What's new in Groovy 2.0

Guillaume Laforge
Groovy Project Manager
SpringSource / VMware
@glaforge

vmware®

Guillaume Laforge

- **Groovy Project Manager** at VMware
  - Initiator of the **Grails** framework
  - Creator of the **Gaelyk**

- Co-author of **Groovy in Action**

- Follow me on...
  - My blog: **http://glaforge.appspot.com**
  - Twitter: **@glaforge**
  - Google+: **http://gplus.to/glaforge**

Agenda (1/2)

- **What's in Groovy 1.8?**
  - Nicer DSLs with command chains
  - Runtime performance improvements
  - GPars bundled for taming your multicores
  - Closure enhancements
  - Builtin JSON support
  - New AST transformations

Agenda (2/2)

- **What's new in Groovy 2.0?**
  - Alignments with JDK 7
    - Project Coin (small language changes)
    - Invoke Dynamic support
      - Continued runtime performance improvements
  - Static type checking
  - Static compilation
  - Modularity

Command chains

- A grammar improvement allowing you to **drop dots** & **parens** when **chaining method calls**
  - an extended version of top-level statements like println

- Less dots, less parens allow you to
  - write more readable business rules
  - in almost plain English sentences
    - (or any language, of course)

Command chains

```
pull request  on github
```

Command chains

Alternation of method names

```
pull request  on github
```

Command chains

Alternation of method names

```
pull request  on github
```

and parameters (even named ones)

Command chains

```
pull request  on github
```

Command chains

Equivalent to:

```
pull(request).on(github)
```

Command chains

Command chains

```
// methods with multiple arguments (commas)
```

Command chains

```
// methods with multiple arguments (commas)
take coffee  with sugar, milk  and liquor
```

Command chains

```
// methods with multiple arguments (commas)
take coffee  with sugar, milk  and liquor

// leverage named-args as punctuation
```

Command chains

```
// methods with multiple arguments (commas)
take coffee  with sugar, milk  and liquor

// leverage named-args as punctuation
check that: margarita  tastes good
```

Command chains

```
// methods with multiple arguments (commas)
take coffee  with sugar, milk  and liquor

// leverage named-args as punctuation
check that: margarita  tastes good

// closure parameters for new control structures
```

Command chains

```
// methods with multiple arguments (commas)
take coffee  with sugar, milk  and liquor

// leverage named-args as punctuation
check that: margarita  tastes good

// closure parameters for new control structures
given {}  when {}  then {}
```

Command chains

```
// methods with multiple arguments (commas)
take coffee  with sugar, milk  and liquor

// leverage named-args as punctuation
check that: margarita  tastes good

// closure parameters for new control structures
given {}  when {}  then {}

// zero-arg methods require parens
```

Command chains

```
// methods with multiple arguments (commas)
take coffee  with sugar, milk  and liquor

// leverage named-args as punctuation
check that: margarita  tastes good

// closure parameters for new control structures
given {}  when {}  then {}

// zero-arg methods require parens
select all  unique() from names
```

Command chains

```
// methods with multiple arguments (commas)
take coffee  with sugar, milk  and liquor

// leverage named-args as punctuation
check that: margarita  tastes good

// closure parameters for new control structures
given {}  when {}  then {}

// zero-arg methods require parens
select all  unique() from names

// possible with an odd number of terms
```

Command chains

```
// methods with multiple arguments (commas)
take coffee  with sugar, milk  and liquor

// leverage named-args as punctuation
check that: margarita  tastes good

// closure parameters for new control structures
given {}  when {}  then {}

// zero-arg methods require parens
select all  unique() from names

// possible with an odd number of terms
take 3  cookies
```

Command chains

```
// methods with multiple arguments (commas)
take(coffee).with(sugar, milk).and(liquor)

// leverage named-args as punctuation
check that: margarita  tastes good

// closure parameters for new control structures
given {}  when {}  then {}

// zero-arg methods require parens
select all  unique() from names

// possible with an odd number of terms
take 3  cookies
```

Command chains

```
// methods with multiple arguments (commas)
take(coffee).with(sugar, milk).and(liquor)

// leverage named-args as punctuation
check(that: margarita).tastes(good)

// closure parameters for new control structures
given {}  when {}  then {}

// zero-arg methods require parens
select all  unique() from names

// possible with an odd number of terms
take 3  cookies
```

Command chains

```
// methods with multiple arguments (commas)
take(coffee).with(sugar, milk).and(liquor)

// leverage named-args as punctuation
check(that: margarita).tastes(good)

// closure parameters for new control structures
given({}).when({}).then({})

// zero-arg methods require parens
select all  unique() from names

// possible with an odd number of terms
take 3  cookies
```

Command chains

```
// methods with multiple arguments (commas)
take(coffee).with(sugar, milk).and(liquor)

// leverage named-args as punctuation
check(that: margarita).tastes(good)

// closure parameters for new control structures
given({}).when({}).then({})

// zero-arg methods require parens
select(all).unique().from(names)

// possible with an odd number of terms
take 3  cookies
```

Command chains

```
// methods with multiple arguments (commas)
take(coffee).with(sugar, milk).and(liquor)

// leverage named-args as punctuation
check(that: margarita).tastes(good)

// closure parameters for new control structures
given({}).when({}).then({})

// zero-arg methods require parens
select(all).unique().from(names)

// possible with an odd number of terms
take(3).cookies
```

GPars bundled

- **GPars** is bundled in the Groovy distribution

- GPars covers a wide range of parallel and concurrent paradigms
  - actors, fork/join, map/filter/reduce, dataflow, agents
  - parallel arrays, executors, STM, and more...

- And you can **use it from plain Java** as well!

- http://gpars.codehaus.org/

Closure enhancements

- **Closure annotation parameters**

- Some more functional flavor
  - composition
    - compose several closures into one single closure
  - trampoline
    - avoid stack overflow errors for recursive algorithms
  - **memoization**
    - remember the outcome of previous closure invocations
  - currying improvements

Closure annotation parameters

```
@Retention(RetentionPolicy.RUNTIME)
@interface Invariant {
    Class value() // a closure class
}

@Invariant({ number >= 0 })
class Distance {
    float number
    String unit
}

def d = new Distance(number: 10, unit: "meters")
def anno = Distance.getAnnotation(Invariant)
def check = anno.value().newInstance(d, d)
assert check(d)
```

Closure annotation parameters

```
@Retention(RetentionPolicy.RUNTIME)
@interface Invariant {
    Class value() // a closure class
}

@Invariant({ number >= 0 })
class Distance {
    float number
    String unit
}

def d = new Distance(number: 10, unit: "meters")
def anno = Distance.getAnnotation(Invariant)
def check = anno.value().newInstance(d, d)
assert check(d)
```

Poor-man's GContracts

Closure memoization

Closure memoization

```
def plus = { a, b -> sleep 1000; a + b }.memoize()
```

Closure memoization

```
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
```

Closure memoization

```
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
```

Closure memoization

```
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
assert plus(2, 2) == 4 // after 1000ms
```

Closure memoization

```
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
assert plus(2, 2) == 4 // after 1000ms
assert plus(2, 2) == 4 // return immediately
```

Closure memoization

```
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
assert plus(2, 2) == 4 // after 1000ms
assert plus(2, 2) == 4 // return immediately
```

Closure memoization

```
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
assert plus(2, 2) == 4 // after 1000ms
assert plus(2, 2) == 4 // return immediately

// at least 10 invocations cached
```

Closure memoization

```
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
assert plus(2, 2) == 4 // after 1000ms
assert plus(2, 2) == 4 // return immediately

// at least 10 invocations cached
def plusAtLeast = { ... }.memoizeAtLeast(10)
```

Closure memoization

```
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
assert plus(2, 2) == 4 // after 1000ms
assert plus(2, 2) == 4 // return immediately

// at least 10 invocations cached
def plusAtLeast = { ... }.memoizeAtLeast(10)
```

Closure memoization

```groovy
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
assert plus(2, 2) == 4 // after 1000ms
assert plus(2, 2) == 4 // return immediately

// at least 10 invocations cached
def plusAtLeast = { ... }.memoizeAtLeast(10)

// at most 10 invocations cached
```

Closure memoization

```groovy
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
assert plus(2, 2) == 4 // after 1000ms
assert plus(2, 2) == 4 // return immediately

// at least 10 invocations cached
def plusAtLeast = { ... }.memoizeAtLeast(10)

// at most 10 invocations cached
def plusAtMost = { ... }.memoizeAtMost(10)
```

Closure memoization

```groovy
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
assert plus(2, 2) == 4 // after 1000ms
assert plus(2, 2) == 4 // return immediately

// at least 10 invocations cached
def plusAtLeast = { ... }.memoizeAtLeast(10)

// at most 10 invocations cached
def plusAtMost = { ... }.memoizeAtMost(10)

// between 10 and 20 invocations cached
```

Closure memoization

```
def plus = { a, b -> sleep 1000; a + b }.memoize()

assert plus(1, 2) == 3 // after 1000ms
assert plus(1, 2) == 3 // return immediately
assert plus(2, 2) == 4 // after 1000ms
assert plus(2, 2) == 4 // return immediately

// at least 10 invocations cached
def plusAtLeast = { ... }.memoizeAtLeast(10)

// at most 10 invocations cached
def plusAtMost = { ... }.memoizeAtMost(10)

// between 10 and 20 invocations cached
def plusAtLeast = { ... }.memoizeBetween(10, 20)
```

Builtin JSON support

- Consuming
- Producing
- Pretty-printing

Builtin JSON support


```
import groovy.json.*

def payload = new URL(
    "http://github.../json/commits/...").text

def slurper = new JsonSlurper()
def doc = slurper.parseText(payload)

doc.commits.message.each { println it }
```

Builtin JSON support

```
import groovy.json.*

def json = new JsonBuilder()

json.person {
    name "Guillaume"
    age 35
    pets "Hector", "Felix"
}
println json.toString()
```

Builtin JSON support

```groovy
import groovy.json.*

def json = new JsonBuilder()

json.person {
    name "Guillaume"
    age 35
    pets "Hector", "Felix"
}
println json.toString()
```

```json
{
    "person": {
        "name": "Guillaume",
        "age": 35,
        "pets": [
            "Hector",
            "Felix"
        ]
    }
}
```

Builtin JSON support

```
import groovy.json.*

println JsonOutput.prettyPrint(
    '{"person":{"name":"Guillaume","age":35,' +
    '"pets":["Hector","Felix"]}}')
```

```
{
    "person": {
        "name": "Guillaume",
        "age": 35,
        "pets": [
            "Hector",
            "Felix"
        ]
    }
}
```

New AST transformations

- @Log
- @Field
- @AutoClone
- @AutoExternalizable
- @Canonical
  - @ToString
  - @EqualsAndHashCode
  - @TupleConstructor
- Controlling execution
  - @ThreadInterrupt
  - @TimedInterrupt
  - @ConditionalInterrupt
- @InheritConstructor
- @WithReadLock
- @WithWriteLock
- @ListenerList

@Log

- Four different loggers can be injected
  - @Log
  - @Commons
  - @Log4j
  - @Slf4j
- Possible to implement your own strategy

```
import groovy.util.logging.*

@Log
class Car {
    Car() {
        log.info 'Car constructed'
    }
}

def c = new Car()
```

@Log

- Four different loggers can be injected
  - @Log
  - @Commons
  - @Log4j
  - @Slf4j

Guarded w/ an if

- Possible to implement your own strategy

```
import groovy.util.logging.*

@Log
class Car {
    Car() {
        log.info 'Car constructed'
    }
}

def c = new Car()
```

Controlling code execution

- Your application may run user's code
  - what if the code runs in infinite loops or for too long?
  - what if the code consumes too many resources?

- 3 new transforms at your rescue
  - **@ThreadInterrupt**: adds Thread#isInterrupted checks so your executing thread stops when interrupted
  - **@TimedInterrupt**: adds checks in method and closure bodies to verify it's run longer than expected
  - **@ConditionalInterrupt**: adds checks with your own conditional logic to break out from the user code

@ThreadInterrupt

```
@ThreadInterrupt
import groovy.transform.ThreadInterrupt

while (true) {


    // eat lots of CPU
}
```

@ThreadInterrupt

```
@ThreadInterrupt
import groovy.transform.ThreadInterrupt

while (true) {
    if (Thread.currentThread().isInterrupted())
        throw new InterruptedException()
    // eat lots of CPU
}
```

@ToString

- Provides a default toString() method to your types
- Available annotation options
  - includeNames, includeFields, includeSuper, excludes

```groovy
import groovy.transform.ToString

@ToString
class Person {
    String name
    int age
}

println new Person(name: 'Pete', age: 15)
// => Person(Pete, 15)
```

@EqualsAndHashCode

- Provides default implementations for equals() and hashCode() methods

```
import groovy.transform.EqualsAndHashCode

@EqualsAndHashCode
class Coord {
    int x, y
}

def c1 = new Coord(x: 20, y: 5)
def c2 = new Coord(x: 20, y: 5)

assert c1 == c2
assert c1.hashCode() == c2.hashCode()
```

@TupleConstructor

- Provides a « classical » constructor with all properties
- Several annotation parameter options available

```groovy
import groovy.transform.TupleConstructor

@TupleConstructor
class Person {
    String name
    int age
}

def m = new Person('Marion', 4)

assert m.name == 'Marion'
assert m.age  == 4
```

@InheritConstructors

- Classes like Exception are painful when extended, as all the base constructors should be replicated

```
class CustomException extends Exception {
    CustomException()                          { super()       }
    CustomException(String msg)                { super(msg)    }
    CustomException(String msg, Throwable t) { super(msg, t) }
    CustomException(Throwable t)               { super(t)      }
}
```

@InheritConstructors

- Classes like Exception are painful when extended, as all the base constructors should be replicated

```
import groovy.transform.*

@InheritConstructors
class CustomException extends Exception {



}
```

Miscellaneous

- Compilation customizers
- Java 7 diamond operator
- Slashy and dollar slashy strings
- New GDK methods
- (G)String to Enum coercion
- Customizing the Groovysh prompt
- Executing remote scripts

What's in store for 2.0?

Groovy 2.0

- A **more modular** Groovy
- Java 7 alignements: **Project Coin**
  - binary literals
  - underscore in literals
  - multicatch
- JDK 7: **InvokeDynamic**
- **Static type checking**
- **Static compilation**

Groovy Modularity

- Groovy's « all » JAR weighs in at 6 MB

- **Nobody needs everything**
  - Template engine, Ant scripting, Swing UI building...

- Provide a smaller core
  - and several smaller JARs per feature

- Provide hooks for setting up DGM methods, etc.

The new JARs

- A smaller groovy.jar: 3MB

- Modules
  - console
  - docgenerator
  - groovydoc
  - groovysh
  - ant
  - bsf
  - jsr-223
  - jmx
  - sql
  - swing
  - servlet
  - templates
  - test
  - testng
  - json
  - xml

Extension modules

- Create your own module
  - contribute instance extension methods

```
package foo

class StringExtension {
    static introduces(String self, String name) {
        "Hi ${name), I’m ${self}"
    }
}

// usage: "Guillaume".introduces("Cédric")
```

Extension modules

- Create your own module
  - contribute instance extension methods

Same structure as Categories

```
package foo

class StringExtension {
    static introduces(String self, String name) {
        "Hi ${name), I'm ${self}"
    }
}

// usage: "Guillaume".introduces("Cédric")
```

Extension modules

- Create your own module
  - contribute class extension methods

```
package foo

class StaticStringExtension {
    static hi(String self) {
        "Hi!"
    }
}

// usage: String.hi()
```

Extension module descriptor

- Descriptor in: META-INF/services/ org.codehaus.groovy.runtime.ExtensionModule

```
moduleName = stringExtensions
moduleVersion = 1.0
// comma-separated list of classes
extensionClasses = foo.StringExtension
// comma-separated list of classes
staticExtensionClasses = foo.StaticStringExtension
```

Java 7 / JDK 7

- **Project Coin** and **InvokeDynamic**

Binary literals

- We had decimal, octal and hexadecimal notations for number literals
- We can now use binary representations too

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

byte aByte = 0b00100001
assert aByte == 33

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

Underscore in literals

- Now we can also add underscores in number literals for more readability

```
long creditCardNumber = 1234_5678_9012_3456L
long socialSecurityNumbers = 999_99_9999L
float 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

- One block for multiple exception caught
  - rather than duplicating the block

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

InvokeDynamic

- **Groovy 2.0 supports JDK 7's invokeDynamic**
  - compiler has a flag for compiling against JDK 7
  - might use the invokeDynamic backport for < JDK 7

- Benefits
  - **more runtime performance!**
    - at least as fast as current « dynamic » Groovy
  - in the long run, will allow us to get rid of code!
    - call site caching, thanks to MethodHandles
    - metaclass registry, thanks to ClassValues
    - will let the JIT inline calls more easily

Static Type Checking

- Goal: **make the Groovy compiler «grumpy»**!
  - and throw **compilation errors** (not at runtime)
- **Not everybody needs dynamic features all the time**
  - think Java libraries scripting
- Grumpy should...
  - tell you about your method or variable typos
  - complain if you call methods that don't exist
  - shout on assignments of wrong types
  - infer the types of your variables
  - figure out GDK methods
  - etc...

Typos in a variable or method

```
import groovy.transform.TypeChecked

void method() {}

@TypeChecked test() {
    // Cannot find matching method metthhoood()
    metthhoood()

    def name = "Guillaume"
    // variable naamme is undeclared
    println naamme
}
```

Typos in a variable or method

```
import groovy.transform.TypeChecked

void method() {}

@TypeChecked test() {
    // Cannot find matching method metthhoood()
    metthhoood()

    def name = "Guillaume"
    // variable naamme is undeclared
    println naamme
}
```

Compilation errors!

Typos in a variable or method

Annotation can be at class or method level

```
import groovy.transform.TypeChecked

void method() {}

@TypeChecked test() {
    // Cannot find matching method metthhoood()
    metthhoood()

    def name = "Guillaume"
    // variable naamme is undeclared
    println naamme
}
```

Compilation errors!

Wrong assignments

```
// cannot assign value of type... to variable...
int x = new Object()
Set set = new Object()

def o = new Object()
int x = o

String[] strings = ['a','b','c']
int str = strings[0]

// cannot find matching method plus()
int i = 0
i += '1'
```

Wrong assignments

```
// cannot assign value of type... to variable...
int x = new Object()
Set set = new Object()

def o = new Object()
int x = o

String[] strings = ['a','b','c']
int str = strings[0]

// cannot find matching method plus()
int i = 0
i += '1'
```

Compilation errors!

Wrong return types

```
// checks if/else branch return values
@TypeChecked
int method() {
    if (true) { 'String' }
    else { 42 }
}
// works for switch/case & try/catch/finally

// transparent toString() implied
@TypeChecked
String greeting(String name) {
    def sb = new StringBuilder()
    sb << "Hi " << name
}
```

Wrong return types

```
// checks if/else branch return values
@TypeChecked
int method() {
    if (true) { 'String' }
    else { 42 }
}
// works for switch/case & try/catch/finally

// transparent toString() implied
@TypeChecked
String greeting(String name) {
    def sb = new StringBuilder()
    sb << "Hi " << name
}
```

Compilation error!

Type inference

```
@TypeChecked test() {
    def name = "  Guillaume  "

    // String type infered (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 n
    }
}
```

Statically checked & dyn. methods

```
@TypeChecked
String greeting(String name) {
    // call method with dynamic behavior
    // but with proper signature
    generateMarkup(name.toUpperCase())
}

// usual dynamic behavior
String generateMarkup(String name) {
    def sw = new StringWriter()
    new MarkupBuilder(sw).html {
        body {
            div name
        }
    }
    sw.toString()
}
```

Instanceof checks

```
@TypeChecked
void test(Object val) {

    if (val instanceof String) {
        println val.toUpperCase()
    } else if (val instanceof Number) {
        println "X" * val.intValue()
    }

}
```

Instanceof checks

```
@TypeChecked
void test(Object val) {

    if (val instanceof String) {
        println val.toUpperCase()
    } else if (val instanceof Number) {
        println "X" * val.intValue()
    }

}
```

No need for casts

Instanceof checks

```
@TypeChecked
void test(Object val) {

    if (val instanceof String) {
        println val.toUpperCase()
    } else if (val instanceof Number) {
        println "X" * val.intValue()
    }

}
```

No need for casts

Can call String#multiply(int) from the Groovy Development Kit

Lowest Upper Bound

- Represents the lowest « super » type classes have in common
  - may be virtual (aka « non-denotable »)

```
@TypeChecked test() {
    // an integer and a BigDecimal
    return [1234, 3.14]
}
```

Lowest Upper Bound

- Represents the lowest « super » type classes have in common
  - may be virtual (aka « non-denotable »)

```
@TypeChecked test() {
    // an integer and a BigDecimal
    return [1234, 3.14]
}
```

Inferred return type:
List<Number & Comparable>

Lowest Upper Bound

- Represents the lowest « super » type classes have in common
  - may be virtual (aka « non-denotable »)


```
@TypeChecked test() {
    // an integer and a BigDecimal
    return [1234, 3.14]
}
```

Inferred return type:
List<Number & Comparable>

Flow typing

- Static type checking shouldn't complain even for bad coding practicies which work without type checks

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

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

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

Gotcha: static checking vs dynamic

- Type checking works at compile-time
  - adding @TypeChecked doesn't change behavior
    - do not confuse with static compilation

- Most dynamic features cannot be type checked
  - metaclass changes, categories
  - dynamically bound variables (ex: script's binding)

- However, compile-time metaprogramming works
  - as long as proper type information is defined

Gotcha: runtime metaprogramming

```
@TypeChecked
void test() {
    Integer.metaClass.foo = {}
    123.foo()
}
```

Gotcha: runtime metaprogramming

```
@TypeChecked
void test() {
    Integer.metaClass.foo = {}
    123.foo()
}
```

Not allowed:
metaClass property
is dynamic

Gotcha: runtime metaprogramming

```
@TypeChecked
void test() {
    Integer.metaClass.foo = {}
    123.foo()
}
```

Method not recognized

Not allowed: metaClass property is dynamic

Gotcha: explicit type for closures

```
@TypeChecked test() {
    ["a", "b", "c"].collect {
        it.toUpperCase() // Not OK
    }
}
```

- A « Groovy Enhancement Proposal » to address the issue

Gotcha: explicit type for closures

```
@TypeChecked test() {
    ["a", "b", "c"].collect { String it ->
        it.toUpperCase() // OK, it's a String
    }
}
```

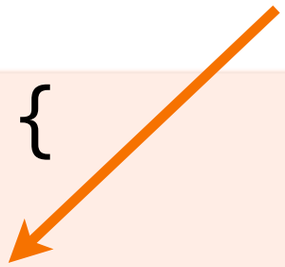
- A « Groovy Enhancement Proposal » to address the issue

Closure shared variables

```
@TypeChecked test() {
    def var = "abc"
    def cl = { var = new Date() }
    cl()
    var.toUpperCase()  // Not OK!
}
```

Closure shared variables

var assigned in the closure:
«shared closure variable»



```
@TypeChecked test() {
    def var = "abc"
    def cl = { var = new Date() }
    cl()
    var.toUpperCase()  // Not OK!
}
```

Closure shared variables

var assigned in the closure:
«shared closure variable»

Impossible to ensure the assignment really happens

```
@TypeChecked test() {
    def var = "abc"
    def cl = { var = new Date() }
    cl()
    var.toUpperCase()   // Not OK!
}
```

Closure shared variables

var assigned in the closure:
«shared closure variable»

Impossible to ensure the assignment really happens

```
@TypeChecked test() {
    def var = "abc"
    def cl = { var = new Date() }
    cl()
    var.toUpperCase()   // Not OK!
}
```

Only methods of the most specific compatible type (LUB) are allowed by the type checker

Closure shared variables

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

@TypeChecked test() {
    def var = new A()
    def cl = { var = new B() }
    cl()
    var.foo()    // OK!
}
```

Closure shared variables

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

@TypeChecked test() {
    def var = new A()
    def cl = { var = new B() }
    cl()
    var.foo()    // OK!
}
```

var is at least
an instance of A

Static Compilation

- Given your Groovy code can be type checked...
  we can as well compile it « statically »

- ie. **generate the same byte code as javac**

- Also interesting for those stuck in JDK < 7
  to benefit from performance improvements

Static Compilation: advantages

- You gain:
  - **Type safety**
    - thanks to static type checking
      - static compilation builds upon static type checking
  - **Faster code**
    - as close as possible to Java's performance
  - **Code immune to « monkey patching »**
    - metaprogramming badly used can interfere with framework code
  - Smaller bytecode size

Static Compilation: disadvantages

- But you loose:
  - Dynamic features
    - metaclass changes, categories, etc.
  - Dynamic method dispatch
    - although as close as possible to « dynamic » Groovy

Statically compiled & dyn. methods

```
@CompileStatic
String greeting(String name) {
    // call method with dynamic behavior
    // but with proper signature
    generateMarkup(name.toUpperCase())
}

// usual dynamic behavior
String generateMarkup(String name) {
    def sw = new StringWriter()
    new MarkupBuilder(sw).html {
        body {
            div name
        }
    }
    sw.toString()
}
```

What about performance?

- Comparisons between:
  - Java
    - Groovy **static compilation** (Groovy 2.0)
    - Groovy with **primitive optimizations** (Groovy 1.8+)
    - Groovy without optimizations (Groovy 1.7)

What about performance?

	Fibonacci	Pi (π) quadrature	Binary trees
Java	191 ms	97 ms	3.6 s
Static compilation	**197 ms**	**101 ms**	**4.3 s**
Primitive optimizations	360 ms	111 ms	23.7 s
No prim. optimizations	2590 ms	3220 ms	50.0 s

2.0 1.8 1.7

Summary

- Main themes of the Groovy 2.0 release
  - **Modularity**

  - Embark the JDK 7 enhancements
    - **Project Coin**
    - **Invoke Dynamic**

- Static aspects
  - **Static type checking**
  - **Static compilation**

Thank you!


spring
source
A division of VMware
Guillaume Laforge
Head of Groovy Development
Email: glaforge@gmail.com
Twitter: @glaforge
Google+: http://gplus.to/glaforge
Blog: http://glaforge.appspot.com

Questions & Answers

- Got questions, really?

Image credits

- Bugs bunny: http://storage.canalblog.com/63/56/517188/47634843.jpeg
- Pills: http://www.we-ew.com/wp-content/uploads/2011/01/plan-b-pills.jpg
- Chains: http://2.bp.blogspot.com/-GXDVqUYSCa0/TVdBsON4tdI/AAAAAAAAAW4/EgJOUmAxB28/s1600/breaking-chains5_copy9611.jpg
- Slurp: http://www.ohpacha.com/218-532-thickbox/gamelle-slurp.jpg
- Grumpy: http://mafeuilledechou.fr/__oneclick_uploads/2010/05/grumpy.jpg
- Modularity: http://php.jglobal.com/blog/wp-content/uploads/2009/11/modularity.jpg
- Agenda: http://www.plombiereslesbains.fr/images/stories/agenda.jpg
- Java 7: http://geeknizer.com/wp-content/uploads/java7.jpg
- Road Runner: http://newspaper.li/static/4ada046b7772f0612fec4e3840142308.jpg
- Porky: http://joshuadsandy.files.wordpress.com/2011/05/porky-pig-5.jpg
- Will E help: http://www.clipart-fr.com/data/clipart/looney/looney_004.jpg
- Coyote road runner: http://www.jenkle.com/wp-content/uploads/2010/12/coyote-wallpaper1.jpg
- Lube: http://images.motorcycle-superstore.com/ProductImages/OG/0000_Motul_Chain_Lube_Road_--.jpg