers when for example we want to run a Groovy script from our application code. Cedric Champeau has a nice blog post about this feature. And we already learned about the import customizer in a previous Groovy Goodness blog post.

Another customizer is the ASTTransformationCustomizer. We can use this customizer to add an AST transformation to a script. The AST transformation is then applied to all classes in the script. This means we can only apply transformations that can be used at class level. Another thing we need to notice is that we cannot set parameters for the AST transformation, so only the defaults for the parameters are used.

We can use both local and global AST transformations with the ASTTransformationCustomizer. For a local transformation we use the annotation class, for a global transformation we pass the AST transformation class to the constructor.

```
package com.mrhaki.blog

import org.codehaus.groovy.control.customizers.ASTTransformationCustomizer
import org.codehaus.groovy.control.CompilerConfiguration
import groovy.transform.Canonical

def conf = new CompilerConfiguration()
conf.addCompilationCustomizers(new ASTTransformationCustomizer(Canonical))

def shell = new GroovyShell(conf)
shell.evaluate '''
package com.mrhaki.blog

class User {
    String username, fullname }

// TupleConstructor is added by Canonical transformation.
def user = new User('mrhaki', 'Hubert A. Klein Ikkink')
// ToString is added by Canonical transformation.
assert user.toString() == 'com.mrhaki.blog.User(mrhaki, Hubert A. Klein Ikkink)'

// AST transformation is also applied to the Script class.
assert this.toString() == 'com.mrhaki.blog.Script1()' '''
```

Add Imports Transparently to Scripts with ImportCustomizer

Since Groovy 1.8.0 we can easily setup import statements for a Groovy compilation unit (for example a Groovy script file) *without* adding the imports to the script source. For example we have created domain classes in our own package. The script authors shouldn't know about the packages and just pretend the classes are accessible from

the 'default' package.

We define an ImportCustomizer and use the addImport methods to add packages, classes, aliases to our script. The configuredImportCustomizer is added to a CompilerConfiguration object. We will see in future blog posts how we can even add more customizers to the CompilerConfiguration. We use the configuration when we create a GroovyShell and the information is applied to scripts we run with the created shell.

First we create a class and enum in the com.mrhaki.blog package. We compile the code so we have class files we can add to a classpath of another application.

```
// File: Post.groovy
// Compile with groovyc Post.groovy
package com.mrhaki.blog

class Post {
    String title
    Type type = Type.BLOG }

enum Type {
    BLOG, NEWS }
```

Next we create a Groovy script that will execute the following code:

```
// File: sample.groovy
// Article is imported as alias for com.mrhaki.blog.Post
def article = new Article(title: 'Groovy Goodness')

assert article.title == 'Groovy Goodness'
// BLOG is statically imported from com.mrhaki.blog.Type.*
assert article.type == BLOG
assert article.class.name == 'com.mrhaki.blog.Post'
```

Now we are ready to create a small application that will execute *sample.groovy*. We must add the com.mrhaki.blog.Post andcom.mrhaki.blog.Type compiled classes to the classpath if we run the following script:

```
// File: RunScript.groovy
// Run with groovy -cp . RunScript
import org.codehaus.groovy.control.customizers.ImportCustomizer
import org.codehaus.groovy.control.CompilerConfiguration

// Add imports for script.
def importCustomizer = new ImportCustomizer()
// import static com.mrhaki.blog.Type.*
importCustomizer.addStaticStars 'com.mrhaki.blog.Type'
// import com.mrhaki.blog.Post as Article
importCustomizer.addImport 'Article', 'com.mrhaki.blog.Post'

def configuration = new CompilerConfiguration()
```

```
configuration.addCompilationCustomizers(importCustomizer)

// Create shell and execute script.
def shell = new GroovyShell(configuration)
shell.evaluate new File('sample.groovy')
```

Apply Read and Write Locking

When we write applications with concurrency needs we can use the ReentrantReadWriteLock class to specify a lock on specific methods. Normally we have to add a field to our class of the type ReentrantReadWriteLock and then write code to get a read lock or write lock, do our stuff and finally release the lock again. Since Groovy 1.8 we can use the AST transformation annotations duo WithReadLock andWithWriteLock. Groovy will add a lock field and adds the code to acquire and release the locks again in our compiled class file. We don't have to write this code ourselves anymore.

```
import groovy.transform.*

class Bucket {
    final def items = []

    @WithWriteLock
    void add(String item) {
        items << item
    }

    @WithReadLock
    List getAllItems() {
        items
    } }

def bucket = new Bucket()

// Create read/write threads that use the same
// bucket instance.
def allThreads = []
3.times {
    allThreads << Thread.start {
        5.times {
            bucket.add Thread.currentThread().name + ': Groovy rocks'
            println "> ${Thread.currentThread().name} adds item to bucket."
            sleep 100
        }
    }

    allThreads << Thread.start {
        3.times {
            int numberOfItems = bucket.allItems.size()
            println "< ${Thread.currentThread().name} says $numberOfItems items are
in the bucket."
            sleep 150
        }
```

```
    } }

allThreads.each { it.join() }

assert bucket.items.size() == 15
println bucket.items
```

The following output shows the results of running the code. This output will be different each time we run the code.

```
> Thread-482 adds item to bucket.
< Thread-483 says 1 items are in the bucket.
> Thread-484 adds item to bucket.
< Thread-485 says 2 items are in the bucket.
> Thread-486 adds item to bucket.
< Thread-487 says 3 items are in the bucket.
> Thread-482 adds item to bucket.
> Thread-484 adds item to bucket.
> Thread-486 adds item to bucket.
< Thread-483 says 6 items are in the bucket.
< Thread-485 says 6 items are in the bucket.
< Thread-487 says 6 items are in the bucket.
> Thread-482 adds item to bucket.
> Thread-484 adds item to bucket.
> Thread-486 adds item to bucket.
> Thread-482 adds item to bucket.
< Thread-483 says 10 items are in the bucket.
> Thread-484 adds item to bucket.
< Thread-485 says 11 items are in the bucket.
> Thread-486 adds item to bucket.
< Thread-487 says 12 items are in the bucket.
> Thread-482 adds item to bucket.
> Thread-484 adds item to bucket.
> Thread-486 adds item to bucket.
[Thread-482: Groovy rocks, Thread-484: Groovy rocks, Thread-486: Groovy rocks,
Thread-482: Groovy rocks, Thread-484: Groovy rocks, Thread-486: Groovy rocks, Thread-
482: Groovy rocks, Thread-484: Groovy rocks, Thread-486: Groovy rocks, Thread-482:
Groovy rocks, Thread-484: Groovy rocks, Thread-486: Groovy rocks, Thread-482: Groovy
rocks, Thread-484: Groovy rocks, Thread-486: Groovy rocks]
```

Command Chain Expressions for Fluid DSLs

Groovy 1.8 introduces command chain expression to further the support for DSLs. We already could leave out parenthesis when invoking top-level methods. But now we can also leave out punctuation when we chain methods calls, so we don't have to type the dots anymore. This results in a DSL that looks and reads like a natural language.

Let 's see a little sample where we can see how the DSL maps to real methods with arguments:

```
// DSL:
take 3.apples from basket
```

```
// maps to:
take(3.apples).from(basket)

// DSL (odd number of elements):
calculate high risk
// maps to:
calculate(high).risk
// or:
calculate(high).getRisk()

// DSL:
// We must use () for last method call, because method doesn't have arguments.
talk to: 'mrhaki' loudly()
// maps to:
talk(to: 'mrhaki').loudly()
```

Implementing the methods to support these kind of DSLs can be done using maps and closures. The following sample is a DSL to record the time spent on a task at different clients:

```
worked 2.hours on design at GroovyRoom
developed 3.hours at OfficeSpace
developed 1.hour at GroovyRoom
worked 4.hours on testing at GroovyRoom
```

We see how to implement the methods to support this DSL here:

```
// Constants for tasks and clients.
enum Task { design, testing, developing }
enum Client { GroovyRoom, OfficeSpace }

// Supporting class to save work item info.
class WorkItem {
    Task task
    Client client
    Integer hours }

// Support syntax 1.hour, 3.hours and so on.
Integer.metaClass.getHour = { -> delegate }
Integer.metaClass.getHours = { -> delegate }

// Import enum values as constants.
import static Task.*
import static Client.*

// List to save hours spent on tasks at
// different clients.
workList = []

def worked(Integer hours) {
    ['on': { Task task ->
        ['at': { Client client ->
            workList << new WorkItem(task: task, client: client, hours: hours)
```

```
        }]
    }] }

def developed(Integer hours) {
    ['at': { Client client ->
        workList << new WorkItem(task: developing, client: client, hours: hours)
    }] }

// ---------------------------------
// DSL
// ---------------------------------
worked 2.hours on design at GroovyRoom
developed 3.hours at OfficeSpace
developed 1.hour at GroovyRoom
worked 4.hours on testing at GroovyRoom


// Test if workList is filled
// with correct data.
def total(condition) {
    workList.findAll(condition).sum { it.hours } }

assert total({ it.client == GroovyRoom }).hours == 7
assert total({ it.client == OfficeSpace }).hours == 3
assert total({ it.task == developing }).hours == 4
assert total({ it.task == design }).hours == 2
assert total({ it.task == testing }).hours == 4
```

Canonical Annotation to Create Mutable Class

With Groovy 1.8 we get a lot of AST transformations. We can combine @ToString, @EqualsAndHashCode and @TupleConstructor with the@Canonical annotation. This annotation will do all the transformations at once. If we want to customize one of the AST transformations, we add the annotation with configuration parameters extra after @Canonical.

```
import groovy.transform.*

@Canonical
class Building {
    String name
    int floors
    boolean officeSpace }

// Constructors are added.
def officeSpace = new Building('Initech office', 1, true)

// toString() added.
assert officeSpace.toString() == 'Building(Initech office, 1, true)'

// Default values are used if constructor
// arguments are not assigned.
def theOffice = new Building('Wernham Hogg Paper Company')
```

```
assert theOffice.floors == 0
theOffice.officeSpace = true

def anotherOfficeSpace = new Building(name: 'Initech office', floors: 1, officeSpace:
true)

// equals() method is added.
assert anotherOfficeSpace == officeSpace

// equals() and hashCode() are added, so duplicate is not in Set.
def offices = [officeSpace, anotherOfficeSpace, theOffice] as Set
assert offices.size() == 2
assert offices.name.join(',') == 'Initech office,Wernham Hogg Paper Company'

@Canonical
@ToString(excludes='age')  // Customize one of the transformations.
class Person {
    String name
    int age }

def mrhaki = new Person('mrhaki', 37)
assert mrhaki.toString() == 'Person(mrhaki)'
```

Cache Closure Results with Memoization

Closures are very powerful in Groovy. Groovy 1.8 introduces closure memoization. This means we can cache the result of a closure, so the next time we invoke the closure the result is returned immediately. This is very useful for time consuming computations in a closure.

To use this feature we invoke the memoize() method on a closure. Now the results from calls to the closure are cached. We can use three other methods to define for example the maximum number of calls to cache, or the least number of calls with memoizeAtMost(),memoizeAtLeast() and memoizeBetween().

```
// Closure simple increments parameter.
// Also script variable incrementChange is
// changed so we can check if the result is
// from a cached call or not.
def incrementChange = false
def increment = {
    incrementChange = true
    it + 1  }
// Just invoke the closure 5 times with different parameters.
(0..5).each {
    incrementChange  = false
    assert increment(it) == it + 1
    assert incrementChange }
incrementChange = false
assert increment(1) == 2  // Call is not cached.
assert incrementChange
```

```
// Use memoize() so all calls are cached.
incrementChange = false
def incrementMemoize = increment.memoize()
// Just invoke the closure 5 times with different parameters.
(0..5).each {
    incrementChange = false
    assert incrementMemoize(it) == it + 1
    assert incrementChange }
incrementChange = false
assert incrementMemoize(2) == 3  // Cached call.
assert !incrementChange

// Use memoizeAtMost().
incrementChange = false
def memoizeAtMostOnce = increment.memoizeAtMost(1)
// Just invoke the closure 5 times with different parameters.
(0..5).each {
    incrementChange = false
    assert memoizeAtMostOnce(it) == it + 1
    assert incrementChange }
incrementChange = false
assert memoizeAtMostOnce(1) == 2  // 2nd call is not cached.
assert incrementChange
```

Recursion with Closure Trampoline Capability

When we write recursive code we might get a stack overflow exception, because calls are placed on the stack to be resolved. Since Groovy 1.8 we can use the trampoline capability of closures to overcome this problem.

We invoke a trampoline() method on a closure and our original closure is now wrapped in TrampolineClosure instance. Calls to theTrampolineClosure are executed sequentially invoking the original closure, until the original closure returns something else then aTrampolineClosure instance. This way the stack isn't filled and we won't get the stack overflow exceptions.

```
def sizeList
sizeList = { list, counter = 0 ->
    if (list.size() == 0) {
        counter
    } else {
        sizeList.trampoline(list.tail(), counter + 1)
    } }
sizeList = sizeList.trampoline()

assert sizeList(1..10000) == 10000
```

New Dollar Slashy Strings

Groovy already has a lot of ways to define a String value, and with Groovy 1.8 we have another one: the dollar slashy String. This is closely related to the slashy String

definition we already knew (which also can be multi-line by the way, added in Groovy 1.8), but with different escaping rules. We don't have to escape a slash if we use the dollar slashy String format, which we would have to do otherwise.

```
def source = 'Read more about "Groovy" at http://mrhaki.blogspot.com/'

// 'Normal' slashy String, we need to escape / with \/
def regexp = /.*"(.*)".*\/(.*)\//

def matcher = source =~ regexp
assert matcher[0][1] == 'Groovy'
assert matcher[0][2] == 'mrhaki.blogspot.com'

// Dollar slash String.
def regexpDollar = $/.*"(.*)".*/(.*)//$

def matcherDollar = source =~ regexpDollar
assert matcher[0][1] == 'Groovy'
assert matcher[0][2] == 'mrhaki.blogspot.com'

def multiline = $/
Also multilines
are supported.
/$
```

Chain Closures Together with Closure Composition

There are a lot of new features in Groovy 1.8. One of them is the possibility to compose a new closure by chaining two other closures together. We use the leftShift and rightShift operators (<< and >>) to combine multiple closures to create a new closure.

```
def convert = { new Expando(language: it) }
def upper = { it.toUpperCase() }

// Composition.
def upperConvert = convert << upper

def languages = ['Groovy', 'Scala', 'Clojure'].collect(upperConvert)
println languages // Output: [{language=GROOVY}, {language=SCALA}, {language=CLOJURE}
]
assert languages[0].language == 'GROOVY'
assert languages[1].language == 'SCALA'
assert languages[2].language == 'CLOJURE'

// Reverse composition.
def lastLetter = { it[-1] }
def firstLetters = ['Groovy', 'Clojure', 'Scala'].collect(upper >> lastLetter)
assert firstLetters.join() == 'YEA'
```

See if Sets are Equal

Since Groovy 1.8 we can use the equals() method to compare the contents of two Set

collections. The two sets are equals if they have the same size and all elements are in both sets.

```
def set1 = ['Java', 42, true] as Set
def set2 = ['Groovy', 42, true, 'Java'] as Set
def set3 = [42L, true, 'Java'] as Set

assert set1.equals(set3)
assert set1 == set3
assert !set1.equals(set2)
assert set2 != set3
```

Add Items to a List at Specified Position

Since Groovy 1.8 we can create a new list from two lists, where the second list is 'inserted' at a specified index position in the first list. The original lists are not changed: the result is a new list. Groovy also adds the addAll() method to List objects (since Groovy 1.7.2), but then the original list object is changed.

```
def list = ['Gr', 'vy']

assert list.plus(1, 'oo') == ['Gr', 'oo', 'vy']
assert list == ['Gr', 'vy']

list.addAll(1, 'oo')
assert list == ['Gr', 'oo', 'vy']

assert (1..10).plus(5, 6..7) == [1,2,3,4,5,6,7,6,7,8,9,10]
```

Generate equals and hashcode Methods with EqualsAndHashCode Annotation

There are a lot of new bytecode generation annotation in Groovy 1.8. One of them is the @EqualsAndHashCode annotation. With this annotation an equals() and hashCode() method is generated for a class. The hashCode() method is implemented using Groovy'sorg.codehaus.groovy.util.HashCodeHelper (following an algorithm from the book *Effective Java*). The equals() method looks at all the single properties of a class to see of both objects are the same.
We can even include class fields instead of only properties for generating both methods. We only have to use includeFields=true when we assign the annotation.
To include calls to a super class we use the annotation attribute callSuper and assign the value true. Finally we can also exclude properties or fields from hashcode calculation or equal comparisons. We use the annotation attribute excludes for this and we can assign a list of property and field names.

```
import groovy.transform.EqualsAndHashCode

@EqualsAndHashCode(includeFields=true)
class User {
```

```
    String name
    boolean active
    List likes
    private int age = 37 }

def user = new User(name: 'mrhaki', active: false, likes: ['Groovy', 'Java'])
def mrhaki = new User(name: 'mrhaki', likes: ['Groovy', 'Java'])
def hubert = new User(name: 'Hubert Klein Ikkink', likes: ['Groovy', 'Java'])

assert user == mrhaki
assert mrhaki != hubert

Set users = new HashSet()
users.add user
users.add mrhaki
users.add hubert
assert users.size() == 2
```

Tuple Constructor Creation

Groovy 1.8 adds the @TupleConstructor annotation. With this annotation we can automatically create a tuple constructor at compile time. So the constructor can be found in the compiled class. For each property in the class a parameter in the constructor is created with a default value. The order of the properties defined in the class also defines the order of parameters in the constructor. Because the parameters have default values we can use Groovy syntax and leave parameters at the end of the parameter list out when we use the constructor.

We can also include fields as constructor parameters. We use the annotation attribute includeFields=true to activate this.

If we define our constructors in the class than the TupleConstructor annotation will not create extra constructors. But we can override this behaviour with the attribute value force=true. We have to make sure we don't have constructor conflicts ourselves, because now the annotation will create the extra constructors.

If our class extends another class and we want to include the properties or fields of the super class we can use the attributesincludeSuperProperties and includeSuperFields. We can even instruct the annotation to create code in the constructor to call the super constructor of the super class with the properties. We must set the annotation attribute callSuper=true to make this happen.

```
import groovy.transform.TupleConstructor

@TupleConstructor()
class Person {
    String name
    List likes
    private boolean active = false }
```

```
def person = new Person('mrhaki', ['Groovy', 'Java'])

assert person.name == 'mrhaki'
assert person.likes == ['Groovy', 'Java']

person = new Person('mrhaki')

assert person.name == 'mrhaki'
assert !person.likes

// includeFields in the constructor creation.
import groovy.transform.TupleConstructor

@TupleConstructor(includeFields=true)
class Person {
    String name
    List likes
    private boolean active = false

    boolean isActivated() { active } }

def person = new Person('mrhaki', ['Groovy', 'Java'], true)

assert person.name == 'mrhaki'
assert person.likes == ['Groovy', 'Java']
assert person.activated

// use force attribute to force creation of constructor
// even if we define our own constructors.
import groovy.transform.TupleConstructor

@TupleConstructor(force=true)
class Person {
    String name
    List likes
    private boolean active = false

    Person(boolean active) {
        this.active = active
    }

    boolean isActivated() { active } }

def person = new Person('mrhaki', ['Groovy', 'Java'])

assert person.name == 'mrhaki'
assert person.likes == ['Groovy', 'Java']
assert !person.activated

person = new Person(true)

assert person.activated

// include properties and fields from super class.
import groovy.transform.TupleConstructor
```

```
@TupleConstructor(includeFields=true)
class Person {
    String name
    List likes
    private boolean active = false

    boolean isActivated() { active } }

@TupleConstructor(callSuper=true, includeSuperProperties=true,
includeSuperFields=true)
class Student extends Person {
    List courses }

def student = new Student('mrhaki', ['Groovy', 'Java'], true, ['IT'])

assert student.name == 'mrhaki'
assert student.likes == ['Groovy', 'Java']
assert student.activated
assert student.courses == ['IT']
```

Change Scope Script Variable with Field Annotation

The scope of a script variable changes once we add a type definition to the variable. Then the variable is no longer visible in methods of the script. To make a type variable visible in the entire script scope we add the annotation @Field. This way the variable has a type and can be used in methods of the script.

```
import groovy.transform.Field

String stringValue = 'I am typed without @Field.'
def i = 42
@Field String stringField = 'I am typed with @Field.'
counter = 0 // No explicit type definition.

def runIt() {
    try {
        assert stringValue == 'I am typed without @Field.'
    } catch (e) {
        assert e instanceof MissingPropertyException
    }
    try {
        assert i == 42
    } catch (e) {
        assert e instanceof MissingPropertyException
    }

    assert stringField == 'I am typed with @Field.'

    assert counter++ == 0 }

runIt()
```

```
assert stringValue == 'I am typed without @Field.'
assert stringField == 'I am typed with @Field.'
assert i == 42
assert counter == 1
```

Group and Count Elements in Collection or Map

Since Groovy 1.8 we can group items in a collection, map or array and count the number of elements in a group. We use a closure to define the keys for the grouping. Then the number of elements in the group are counted and that is the value of the grouping key.

```
def list = ['Groovy', 'Grails', 'Java']
assert list.countBy { it[0] } == ['G': 2, 'J': 1]  // 2 items start with G, 1 with J.

def numbers = [1,2,3,4,5] as Integer[]
assert numbers.countBy { it % 2 } == [0: 2, 1: 3]  // 2 even, 3 uneven numbers

def map = [user: 'mrhaki', city: 'Tilburg', age: 37]
assert map.countBy { key, value -> value.class } == [(String.class): 2,
(Integer.class): 1]  // 2 values of type String and 1 of type Integer
```

Count Occurrences in a Collection or Map

We can count occurrences of elements in a collection that apply to a condition given by a closure. The closure contains the condition the elements we want to count must apply to. We can use the new count() method since Groovy 1.8 and it can be applied to collections, maps and arrays.

```
def list = ['Groovy', 'Grails', 'Java']
assert list.count { it.startsWith('G') } == 2

def numbers = [1,2,3,4] as Integer[]
assert numbers.count { it > 2 } == 2

def map = [user: 'mrhaki', city: 'Tilburg', age: 37]
assert map.count { key, value -> key.size() == 3 } == 1
```

Easy toString Creation for Our Classes

Since Groovy 1.8 we can use the @ToString annotation for easy creation of a toString() method. We only have to add the annotation to our class definition and we get a nicely formatted output of the properties of our class.
We can even customize what we want to see in the output. We can see the names of the properties of our class in the toString() output if we add the attribute includeNames=true. By default only properties are added to the output, but we can include fields as well with the annotation attribute includeFields=true. To exclude properties we use the attribute excludes and assign the names of the properties we don't want in the output separated by a comma.
Finally we can include properties from a super class with the annotation atribute

includeSuper=true.

Let's see the @ToString in action with a few samples:

```
// Most simple implementation of toString.
import groovy.transform.ToString

@ToString
class Person {
    String name
    List likes
    private boolean active = false }

def person = new Person(name: 'mrhaki', likes: ['Groovy', 'Java'])

assert person.toString() == 'Person(mrhaki, [Groovy, Java])'

// includeNames to output the names of the properties.
import groovy.transform.ToString

@ToString(includeNames=true)
class Person {
    String name
    List likes
    private boolean active = false }

def person = new Person(name: 'mrhaki', likes: ['Groovy', 'Java'])

assert person.toString() == 'Person(name:mrhaki, likes:[Groovy, Java])'

// includeFields to not only output properties, but also field values.
import groovy.transform.ToString

@ToString(includeNames=true, includeFields=true)
class Person {
    String name
    List likes
    private boolean active = false }

def person = new Person(name: 'mrhaki', likes: ['Groovy', 'Java'])

assert person.toString() == 'Person(name:mrhaki, likes:[Groovy, Java], active:false)'

// Use includeSuper to include properties from super class in output.
import groovy.transform.ToString

@ToString(includeNames=true)
class Person {
    String name
    List likes
    private boolean active = false }

@ToString(includeSuper=true, includeNames=true)
class Student extends Person {
```

```
    List courses }

def student = new Student(name: 'mrhaki', likes: ['Groovy', 'Java'], courses:
['IT', 'Business'])

assert student.toString() == 'Student(courses:[IT, Business],
super:Person(name:mrhaki, likes:[Groovy, Java]))'

// excludes active field and likes property from output
import groovy.transform.ToString

@ToString(includeNames=true, includeFields=true, excludes='active,likes')
class Person {
    String name
    List likes
    private boolean active = false }

def person = new Person(name: 'mrhaki', likes: ['Groovy', 'Java'])

assert person.toString() == 'Person(name:mrhaki)'
```

Check if Maps are Equal

With Groovy 1.8 the equals() method is added to Map. This means we can check if maps are equals. They are equals if both maps have the same size, and keys and values are the same.

```
def map1 = [user: 'mrhaki', likes: 'Groovy', age: 37]
def map2 = [age: 37.0, likes: 'Groovy', user: 'mrhaki']
def map3 = [user: 'Hubert Klein Ikkink', likes: 'Groovy']

assert map1.equals(map2)
assert map1 == map2
assert !map1.equals(map3)
assert map2 != map3
```

Inject Logging Using Annotations

With Groovy 1.8 we can inject a log field into our classes with a simple annotation. In our class we can invoke method on the log field, just as we would do if we wrote the code to inject the log field ourselves. How many times have we written code like this Logger log = LoggerFactory.getLogger(<class>) at the top of our classes to use for example the Slf4j API? Since Groovy 1.8 we only have to add the@Slf4j annotation to our class and get the same result. **AND** each invocation of a log method is encapsulated in a check to see if the log level is enabled.

```
// File: LogSlf4j.groovy
// Add dependencies for Slf4j API and Logback
@Grapes([
    @Grab(group='org.slf4j', module='slf4j-api', version='1.6.1'),
    @Grab(group='ch.qos.logback', module='logback-classic', version='0.9.28')
])
```

```
import org.slf4j.*
import groovy.util.logging.Slf4j

// Use annotation to inject log field into the class.
@Slf4j
class HelloWorldSlf4j {
    def execute() {
        log.debug 'Execute HelloWorld.'
        log.info 'Simple sample to show log field is injected.'
    } }

def helloWorld = new HelloWorldSlf4j()
helloWorld.execute()
```

When we run this script we get the following output:

```
$ groovy LogSlf4j.groovy
21:20:02.392 [main] DEBUG HelloWorldSlf4j - Execute HelloWorld.
21:20:02.398 [main] INFO  HelloWorldSlf4j - Simple sample to show log field is
injected.
```

Besides an annotation for the Slf4j API other logging frameworks are supported with annotations:

Logging framework	Annotation
java.util.logging	@Log
Log4j	@Log4j
Apache Commons Logging	@Commons
Slf4j API	@Slf4j

Build JSON with JsonBuilder and Pretty Print JSON Text

Groovy 1.8 adds JSON support. We can build a JSON data structure with the JsonBuilder class. This class functions just like other builder classes. We define a hiearchy with values and this is converted to JSON output when we view the String value. We notice the syntax is the same as for a MarkupBuilder.

```
import groovy.json.*

def json = new JsonBuilder()

json.message {
    header {
        from('mrhaki')  // parenthesis are optional
        to 'Groovy Users', 'Java Users'
    }
    body "Check out Groovy's gr8 JSON support." }

assert json.toString() == '{"message":{"header":{"from":"mrhaki","to":["Groovy
```

```
Users","Java Users"]},"body":"Check out Groovy\'s gr8 JSON support."}}'

// We can even pretty print the JSON output
def prettyJson = JsonOutput.prettyPrint(json.toString())
assert prettyJson == '''{
    "message": {
        "header": {
            "from": "mrhaki",
            "to": [
                "Groovy Users",
                "Java Users"
            ]
        },
        "body": "Check out Groovy's gr8 JSON support."
    } }'''
```

Parse JSON with JsonSlurper

With Groovy 1.8 we can parse JSON text with the JsonSlurper class. We only have to feed the text to the parseText() method and we can the values are mapped to Maps and Lists. And getting the content then is very easy:

```
import groovy.json.*

def jsonText = '''
{
    "message": {
        "header": {
            "from": "mrhaki",
            "to": ["Groovy Users", "Java Users"]
        },
        "body": "Check out Groovy's gr8 JSON support."
    } }        '''

def json = new JsonSlurper().parseText(jsonText)

def header = json.message.header
assert header.from == 'mrhaki'
assert header.to[0] == 'Groovy Users'
assert header.to[1] == 'Java Users'
assert json.message.body == "Check out Groovy's gr8 JSON support."
```

Format Dates with TimeZone

Since Groovy 1.8.3 we can use an extra TimeZone parameter with the format() method of the Date class. This can be used to print a date/time for a particular timezone.

```
import static java.util.Calendar.*

def timeZone = TimeZone.getTimeZone('Europe/Amsterdam')
def otherTimeZone = TimeZone.getTimeZone('Australia/Canberra')
```

```
def cal = Calendar.instance
cal.set(year: 2011, month: OCTOBER, date: 20, hourOfDay: 12, minute: 30)

def date = cal.time
def dateFormat = 'yyyy/MM/dd HH:mm'

assert date.format(dateFormat, timeZone) == '2011/10/20 12:30'
assert date.format(dateFormat, otherTimeZone) == '2011/10/20 21:30'
```

Run Remote Scripts via URL

Since Groovy 1.8.3 we can run remote Groovy scripts. We can use an URL to refer to the Groovy script that we want to execute. This can be very useful to build a library of Groovy scripts and publish them on a web server or a code repository like Github. Then we can run those scripts by referring the scripts by URL.

```
// File:
// http://www.mrhaki.com/samples/remotesample.groovy

printMessage 'Remote Groovy script @ http://www.mrhaki.com/samples/
remotesample.groovy'

['Groovy', 'rocks'].each {
    print "${it.toUpperCase()} " }
println '!'

def printMessage(message) {
    def LINE_LENGTH = message.size() + 4
    println '*' * LINE_LENGTH
    println "* $message *"
    println '*' * LINE_LENGTH }
```

On a command line we run $ groovy http://www.mrhaki.com/samples/remotesample.groovy and we get the following output:

```
$ groovy http://www.mrhaki.com/samples/remotesample.groovy
**************************************************************************
* Remote Groovy script @ http://www.mrhaki.com/samples/remotesample.groovy *
**************************************************************************
GROOVY ROCKS !
$
```