What’s New in Groovy 2.0?

Guillaume Laforge
Groovy project Manager
Guillaume Laforge

• Groovy Project Manager at VMware
  • Initiator of the Grails framework
  • Creator of the Gaelyk and Caelyf toolkits

• Co-author of Groovy in Action

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

• What’s new in Groovy 1.8?
  – Nicer DSLs with command chains
  – Runtime performance improvements
  – GPars bundled for taming your multicores
  – Closure enhancements
  – Built-in JSON support
  – New AST transformations
Agenda

• What’s cooking for 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
  – write more readable business rules
  – in almost plain English sentences
    • (or any language, of course)
Command chains

turn left then right
Command chains

Alternation of method names

turn left then right
Command chains

Alternation of method names

turn left then right

and parameters (even named ones)
Command chains

turn left then right
Command chains

Equivalent to:

\texttt{turn(left).then(right)}
Look Ma!
No parens, no dots!
Command chains

Before... we used to do...

take 2 pills, of: chloroquine, after: 6 hours
Before... we used to do...

`take 2.pills, of: chloroquine, after: 6.hours`
Before... we used to do...

take 2.pills, of: chloroquinine, after: 6.hours
Command chains

Before... we used to do...

take 2.pills, of: chloroquinine, after: 6.hours

Would call:

def take(Map m, Quantity q)
Command chains

Now, even less punctuation!

take 2 pills of chloroquine after 6 hours
Command chains

Now, even less punctuation!

\texttt{take(2).pills(of).chloroquinine (after).6(hours)}
// Java fluent API approach

class RegimenBuilder {
    ...
    
    def take(int n) {
        this.pills = n
        return this
    }
    
    def pills(String of) {
        return this
    }
    
    ...
}
Command chains

```python
// variable injection
def (of, after, hours) = /*...*/

// implementing the DSL logic
def take(n) {
    pills: { of ->
        chloroquine: { after ->
            ['6': { time -> }]
        }
    }
}

// -----------------------------
take 2 pills of chloroquine after 6 hours
```
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
// 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
• Significant runtime improvements for primitive type operations
  – classical Fibonacci example x13 faster!
  – closer to Java performance

• Some direct method calls on this

• More on performance later on
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!
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

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

@Invariant({
    number \geq 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

```java
@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.getAnnotationInvariant()
def check = anno.value().newInstance(d, d)
assert check(d)
```
Builtin JSON support

- Consuming
- Producing
- Pretty-printing
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 }
import groovy.json.*

def json = new JsonBuilder()

json.person {
    name "Guillaume"
    age 34
    pets "Hector", "Felix"
}
println json.toString()
import groovy.json.*

def json = new JsonBuilder()

json.person {
  name "Guillaume"
  age 34
  pets "Hector", "Felix"
}
println json.toString()

```json
{
  "person": {
    "name": "Guillaume",
    "age": 34,
    "pets": [
      "Hector",
      "Felix"
    ]
  }
}
```
import groovy.json.*

println JsonOutput.prettyPrint(
  '''{
    "person": {
      "name": "Guillaume",
      "age": 34,
      "pets": [
        "Hector",
        "Felix"
      ]
    }
  }''' +
  '''
)
New AST transformations

• @Log

• @Field

• @AutoClone

• @AutoExternalizable

• @Canonical
  – @ToString,
  – @EqualsAndHashCode,
  – @TupleConstructor

• Controlling the execution of your code
  – @ThreadInterrupt,
  – @TimedInterrupt,
  – @ConditionalInterrupt

• @InheritConstructor

• @WithReadLock

• @WithWriteLock

• @ListenerList

@glaforge
• Four different loggers can be injected
  – @Log, @Commons, @Log4j, @Slf4j
• Possible to implement your own strategy

```groovy
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
• Possible to implement your own strategy

```groovy
import groovy.util.logging.*

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

def c = new Car()
```

Guarded w/ an if
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?
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?
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

```groovy
import groovy.transform.ThreadInterrupt

while (true) {

    // eat lots of CPU

}
```
@ThreadInterrupt
import groovy.transform.ThreadInterrupt

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

@ThreadInterrupt

import groovy.transform.ThreadInterrupt

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

• Two optional annotation parameters available
  – checkOnMethodStart (true by default)
  – applyToAllClasses (true by default)
@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

```groovy
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

```java
text
import groovy.transform.TupleConstructor

@TupleConstructor
class Person {
    String name
    int age
}

def m = new Person('Marion', 3)

assert m.name == 'Marion'
assert m.age == 3
```
• One annotation to rule them all!
  – @Canonical mixes together
    • @ToString
    • @EqualsAndHashCode
    • @TupleConstructor

• You can customize behavior by combining @Canonical and one of the other annotations
@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);
  }
}

jeudi 29 mars 12
• Classes like Exception are painful when extended, as all the base constructors should be replicated

```java
import groovy.transform.*

@InheritConstructors
class CustomException extends Exception {

}
```
Miscelaneous

• 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
Compilation customizers

• Ability to apply some customization to the Groovy compilation process

• Three available customizers
  – ImportCustomizer
  – ASTTransformationCustomizer
  – SecureASTCustomizer

• But you can implement your own
Imports customizer

def configuration = new CompilerConfiguration()
def custo = new ImportCustomizer()
custo.addStaticStar(Math.name)
configuration.addCompilationCustomizers(custo)
def result = new GroovyShell(configuration)
    // import static java.lang.Math.*
    .evaluate(" cos PI/3 ")
Customizing the Groovysh prompt

```bash
glaforge-2:groovy-git glaforge$ export GROOVYSH_PROMPT="I ♥☆ ➜ "
glaforge-2:groovy-git glaforge$ groovysh
Groovy Shell (1.8.3, JVM: 1.6.0_26)
Type 'help' or '\h' for help.

I ♥☆ ➜ :000> println "nice custom prompt, dude!"
nice custom prompt, dude!
===> null
I ♥☆ ➜ :000> 
```
What’s cooking for 2.0?
Groovy 2.0 roadmap

- Java 7 alignements: **Project Coin**
  - binary literals
  - underscore in literals
  - multicatch

- JDK 7: **InvokeDynamic**

- Towards a **more modular** Groovy

- **Static type checking**

- **Static compilation**
Groovy 2.0 roadmap

• Java 7 alignements: Project Coin
  – binary literals
  – underscore in literals
  – multicatch

• JDK 7: InvokeDynamic

• Towards a more modular Groovy

• Static type checking

• Static compilation
Groovy 2.0 roadmap

• Java 7 alignments: Project Coin
  – binary literals
  – underscore in literals
  – multicatch

• JDK 7: InvokeDynamic

• Towards a more modular Groovy

• Static type checking

• Static compilation
Groovy 2.0 roadmap

• Java 7 alignments: Project Coin
  – binary literals
  – underscore in literals
  – multicatch

• JDK 7: InvokeDynamic

• Towards a more modular Groovy

• Static type checking

• Static compilation
Groovy 2.0 roadmap

• Java 7 alignments: Project Coin
  – binary literals
  – underscore in literals
  – multicatch

• JDK 7: InvokeDynamic

• Towards a more modular Groovy

• Static type checking

• Static compilation
Groovy 2.0 roadmap

• Java 7 alignments: **Project Coin**
  – binary literals
  – underscore in literals
  – multicatch

• JDK 7: **InvokeDynamic**

• Towards a **more modular Groovy**

• **Static type checking**

• **Static compilation**
Groovy 2.0 roadmap

• Java 7 alignements: Project Coin
  – binary literals
  – underscore in literals
  – multicatch

• JDK 7: InvokeDynamic

• Towards a more modular Groovy

• Static type checking

• Static compilation
Groovy 2.0 roadmap

• Java 7 alignments: Project Coin
  – binary literals
  – underscore in literals
  – multicatch

• JDK 7: InvokeDynamic

• Towards a more modular Groovy

• Static type checking

• Static compilation
Groovy 2.0 roadmap

- Java 7 alignments: Project Coin
  - binary literals
  - underscore in literals
  - multicatch

- JDK 7: InvokeDynamic

- Towards a more modular Groovy

- Static type checking

- Static compilation
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

```java
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

```java
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

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

• Groovy 2.0 supports JDK 7’s invokeDynamic
  – compiler will have 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
Groovy Modularity

• Groovy’s «all» JAR weighs in at 4MB

• 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.
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 your variable or method

```groovy
import groovy.transform.TypeChecked

void method() {}

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

  def name = "Guillaume"
  // variable naamme is undeclared
  println naamme
}
```
Typos in your variable or method

```groovy
import groovy.transform.TypeChecked

void method() {
  // Cannot find matching method method() method()
}

@TypeChecked test() {
  // Cannot find matching method method() method()
  def name = "Guillaume"
  // variable naamme is undeclared
  println naamme
}
```

Compilation errors!
Typos in your variable or method

```groovy
import groovy.transform.TypeChecked

void method() {}

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

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

Annotation can be at class or method level

Compilation errors!
Complain on wrong assignments

```java
// 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'
```
Complain on wrong assignments

```java
// 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!
Complain on wrong return types

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

// transparent toString() implied
String greeting(String name) {
  def sb = new StringBuilder()
  sb << "Hi " << name
}
Complain on wrong return types

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

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

Compilation error!
@TypeChecked test() {
    def name = " Guillaume "

    // String type inferred (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
    }
}
@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()
}
@TypeChecked

void test(Object val) {

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

}
Instanceof checks

@TypeChecked
t
```java
void test(Object val) {
    if (val instanceof String) {
        println val.toUpperCase()
    } else if (val instanceof Number) {
        println "X" * val.intValue()
    }
}
```

No need for casts
@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»)

```java
@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»)

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

Inferred return type: List<Number>
Flow typing

• Static type checking shouldn’t complain even for bad coding practices which work w/o type checks

```java
@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!
}
```
Gotchas: 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
Gotchas: runtime metaprogramming

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

Not allowed: metaClass property is dynamic
@TypeChecked

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

Method not recognized

Not allowed: metaClass property is dynamic
Gotchas: closures need explicit types

A «Groovy Enhancement Proposal» to address this issue

```java
@TypeChecked
  test()
  {
    @TypeChecked
      ["a", "b", "c"].collect {
        it.toUpperCase() // Not OK
      }
  }
```
Gotchas: closures need explicit types

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

- A «Groovy Enhancement Proposal» to address this issue
Closure shared variables

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

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

var assigned in the closure: «shared closure variable»
Closure shared variables

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

var assigned in the closure:  
«shared closure variable»

Impossible to ensure  
the assignment  
really happens
Closure shared variables

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

- var assigned in the closure: «shared closure variable»
- Impossible to ensure the assignment really happens
- 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
Static Compilation: advantages

• You gain:
  
  – Type safety
    • thanks to static type checking
      – static compilation builds upon static type checking
  
  – Faster code
    • as close to Java’s performance as possible
  
  – 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 emulated as close as possible to «dynamic» Groovy
@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
    • ie. Groovy 2.0

  – Groovy with primitive optimizations
    • ie. since Groovy 1.8

  – Groovy without optimizations
    • ie. Groovy 1.7 and before
What about performance?

<table>
<thead>
<tr>
<th></th>
<th>Fibonacci</th>
<th>Pi (π) quadrature</th>
<th>Binary trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>140 ms</td>
<td>93 ms</td>
<td>4.5 s</td>
</tr>
<tr>
<td>Static compilation</td>
<td>150 ms</td>
<td>220 ms</td>
<td>7.6 s</td>
</tr>
<tr>
<td>Primitive optimizations</td>
<td>270 ms</td>
<td>110 ms</td>
<td>29.6 s</td>
</tr>
<tr>
<td>No prim. optimizations</td>
<td>3800 ms</td>
<td>3.8 s</td>
<td>50.0 s</td>
</tr>
</tbody>
</table>
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
Summary

ROCKS!
Guillaume Laforge
Head of Groovy Development

Email: glaforge@gmail.com
Twitter: @glaforge
Google+: http://gplus.to/glaforge
Blog: http://glaforge.appspot.com
Q&A

Got questions, really?

33rd Degree Conference
Conference for Java Masters
19-21 March 2012
Kraków, Poland
Image credits

- Iceberg: http://hqworld.net/gallery/data/media/20/tip_of_the_iceberg.jpg
- Bicycle: http://drunkcyclist.com/gallery/main.php?q2_view=core.DownloadItem&g2_itemId=2131&g2_serialNumber=2
- Chains: http://2.bp.blogspot.com/-GXDVqUYSCa0/TVdBsON4tdI/AAAAAAAAAW4/EgJOUmAxB28/s1600/breaking-chains5_copy9611.jpg
- Cooking: http://www.flickr.com/photos/eole/449958332/sizes/l/
- Oui nide iou: http://desencycopedie.wikia.com/wiki/Fichier:Superdupont_we_need_you.jpg
- Guitar: http://www.lelitteraire.com/IMG/breven323.jpg
- Trampoline: http://www.fitness-online.fr/images/trampoline-1-m.jpg
- Grumpy: http://mafeuilledechou.fr/_oneclick_uploads/2010/05/grumpy.jpg
- Agenda: http://www.plombiereslesbains.fr/images/stories/agenda.jpg