

# Groovy

# Introduction

Literals, Lambdas, Operators, Syntax tidbits.

The GDK

Builders

MOP

AST transformations

GPars

# Introduction

Created in 2003

Dynamic Language with "optional typing"

Java super set

Batteries included

# Literals

```
list = [1, 2, 3]
```

```
map = [one:1, two:2]
```

```
println "1234" ==~ /\d+/
```

# Lambdas

```
say = {m -> println m}
```

```
say("hello")
```

```
println say instanceof Closure
```

```
def apply(name, action){  
  action(name)  
}
```

```
apply("hello"){v -> println v }
```

```
apply("hello"){println it}
```

```
class Owner {  
  def say = {m -> println m}  
}  
  
def o = new Owner()  
  
println o.say.metaClass.properties.collect  
  it.name  
}  
  
println o.say.owner == o  
  
println o.say.delegate
```



```
def say = {println m}  
say.delegate = [m:2]  
say()
```

```
person = [name:"bob",last:"builder"]  
  
person.with {// changing delegate  
  println "${name}-${last}"  
}
```

# Operators

```
value = String.&valueOf  
println value.class  
println value(true).class
```

```
person = [name:"bob", last:"builder", birth:n  
println person.birth ?: new Date() // elvis  
println person.birth?.toString() // safe na
```

# The Groovy JDK (GDK)

```
path = "/home/ronen/.zshrc"  
println new File(path).readLines().first()
```

```
multipleOfTwo = [1, 2, 3].collect{it * 2}
sum = multipleOfTwo.inject(0){acc, v ->
    acc += v
}
println multipleOfTwo
println sum
```



# Builders

```
import groovy.xml.*  
  
writer = new StringWriter()  
  
new MarkupBuilder(writer).books{  
    book(name:"groovy in action", isbn:"9781935182443")  
    book(name:"Joy of Clojure", isbn:"1935182641")  
}  
  
println writer
```

```
import java.awt.FlowLayout
import javax.swing.*
import groovy.swing.SwingBuilder

builder = new SwingBuilder()

builder.frame(title:"Swinging with Groovy!", size:[290,100]) {
    panel(layout:new FlowLayout()) {
        ["Groovy", "Ruby", "Python"].each{
            checkBox(text:it)
        }
    }
}.show()
```

```
public class NodeBuilder extends BuilderSupport {  
    protected void setParent(Object parent, Object child) {  
    }  
  
    protected Object createNode(Object name) {  
        return new Node(getCurrentNode(), name, new ArrayList());  
    }  
  
    protected Object createNode(Object name, Object value) {  
        return new Node(getCurrentNode(), name, value);  
    }  
  
    protected Object createNode(Object name, Map attributes) {  
        return new Node(getCurrentNode(), name, attributes, new Arr  
    }  
  
    protected Object createNode(Object name, Map attributes, Object  
        return new Node(getCurrentNode(), name, attributes, value);  
    }  
  
    protected Node getCurrentNode() {  
        return (Node) getCurrent();  
    }  
}
```

# Syntax tidbits

```
class Foo {  
  def a  
  def b  
}
```

```
println new Foo(a:1, b:2).dump()
```

```
def multiReturn(){  
    [1,2,3]  
}  
  
(a,b,c) = multiReturn()  
println a + b + c
```

MOP



```
public interface GroovyObject {  
    public Object invokeMethod(String methodName, Object args);  
    public Object getProperty(String propertyName);  
    public void setProperty(String propertyName, Object newValue);  
    public MetaClass getMetaClass();  
    public void setMetaClass(MetaClass metaClass);  
}
```

```
public abstract class GroovyObjectSupport implements GroovyObject
    // never persist the MetaClass
    private transient MetaClass metaClass;

    public GroovyObjectSupport() {
        this.metaClass = InvokerHelper.getMetaClass(this.getClass());
    }

    public Object getProperty(String property) {
        return getMetaClass().getProperty(this, property);
    }

    public void setProperty(String property, Object newValue) {
        getMetaClass().setProperty(this, property, newValue);
    }

    public Object invokeMethod(String name, Object args) {
        return getMetaClass().invokeMethod(this, name, args);
    }
    // ...
}
```

```
[String, List, "hello"].each {  
  println it.metaClass.name  
}
```

```
objectMethods = Object.methods.collect{it.name}
```

```
interestingMethods = "hello".metaClass.class.methods.grep {  
  !objectMethods.contains(it.name)  
}
```

```
println interestingMethods.collect {it.name}
```

```
String.metaClass.upper = {-> toUpperCase() }  
String.metaClass.foo = 1 // property  
  
println "bla".upper()  
println "bla".foo
```

```
class StringCategory {  
    static String lower(String string) {  
        return string.toLowerCase()  
    }  
}  
  
use(StringCategory) {  
    println "TeSt".lower()  
}
```

```
class Logable {  
    def info(message) {println message}  
}
```

```
class Chatty {  
    def talk(){  
        info("im alive!")  
    }  
}
```

```
@Mixin(Logable)  
class ChattyIntrusive {  
    def talk(){  
        info("im alive!")  
    }  
}
```

```
Chatty.mixin Logable  
new Chatty().talk()  
new ChattyIntrusive().talk()
```

# AST Transformations

```
import groovy.transform.Canonical
```

```
@Canonical class Foo {  
    def lets, play  
}
```

```
println new Foo("lets", "play").equals(new Foo("lets", "play"))
```



```
class Container {  
  @Lazy def expensive = {println "Im so precious!"}  
}  
c = new Container()  
println "nothing happend yet!"  
c.expensive()
```

```
@TimedInterrupt(value=1)  
import groovy.transform.TimedInterrupt  
while(true);
```

```

@Retention (RetentionPolicy.SOURCE)
@Target ([ElementType.METHOD])
@GroovyASTTransformationClass (["PrinterTransformation"])
public @interface Printer { }

@GroovyASTTransformation(phase = CompilePhase.SEMANTIC_ANALYSIS)
class PrinterTransformation implements ASTTransformation {

    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {

        if(!astNodes || !astNodes[0] || !(astNodes[1] instanceof MethodNode))
            return
        }

        MethodNode annotatedMethod = astNodes[1]
        def printStatment = new AstBuilder().buildFromSpec {
            expression {
                methodCall {
                    variable "this"
                    constant "println"
                    argumentList {
                        constant "Hey!"
                    }
                }
            }
        }

        List<Statement> statements = annotatedMethod.getCode().getStatements()
        statements.add(printStatment[0])
    }
}

```

```
class Bla {  
    @Printer  
    def action(){  
    }  
}  
  
new Bla().action()
```

# GPars

Parallel collections

Map reduce

Dataflow

Actors (will not cover)

Agents

```
import groovyx.gpars.GParsPool
GParsPool.withPool {

    def seq = (1..10).makeTransparent()
    def squareSum = seq.collect {
        println Thread.currentThread()
        it*it
    }.fold(0) {a,v-> a+v}
    println squareSum
}
```

```
import static groovyx.gpars.GParsPool.withPool

def words = "lets count"
println count(words)

def count(arg) {
    withPool {
        def grouped = arg.parallel.map{[it, 1]}.groupBy{it[0]}
        println grouped
        grouped.getParallel().map{
            it.value=it.value.size()
            it
        }.sort{-it.value}.collection
    }
}
```



# Dataflow

Operations (in Dataflow programs) consist of "black boxes" with inputs and outputs, all of which are always explicitly defined.

They run as soon as all of their inputs become valid, as opposed to when the program encounters them.

Whereas a traditional program essentially consists of a series of statements saying "do this, now do this", a dataflow program is more like a series of workers on an assembly line, who will do their assigned task as soon as the materials arrive.

This is why dataflow languages are inherently parallel; the operations have no hidden state to keep track of, and the operations are all "ready" at the same time.

No race-conditions

No live-locks

Deterministic deadlocks

Completely deterministic programs

BEAUTIFUL code.

```
import groovyx.gpars.dataflow.DataFlowVariable as WAIT
import static groovyx.gpars.dataflow.DataFlow.task

WAIT<Integer> x = new WAIT() // each of these is a logical channel
WAIT<Integer> y = new WAIT()
WAIT<Integer> z = new WAIT()

task {// a task is a logical work unit
    z << x.val + y.val // x.val,y.val blocks until they have been
}

task { x << 40 }
task { y << 2 }

println "z=${z.val}"
assert 42 == z.val
```

```

import static groovyx.gpars.dataflow.DataFlow.task
import groovyx.gpars.dataflow.DataFlowQueue

def generate(ch) {
    {-->
        for (i in (2..10000)) {
            ch << i
        }
    }
}

def filter(inChannel, outChannel, int prime) {
    {-->
        for (;;) {
            def number = inChannel.val
            if (number % prime != 0) {
                outChannel << number
            }
        }
    }
}

final DataFlowQueue origin = new DataFlowQueue() // multi assign
task generate(origin)
10.times {
    int prime = origin.val
    println prime
    def multiFiltered = new DataFlowQueue()
    task filter(origin, multiFiltered, prime)
    origin = multiFiltered
}

```

# Agents

```
import groovyx.gpars.agent.Agent  
num = new Agent([0])  
num.send {it[0] = 2}  
println num.val // blocks until all messages consumed
```

```
import groovyx.gpars.agent.Agent

numbers = new Agent([])
numbers.send {it.add 1}

t1 = Thread.start { numbers.send {it.add 2} }

t2 = Thread.start {
    numbers {it.add 3}
    numbers {it.add 4}
}

numbers.valAsync {println "Current members: $it"}
[t1, t2]*.join()
println numbers.val
```

Twitter: @narkisr

Blog: narkisr.com

Github: <https://github.com/narkisr>