# Introduction to Groovy

## Part III – More Groovy Grooviness

GROOVY… WHERE LESS REALLY IS MORE

# Overview

Quick Review of Part II

More on Groovy SDK

Traits

Look ahead to Part IV

# Quick Review of Part II

Syntactical Sugar

Groovy truth

Groovy operators

Closures, A First Look

Intro to Collections

# More on Groovy SDK

Groovy Operator Overloading

More java.util Collections JDK enhancements

java.lang.String JDK enhancements

java.lang.Integer JDK enhancements

# Groovy Operator Overloading

Groovy provides an elegant way to override many operators

Each operator that can be overloaded has a corresponding method named like the operator

- + is overloaded with plus()
- - is overloaded with minus()
- ++ is overloaded with next()
- -- is overloaded with previous()
- << is overloaded with leftShift()
- >> is overloaded with rightShift()
- [] is overloaded with getAt and putAt; i.e. a[b] and a[b]=c
- Many more!

# Groovy Operator Overloading

All we have to do is implement the operator method in our class to give meaning to the operator

To support chaining, the return from the method is the reference to the object with the operator behavior

# Groovy Operator Overloading

```
@groovy.transform.ToString
class Pizza {
  String name
  List<String> toppings = []
  Pizza leftShift(String topping) {
    toppings.add(topping)
    this
  }
}
Pizza p = new Pizza(name: "Simply Yummy")
p << 'Cheese' << 'Mushroom' << 'Pepperoni' << 'Sausage'
```

# Groovy Operator Overloading

Some examples from the Groovy SDK

- All collections have +, -, and <<
- Number implements +, -, *, / and ** (power) operators
  - BigDecimals get these behaviors
- Dates
  - Have + and – support, as well as ++ and -- for adding/subtracting days from a date
  - Have a getAt and putAt method for getting / setting fields

# More on collections enhancements

Spread operator (*) works across elements of a collection

```
List list = [‘My’, ‘name’, ‘is’, ‘Jack’]
assert list.size() == 4
assert list*.size() == [2, 4, 2, 4]
```

# More on collections enhancements

Given this List:

**List list = [1,2,3,4,5]**

We can reverse the list:

**list.reverse() == [5,4,3,2,1]**

We can join the list elements

**list.join('-') == '1-2-3-4-5'**

We can find the first and last elements

**list.first() == 1**

**list.last() == 5**

We can even find all permutations of a list

**list.permutations == [[1, 2, 4, 5, 3], [5, 3, 2, 4, 1], [1, 3, 2, 5, 4], [3, 1, 2, 4, 5], [3, 4, 2, 5, 1],,…]**

# More on collections enhancements

Given this collection:

```
List list = [‘My’, ‘name’, ‘is’, ‘Jack’]
```

Collect method takes a closure and applies it to each element yielding a transformed element:

```
List newList = list.collect { it.toUpperCase() }
assert newList == [‘MY’, ‘NAME’, ‘IS’, ‘JACK’]
```

# More on collections enhancements

Given this collection:

```
List list = [‘My’, ‘name’, ‘is’, ‘Jack’]
```

We can slice into the list using subscripts and ranges

```
assert list[3] == [‘Jack’]
assert list[1..3] == [‘name’, ‘is’, ‘Jack’]
```

# More on collections enhancements

Collections also have

- find(Closure) – find first element matching condition in closure
- findAll(Closure) - find all elements matching condition in closure
- inject method taking a closure with two arguments, the initial value and the element value

Combining these, we can perform a Filter/Map/Reduce operation with findAll, collect and inject methods

First, let’s see this in Java

# More on collections enhancements

```
// LineItem has boolean taxable, int qty and BigDecimal price
public BigDecimal calculateTaxableTotal(List<LineItem> items) {
  BigDecimal sum = BigDecimal.ZERO;
  for(LineItem item: items) {
    if(item.isTaxable()) {
      sum = sum.add(item.getPrice().multiply(item.getQty()));
    }
  }
  return sum;
}
```

# More on collections enhancements

```
// LineItem has boolean taxable, int qty and BigDecimal price
BigDecimal calculateTaxableTotal(List<LineItem> items) {
  items.findAll{ it.taxable }
       .collect{ it.price * it.qty }
       .inject { sum, cost -> sum += cost }
}
```

# Some String enhancements

We've seen String interpolation

```
"Hello $name"
```

And Heredocs:

```
'''
<customer>
  <name>Acme</name>
  <id>1234</id>
</customer>
'''
```

# Some String enhancements

There are more … a lot more!

- `capitalize : 'hello'.capitalize() == 'Hello'`
- `reverse: 'hello'.reverse() == 'olleh'`
- `isXXX: '1.0'.isInteger() == false`
- `'1.0'.isDouble() == true`
- `execute: 'cmd.exe /c dir'.execute() // lists dir`
- `center: ' banner '.center(20, '*')`
  `== '****** banner ******'`
- `padLeft/Right: '123'.padLeft(6, '_') == '___123'`
  `'1234'.padLeft(6, '_') == '__1234'`

# Some Integer Enhancements

times: 10.times{ print “$it “ } // prints 0 1 2 3 4 5 6 7 8 9

power: 2**16 == 65536

upto: 1.upto(10) { print "$it " } // prints 1 2 3 4 5 6 7 8 9 10

# Traits

Traits are new in Groovy 2.3

Declared like an interface or class, except with the ***trait*** keyword

```
trait Sailing {
  void sail() { println "I'm sailing!" }
}
```

They kind of feel like classes

- TraitA can extend TraitB and implement interfaces X, Y, Z
- They can have state
- They can declare abstract methods

However, they're used by classes like interfaces

- ClassA implements TraitA, TraitB

# Traits

## Traits are new in Groovy 2.3

```
trait Flying {
  String airplaneType
  void fly() { println "I'm flying a $airplaneType!"
}
trait Sailing {
  void sail() { println "I'm sailing!" }
}
class Person implements Flying, Sailing {}


Person p = new Person(airplaneType: 'Boeing 737')
p.fly()  // prints I'm flying a Boeing 737
p.sail() // prints I'm sailing!
```

# Traits

What if there's a method clash? Last declared implements, wins

```
trait SailingA {
  void sail() { println "I'm sailing A!" }
}

trait SailingB {
  void sail() { println "I'm sailing B!" }
}

class Person implements SailingA, SailingB {}


Person p = new Person()
p.sail() // prints I'm sailing B
// We can override this ordering by overriding method in Person
// and calling SailingA.super.sail()
```

# Traits

## Classes can override trait methods

```
trait Boating {
  void sail() { println "I'm sailing!" }
  void row() { println "I'm rowing!" }
}

class Person implements Boating {
  void sail() { println "I'm sailboating!" }
}


Person p = new Person()
p.sail() // prints from class: I'm sailboating!
p.row() // prints from trait: I'm rowing!
```

# Traits

In Groovy (not Java), we can implement traits at runtime rather than compile time

```
trait Named {
  String name
  void sayName() {
    println "Hello $name"
  }
}
trait Aged {
  String age
  void sayAge() {
    println "Age: $age"
  }
}
```

```
class Person {}

def p1 = new Person() as Named
p1.name = 'Jack'
p1.sayName() // prints Hello Jack

def p2 = p1.withTraits Named, Aged
p2.name = 'Jill'
p2.age = 29
p2.sayName() // prints Hello Jack
p2.sayAge()  // prints Age: 54
```

# Traits

## A few other points

- Traits are compiled into the code
- Compatible with @CompileStatic
- Traits can be chained together (think Servlet Filters) to pass processing through to implemented traits earlier in the chain
- A method on TraitC can call super.x() to pass control to prior trait, TraitB
- If TraitB doesn't have x(), super search will continue to TraitA, then the Bar class

```
class Foo extends Bar implements TraitA, TraitB, TraitC
```

# What's Next

At our next meeting we will have Part III of this Groovy introduction.

We will continue the Groovy Introduction series with the following:

- Some useful AST transformations
- Understanding the Groovy Meta-Object Protocol (MOP)
- Builders and Slurpers
- Adding Groovy to your Maven Projects
- Using Groovy with Your Favorite IDEs

# Resources & Links

## Groovy

- http://groovy.codehaus.org
- http://groovy.codehaus.org/Operator+Overloading
- http://beta.groovy-lang.org/docs/groovy-2.3.0/html/documentation/core-traits.html
- http://groovy.codehaus.org/Operators