Rethinking API design with traits in Groovy

by Cédric Champeau

About me

Pivotal employee

Core Groovy committer

- Compilation configuration
- Static type checking & Static compilation
- Traits, new template engine, ...

Groovy in Action 2 co-author

Misc OSS contribs (Gradle plugins, deck2pdf, jlangdetect, ...)

Social (or why to use #groovylang)

CLEAR 4G Suivre
@CLEAR

Happy Birthday to the LAVA LAMP! #Groovy | shout.lt/lls2

00:13 - 4 Sept 2013

Today Marks Lava Lamps' 50th Year of Lighting Up Your Parents'...

Believe it or not, that beloved 1960s relic that once cast a warm, unnatural glow on hippies everywhere is officially 50 years old today. And even though you may still see lava lamps churning away in...

Gizmodo @Gizmodo

2 RETWEETS

Sean Gilligan Suivre
@msgilligan

Groovy 2,2 on the front page of Hacker News. Give it some love Groovsters! news.ycombinator.com/item?id=6756742 #groovylang

22:12 - 18 Nov 2013

Life without traits
(or why it's primarily a static issue)

Life without traits, pre Java 8

- Contracts are defined through interfaces
- Interfaces are **mandatory** in a static type system
- Base behavior defined through abstract classes
- To implement multiple contracts, you need multiple abstract classes
- … or you can use duck typing in Groovy

Life without traits, pre Java 8

- Even in Groovy...
  - Problems with inheritance
  - Hard to determine whether a class explicitly or implicitly implements a contract

@Mixin

- Deprecated in Groovy 2.3
- Combines static and dynamic features to achieve runtime mixins

```
class Cat {
    void meow() { println 'Meow!' }
}
@Mixin(Cat)
class YoutubeDog {
    void bark() { println 'Woof!' }
}
def dog = new YoutubeDog()
dog.meow()
dog.bark()
```

@Mixin

- Mixed in class is **not** an instance of the mixin
- Problems with inheritance
- Memory
- Difference between the mixed instance and the mixin instance
- Buggy (really)

Delegation

- Delegation vs inheritance
  - Delegation keeps concerns separate
  - Cleaner responsibility model
  - Not always suitable
    - Foo “is a Bar” → inheritance
    - Foo “acts as a Bar” → composition
  - Optionally requires interfaces

@Delegate : composition model

```groovy
/* An API-limited version of a Stack */
class StackOnly<T> {
    Stack<T> delegate = new Stack<>()

    void push(T e) { delegate.push(e) }
    T pop() { delegate.pop() }
}

def t = new StackOnly<String>()
t.push 'a'
t.push 'b'

assert t.pop() == 'b'
```

@Delegate : composition model

```groovy
/* An API-limited version of a Stack */
class StackOnly<T> {
    @Delegate(interfaces=false, includes=['push','pop'])
    Stack<T> delegate = new Stack<>()
}

def t = new StackOnly<String>()
t.push 'a'
t.push 'b'

assert t.pop() == 'b'
assert t.get(0) == 'a' // will fail because get is not delegated
```

@Delegate : composition model

- Excellent fit when the delegating object requires a **subset** of the features of the delegate
- Delegation + interfaces used as the poor man's multiple inheritance solution

Groovy specific : Extension methods

- URL#getText(String encoding)
- Allows adding behavior to **existing** classes
- Supported by @CompileStatic
- Add your own thanks to extension modules
  - See http://docs.groovy-lang.org/latest/html/documentation/#_extension_modules
- **Not** visible from Java

Java 8 default methods

- Some kind of traits
- An interface
  - With default implementation

```
public interface Greeter {
    default void greet() {
        System.out.println("Hello!");
    }
}

// Usable from Groovy
class DefaultGreeter implements Greeter {
}
```

Java 8 default methods

- Stateless
  - No state = No diamond issue
- Virtual

```
class VirtualGreeter implements Greeter {
    @Override
    void greet() {
        println "Virtual Hello!"
    }
}
```

- Backward compatible
- Groovy does **not** support writing default methods

Traits in Groovy

Traits

- Similar to Java 8 default methods
- Supported in JDK 6, 7 and 8
- Stateful
- Composable
- Not really virtual...

Traits : the basics

- Declared through the “trait” new keyword
- Requires Groovy 2.3+

```
trait Flying {
    void fly() { println "I'm flying!" }
}
trait Quacking {
    void quack() { println 'Quack!'    }
}
class Duck implements Flying, Quacking {}
```

Traits : the basics

- Override like any other interface
- Compatible with @CompileStatic

```
@CompileStatic
class Duck implements Flying, Quacking {
    @Override
    void fly() {
        println "Duck flying!"
        quack()
    }
}
```

Traits : the basics

- Stateful

```
trait Geolocalized {
    double longitude
    double latitude
    String placeName
}
class City implements Geolocalized {}

def city = new City(longitude: 32.803468, latitude: -96.769879, placeName: 'Dallas')
```

Traits : the basics

- Multiple stateful traits allowed!
  - Major difference with inheritance

```
trait Inhabited {
    int population
}
class City implements Geolocalized, Inhabited {}
def city = new City(
        longitude: 32.803468,
        latitude: -96.769879,
        placeName: 'Dallas',
        population: 1_280_000
)
println city.population
```

Traits : first insight...

- Traits can be designed as small bricks of behavior
- Classes can be composed with traits
- Increases reusability of components

Traits : beyond basics

- A trait may inherit from another trait

```
trait Named {
    String name
}
trait Geolocalized extends Named {
    long latitude
    long longitude
}
class City implements Geolocalized {}
```

Traits : beyond basics

- A trait may inherit from multiple traits

```
trait WithId {
    Long id
}
trait Place implements Geolocalized, WithId {}

class City implements Place {}
```

Traits : beyond basics

- A trait can define abstract methods

```
trait Polite {
    abstract String getName()
    void thank() {
        println "Thank you, my name is $name"
    }
}
class Person implements Polite {
    String name
}

def p = new Person(name: 'Rob')
p.thank()
```

Traits : beyond basics

- Why *implements*?
  - From Java
    - a trait is only visible as an *interface*
    - default methods of traits are **not** default methods of interface
  - From Groovy
    - A trait can be seen as an interface on steroids
    - A trait without default implementation is nothing but an interface

Traits : conflicts

- What if two traits define the same method ?

```
trait Worker {
    void run() {
        println "I'm a worker!"
    }
}
trait Runner {
    void run() {
        println "I'm a runner"
    }
}
class WhoAmI implements Worker, Runner {}
```

Traits : conflicts

- Conflict resolution rule: last trait wins

```
class WhoAmI implements Worker, Runner {}

def o = new WhoAmI()
o.run() // prints "runner"
```

```
class WhoAmI implements Runner, Worker {}

def o = new WhoAmI()
o.run() // prints "worker"
```

Traits : explicit conflict resolution

- You can choose which method to call

```groovy
trait Lucky {
    String adj() { 'lucky'}
    String name() { 'Alice' }
}
trait Unlucky {
    String adj() { 'unlucky'}
    String name() { 'Bob' }
}

class LuckyBob implements Lucky, Unlucky {
    String adj() {
        Lucky.super.adj() // we select the "adj" implementation from Lucky
    }
    String name() {
        Unlucky.super.name() // and the "name" implementation from Unlucky
    }
    String toString() { "${adj()} ${name()}"}
}

def l = new LuckyBob()
assert l.toString() == 'lucky Bob'
```

Traits : what's *this*?

- “this” represents the trait'ed class

```
trait WhoIsThis {
    def self() { this }
}

class Me implements WhoIsThis {}

def me = new Me()
assert me.is(me.self())
```

Traits : runtime coercion

- Groovy way of implementing interfaces

```
interface MyListener {
    void onEvent(String name)
}

def explicit = { println it } as MyListener
MyListener implicit = { println it }

explicit.onEvent 'Event 1'
implicit.onEvent 'Event 2'
```

Traits : runtime coercion

- Same can be done with traits

```
trait Greeter {
    abstract String getName()
    void greet() { println "Hello $name" }
}

def greeter      = { 'Dan' } as Greeter
Greeter greeter2 = { 'Dan' }

greeter.greet()
greeter2.greet()
```

Traits : runtime coercion

- Support for multiple traits at runtime

```
class PoorObject {
    void doSomething() { println "I can't do much" }
}
trait SuperPower {
    void superPower() { println 'But I have superpowers!' }
}
trait Named {
    String name
}
def o = new PoorObject()
def named = o as Named
named.name = 'Bob'   // now can set a name
named.doSomething()  // and still call doSomething
def powered = o as SuperPower
powered.superPower() // now it has superpowers
powered.doSomething() // it can do something
def superHero = o.withTraits(SuperPower, Named) // multiple traits at once!
superHero.name = 'SuperBob' // then you can set a name
superHero.doSomething() // still call doSomething
superHero.superPower() // but you have super powers too!
```

Traits : who's fastest?

- Courtesy of Erik Pragt (@epragt)

```
trait NoStackTrace {
    Throwable fillInStackTrace() {
        return this
    }
}

class ExpensiveException extends RuntimeException { }
class CheapException extends RuntimeException implements NoStackTrace {}


def e1 = new CheapException()
def e2 = new ExpensiveException()
def e3 = new ExpensiveException() as NoStackTrace
```

Traits : who's fastest?

- Courtesy of Erik Pragt (@epragt)

	user	system	cpu	real
Cheap Exception	42	0	42	43
Expensive Exception	6722	3	6725	6734
Make expensive exception cheap	14982	7	14989	15010

See https://gist.github.com/bodiam/48a06dc9c74a90dfba8c

Traits : not virtual

- Methods from traits are “copied” where applied

```
trait HasColor {
    String getColor() { 'red' }
}

class Colored implements HasColor {}

def c = new Colored()
assert c.color == 'red'

class Colored2 implements HasColor {
    String getColor() { 'blue' }
}

def c2 = new Colored2()
assert c2.color == 'blue'
```


Takes precedence

Traits : not virtual

- Methods from traits are “copied” where applied

```
class Colored3 {
    String getColor() { 'yellow' }
}
class Colored4 extends Colored3 implements HasColor {}

def c4 = new Colored4()
assert c4.color == 'red'
```

Takes precedence

- Traits can be used to share overrides!

Traits : the meaning of “*super*”

- *super* has a special meaning in traits

```
Lucky.super.foo()
```

Qualified super

- But *super* can also be used for **chaining** traits

```
interface MessageHandler {
    void onMessage(String tag, Map<?,?> payload)
}

trait DefaultMessageHandler implements MessageHandler {
    void onMessage(String tag, Map<?, ?> payload) {
        println "Received message [$tag]"
    }
}
```

Traits : the meaning of “*super*”

```
class MyHandler implements DefaultMessageHandler {}

def h = new MyHandler()
h.onMessage("event",[:])
```

Traits : the meaning of “*super*”

```
// A message handler which logs the message and
// passes the message to the next handler in the chain
trait ObserverHandler implements MessageHandler {
    void onMessage(String tag, Map<?, ?> payload) {
        println "I observed a message: $tag"
        if (payload) {
            println "Payload: $payload"
        }
        // pass to next handler in the chain
        super.onMessage(tag, payload)
    }
}

class MyNewHandler implements DefaultMessageHandler, ObserverHandler {}
def h2 = new MyNewHandler()
h2.onMessage('event 2', [pass:true])
```

Traits : chaining with runtime coercion

```
class BaseHandler implements MessageHandler {
    @Override
    void onMessage(final String tag, final Map<?, ?> payload) {
        println "Received [$tag] from base handler"
    }
}

trait UpperCaseHandler implements MessageHandler {
    void onMessage(String tag, Map<?, ?> payload) {
        super.onMessage(tag.toUpperCase(), payload)
    }
}
trait DuplicatingHandler implements MessageHandler {
    void onMessage(String tag, Map<?, ?> payload) {
        super.onMessage(tag, payload)
        super.onMessage(tag, payload)
    }
}

def h = new BaseHandler()
h = h.withTraits(UpperCaseHandler, DuplicatingHandler, ObserverHandler)
h.onMessage('runtime', [:])
```

Traits : AST transformations

Traits : AST transformations

- Unless stated differently, **consider AST xforms incompatible**
- Both for **technical** and **logical** reasons
  - Is the AST xform applied on the trait or should it be applied when the trait is applied?

Traits : AST transformations

- This works

```
@CompileStatic
trait StaticallyCompiledTrait {
    int x
    int y
    int sum() { x + y }
}
```

Traits : AST transformations

- This works too

```
trait Delegating {
    @Delegate Map values = [:]
}

class Dynamic implements Delegating {}
new Dynamic().put('a','b')
```

Traits : AST transformations

- But this doesn't

```
trait Named {
    String name
}

@ToString
// ToString misses the "name" property
class Foo implements Named {}
```

API design?

Traits : rethinking API design conclusion

- Traits let you think about components
- Traits can be seen as generalized delegation
- Methods can accept traits as arguments
- Traits are stackable
- Traits are visible as interfaces from Java

Traits : API independent extension with DSL

- DSL can be used to augment an API
  - No need to subclass the base API

```
trait ListOfStrings implements List<String> {
    int totalLength() {
        sum { it.length() }
    }
}

trait FilteringList implements List<String> {
    ListOfStrings filter() {
        findAll { it.contains('G') || it.contains('J') } as ListOfStrings
    }
}

def list = ['Groovy', 'rocks', 'the', 'JVM']
println list.withTraits(FilteringList).filter().totalLength()
```

Conclusion

- Traits extend the benefit of interfaces to concrete classes
- Traits do not suffer the diamond problem
- Traits allow a new set of API to be written
- APIs can be augmented without modifications or extension

http://docs.groovy-lang.org/2.3.6/html/documentation/core-traits.html

@CedricChampeau

melix

http://melix.github.io/blog