

Advanced Groovy

Daniel Hinojosa

Agenda: Day 1

- Groovy Review
- Setting Up Gradle
- Mopping and MetaProgramming

Lab: Software Installed

Before we begin it is assumed that all of you have the following tools installed:

- JDK 1.8.x

To verify that all your tools work as expected

```
% javac -version
javac 1.8.0_101

% java -version
java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b13, mixed mode)


% groovy -version
Groovy Version: 2.4.8 JVM: 1.8.0_101 Vendor: Oracle Corporation OS: Mac OS X

% gradle --version
3.4.1
```


NOTE | The JDK 8 Version doesn't have to be exact as long as it is Java 8.


Eclipse Plugin

Go to Eclipse Marketplace. Locate Eclipse Buildship Plugin and Install



Buildship Gradle Integration 2.0
Extend your Eclipse IDE to support building software using Gradle. This solution is provided by the Eclipse Foundation. [more info](#)
by [Eclipse Buildship Project](#), EPL
[fileExtension_gradle](#)

 151

 Installs: **171K** (18,736 last month)

Uninstall

Initializing a Project

- Make a directory `advanced_groovy`
- Run the following command

```
gradle init --type=groovy-library
```

Adding some plugins to the `build.gradle`

- In the *build.gradle* file, open with any editor and add the following plugins

```
apply plugin: 'java'  
apply plugin: 'groovy'
```

Opening in IDEA

- Select *File/Open*
- Locate the directory `advanced_groovy`

Opening the project in Eclipse

- Select *File/Import Project*
- Select *Gradle/Existing Gradle Project* in the selection window
- Continue through the Wizard
- Locate the directory `advanced_groovy`

Review of Groovy Tools

- `groovysh`
- `groovyConsole`
- `groovy`

Basic Types Groovy As Review

- Groovy is Java with the following features:
 - The `return` statement is almost always optional.
 - The semicolon (`;`) is almost always optional, only used to separate statements.
 - Methods and classes are `public` by default.

- We are not forced to handle **Exception** if we do not want to.

Customizable Operators in Groovy

```
class USD {
    def amount

    def USD(BigInteger bi) {
        this.amount = bi
    }

    def plus(USD other) { //special word that becomes an operator
        new USD(this.amount + other.amount)
    }

    @Override
    String toString() {
        "USD(amount=$amount)"
    }
}
```

```
USD lawnMowerPay = new USD(3)
println (lawnMowerPay + new USD(10)) //USD(13)
```

Binary Mathematical Operators in Groovy

Operator	Method
$a + b$	<code>a.plus(b)</code>
$a - b$	<code>a.minus(b)</code>
$a * b$	<code>a.multiply(b)</code>
a / b	<code>a.div(b)</code>
$a ** b$	<code>a.power(b)</code>
$a \% b$	<code>a.mod(b)</code>
$a \text{ or } a$	<code>a.next()</code>
$a-- \text{ or } --a$	<code>a.previous()</code>

Logical Operators in Groovy

Operator	Method
$a b$	<code>a.or(b)</code>

Operator	Method
<code>a&b</code>	<code>a.and(b)</code>
<code>a^b</code>	<code>a.xor(b)</code>

Unary Operators in Groovy

Operator	Method
<code>~a</code>	<code>a.bitwiseNegate()</code>
<code>+a</code>	<code>a.positive()</code>
<code>-a</code>	<code>a.negative()</code>

Array Access Operators in Groovy

Operator	Method
<code>a[b]</code>	<code>a.getAt(b)</code>
<code>a[b] = c</code>	<code>a.putAt(b, c)</code>

Shift Operators in Groovy

Operator	Method
<code>a << b</code>	<code>a.leftShift(b)</code>
<code>a >> b</code>	<code>a.rightShift(b)</code>
<code>a >>> b</code>	<code>a.rightShiftUnsigned(b)</code>

Relational Operators in Groovy

Operator	Method
<code>a == b</code>	<code>a.equals(b)</code>
<code>a != b</code>	<code>!a.equals(b)</code>
<code>a <=> b</code>	<code>a.compareTo(b)</code>
<code>a > b</code>	<code>a.compareTo(b) > 0</code>
<code>a >= b</code>	<code>a.compareTo(b) >= 0</code>
<code>a < b</code>	<code>a.compareTo(b) < 0</code>
<code>a <= b</code>	<code>a.compareTo(b) <= 0</code>

Type Conversion Operator in Groovy

```
@Canonical
class Fahrenheit {
    int temperature

    Fahrenheit(temperature) {
        this.temperature = temperature
    }

    def asType(Class target) {
        if (target in Celcius) { //Clever trick!
            return new Celcius(temperature : (this.temperature - 32) * 0.5556)
        }
    }
}
```

```
@Canonical
class Celcius {
    int temperature
}
```

```
new Fahrenheit(temperature:100) as Celcius
```

Classes in Groovy

@groovy.transform.Canonical

- No arg constructor provided
- Set properties by name using Groovy's normal bean conventions.
- Tuple-style constructors allow you to set properties in the same order as defined.
- Default `equals`, `hashCode` and `toString` methods based on properties

```
import groovy.transform.Canonical
import java.time.LocalDate

@Canonical
class Person {
    String firstName;
    String lastName;
    LocalDate dob;
}
```

Alternative Constructions for `@groovy.transform.Canonical`

If a class is labeled as `Canonical` you have the option to create your class in any one of these forms.

```
def person0 = new Person()
person0.firstName = "John"
person0.lastName = "Kennedy"
person0.dob = LocalDate.of(1917, 5, 29)
```

```
def person1 = new Person("John", "Kennedy",
    LocalDate.of(1917, 5, 29))
```

```
def person2 = new Person(firstName: 'John', lastName: 'Kennedy',
    dob: LocalDate.of(1917, 5, 29))
```

A little bit about Alternative JVM Languages

- We can always see what is available in bytecode by using
- It sheds light even in the most complicated scenarios

```
javap -v <class>
```

`@groovy.transform.Newify` Ruby Style

Newify creates a "Ruby Style" mechanism to create Groovy objects.

```
@Canonical
class Person {
    String firstName;
    String lastName;
}

@Newify //Placed atop of the method
def createFirstThreePresidents() {
    [Person.new("George Washington"),
    Person.new("John", "Adams"),
    Person.new("Thomas", "Jefferson")]
}
```

@groovy.transform.Newify Python or Scala Style

"Python Style" or "Scala Style" of instantiation

```
@Canonical
class Person {
    String firstName;
    String lastName;
}

@Newify([Person]) //Be sure to list the class
def createFirstThreePresidents() {
    [Person("George Washington"), //Notice no new keyword
    Person("John", "Adams"),
    Person("Thomas", "Jefferson")]
}
```

@groovy.transform.Immutable

Annotation that creates an immutable class with the following characteristics:

- Made as **final**
- Will create properties of those that are immutable themselves
- Properties have getters only with private backing fields
- Default **equals**, **toString**, **hashCode**
- See more... <http://docs.groovy-lang.org/2.4.7/html/gapi/groovy/transform/Immutable.html>

```
@groovy.transform.Immutable
class Employee {
    String firstName, lastName
}

def e = new Employee('Serena', 'Williams')
println(e)
e.firstName = 'Venus' // Wrong, cannot set readonly property
```

@groovy.transform.TupleConstructor

- This will include default for the properties
- You can enter none, some, or all fields
- If a collection is the last field, we instantiate the class as a vararg


```
@groovy.transform.TupleConstructor
class Employee {
    String firstName, lastName
    private String ssn
    Collection awards
}
new Employee("Bob", "Marley")
new Employee("Bob", "Marley", "Best Singer in the Office",
    "Makes the Best Cookies")
```

@groovy.transform.TupleConstructor with our own constructors

Will not create anything if we create *our own* constructors

```
@groovy.transform.TupleConstructor
class Employee {
    String firstName, lastName
    private String ssn;
    Collection awards

    public Employee(firstName, lastName) {
        this.firstName = firstName
        this.lastName = lastName
    }
}

new Employee("Bob", "Marley", "Best Singer in the Office",
    "Makes the Best Cookies") //Error
```

@groovy.transform.TupleConstructor with a private field

- We can include fields as constructor parameters
- Merely add `includeFields=true` as an annotation parameter
- `private` fields also are done at the end.

```
@groovy.transform.TupleConstructor(includeFields=true)
class Employee {
    String firstName, lastName
    private String ssn
    Collection awards
}
def employee = new Employee("Bob", "Marley", ["Best Work Ethic"], "000-00-0000")
```

@groovy.transform.TupleConstructor with Superclasses

- Use `includeSuperProperties` or `includeSuperFields` to get super class information
- The call *must* be done in order of:
 - Super class properties first
 - Super class fields
 - Class properties
 - Class fields

```
@groovy.transform.TupleConstructor(includeFields=true)
class Employee {
    String firstName, lastName
    private String ssn
    Collection awards
}

@groovy.transform.TupleConstructor(callSuper=true, includeSuperProperties=true,
includeSuperFields=true)
class Supervisor extends Employee {
    Collection employees
}

def gaga = new Employee("Lady", "Gaga", [], "304-12-1230")
def sammy = new Employee("Sammy", "Hagar", [], "123-33-3200")
def bob = new Supervisor("Bob", "Marley", ['Best QA 2014'], '333-22-1441', [gaga,
sammy])
```

@groovy.transform.Memoized

Memoized in any functional, or hybrid functional language will cache the function to avoid re-evaluating an expensive computation.

```
import java.time.*
import groovy.transform.Memoized

@Memoized
def currentTimeAndCached() {
    LocalDateTime.now()
}

def first = currentTimeAndCached()
Thread.sleep(2000)
def second = currentTimeAndCached()

assert first == second //true
```

@groovy.transform.EqualsAndHashCode

- Will create a `equals` and `hashCode` implementation automatically based on the properties
- For `hashCode` it uses Groovy's `org.codehaus.groovy.util.HashCodeHelper`
- Uses the standard algorithm from *Effective Java*

```
import groovy.transform.*

@EqualsAndHashCode
class Employee {
    String firstName
    String lastName
    private String ssn
    Collection awards
}

def employee = new Employee()
employee.firstName = "David"
employee.lastName = "Bowie"
employee.awards = ["Best Music Video 1985"]

def employee2 = new Employee()
employee2.firstName = "David"
employee2.lastName = "Bowie"
employee2.awards = ["Best Music Video 1985"]

assert employee == employee2
assert employee.hashCode() == employee2.hashCode()
```

@groovy.transform.EqualsAndHashCode including fields

To test the equality and include fields add `includeFields=true` to the `EqualsAndHashCode` annotation

```
import groovy.transform.*

@EqualsAndHashCode(includeFields = true)
class Employee {
    String firstName
    String lastName
    private String ssn
    Collection awards

    Employee(firstName, lastName, ssn, awards) {
        this.firstName = firstName
        this.lastName = lastName
        this.ssn = ssn
        this.awards = awards
    }
}

def employee = new Employee("David", "Bowie", "402-12-1230", ["Best Music Video 1985"])
def employee2 = new Employee("David", "Bowie", "402-12-1230", ["Best Music Video 1985"])

assert employee == employee2
assert employee.hashCode() == employee2.hashCode()
```

@groovy.transform.ToString

Implements a `toString` implementation for the class automatically using the properties

```
import groovy.transform.*

@ToString
class Employee {
    String firstName
    String lastName
    private String ssn
    Collection awards

    Employee(firstName, lastName, ssn, awards) {
        this.firstName = firstName
        this.lastName = lastName
        this.ssn = ssn
        this.awards = awards
    }
}

def employee = new Employee("David", "Bowie", "402-12-1230", ["Best Music Video 1985"])
println(employee)
```

Results In:

```
Employee(David, Bowie, [Best Music Video 1985])
```

@groovy.transform.ToString including the names of the properties

To include the names of the properties, add `includNames=true` to the `@ToString` annotation

```

@ToString(includeNames=true)
class Employee {
    String firstName
    String lastName
    private String ssn
    Collection awards

    Employee(firstName, lastName, ssn, awards) {
        this.firstName = firstName
        this.lastName = lastName
        this.ssn = ssn
        this.awards = awards
    }
}

def employee = new Employee("David", "Bowie", "402-12-1230", ["Best Music Video 1985"])
println(employee)

```

Results In:

```
Employee(firstName:David, lastName:Bowie, awards:[Best Music Video 1985])
```

@groovy.transform.ToString including the names of the fields

```

@ToString(includeNames=true, includeFields=true)
class Employee {
    String firstName
    String lastName
    private String ssn
    Collection awards

    Employee(firstName, lastName, ssn, awards) {
        this.firstName = firstName
        this.lastName = lastName
        this.ssn = ssn
        this.awards = awards
    }
}

def employee = new Employee("David", "Bowie", "402-12-1230", ["Best Music Video 1985"])
println(employee)

```

Results In:

```
Employee(firstName:David, lastName:Bowie, awards:[Best Music Video 1985], ssn:402-12-1230)
```

Named Constructor Parameters

```
import groovy.transform.*

@ToString(includeNames=true)
class Employee {
    String firstName
    String lastName
    private String ssn
    Collection awards
}

def employee = new Employee(firstName:'David', lastName:'Bowie') //Using a Map
```

Results In:

```
Employee(firstName:David, lastName:Bowie, awards:null)
```

Regular Expression Matchers

- `==~` (Note the single equals)
- Performs a RegEx partial match

```
def m = '(404) 301-3003' ==~ /\d+/
println(m.size()) //3
println(m[0])     //404
println(m[1])     //301
println(m[2])     //3003
```

Regular Expression Match

The `==~` performs a RegEx exact match and return a boolean whether the entire regex matches

```
'(404) 301-3003' ==~ /\d{3}\s\d{3}-\d{4}/
```

Results In:

```
true
```

Regular Expressions Groups

Groups use the same functionality since it uses a Matcher

```
def m2 = 'The game was Lakers vs. Celtics' =~ /(\w+)\svs.\s(\w+)/
m2.hasGroup() //true
m2.groupCount() //2
println(m2[0][0]) //Lakers vs. Celtics
println(m2[0][1]) //Lakers
println(m2[0][2]) //Celtics
```

Creating a Range

Ranges are created with two periods between characters or numbers

```
def r1 = 1..4
println(r1.getClass().getName()) //IntRange

def r2 = (1..4)
println(r2.getClass().getName())

def r3 = r1 as Set
println(r4.getClass().getName())

def r4 = [1..4] // A list of one range
println(r4.getClass().getName()) //ArrayList
println(r4.size()) //4

def r5 = "A".."Z"
println(r5.size()) //26

def r6 = 'A'..'Z'
println(r6.size()) //26
```

NOTE

All the ranges are defined by the `next` and `prev` methods

Creating a Set

Sets are first creating a `List` and either calling:

- `toSet`

- `as Set` which uses the `asType()` method in `List`

```
def s1 = [1,2,3,2,4,5] as Set
def s2 = [1,2,3,2,4,5].toSet()
```

Creating a Map

```
def stateCaps = [:]

stateCaps['Alabama'] = 'Montgomery'
stateCaps.put('Arizona', 'Phoenix')
stateCaps << ['California' : 'Sacramento']

def stateCaps2 = ['Georgia':'Atlanta', 'New York':'Albany']
```

String and Groovy Strings

```
def regularString = 'Foo'

int age = 30
String city = 'Boise'
String state = 'Idaho'

def groovyString = "Ted is $age"

def multiLineString = '''In a hole in the ground there lived a hobbit.
Not a nasty, dirty, wet hole, filled with the ends of worms and an
oozy smell, nor yet a dry, bare, sandy hole with nothing
in it to sit down on or to eat: it was a hobbit-hole,
and that means comfort.'''

def multiLineGroovyString = """Ted is $age. He lives in $city ($city is
${city.size()} letters long). $city is in $state in case no one
mentioned it to you."""
```

Groovy Characters and Strings

- Groovy makes no distinction between 'a' and "A"

```
def s = 'Groovy is Great'
println s // Groovy is Great
println s.getClass().getName() // java.lang.String

def c = 'a' as Character
println c.getClass().getName() // java.lang.Character
```

dump() and inspect()

dump()

`dump()` reads what is inside of an **object**:

```
def world = "This is a crazy world"
world.dump()
```

Results In:

```
<java.lang.String@b81f46b0 value=This is a crazy world hash=-1205909840>
```

inspect()

`inspect()` provides information how to recreate the object:

```
def list = [1,2,3,4,5]*.plus(4)
println(list.inspect()) // [5, 6, 7, 8, 9]
```

with() method

The **with** method can make filling properties up fairly easy.

```
class Employee {
    String firstName, lastName
    int age
    Collection skills
}

Employee e = new Employee()

e.with {
    firstName = 'Obi-wan'
    lastName = 'Kenobi'
    age = 35
    skills = ['Lightsabering', 'Scaring Tusken']
}
```

Closures

- Help create lightweight reusable code on the JVM
- Help express a verb

```
def callClosure(c) {
    println "Start of method"
    println c.call()
    println "End of method"
}

callClosure{ -> "Need to do something here"}
```

Results in:

```
Start of method
Need to do something here
End of method
```

More Closures

```
def callClosure(c) {
    println "Start of method"
    c.call("Dan", "Hinojosa")
    c.call("Brian", "Sletten")
    println "End of method"
}

callClosure {String fn, String ln ->
    println "${ln}, ${fn} is about to say something profound"}
```

Results in:

```
Start of method
Hinojosa, Dan is about to say something profound
Sletten, Brian is about to say something profound
End of method
```

Roll your own collection and closure

```
class Bag {
  List items
  def myCollect(c) {
    def result = []
    for (Object i : items) {
      result << c(i)
    }
    result
  }
}

Bag groceries = new Bag(items : ["Eggs", "Naan", "Fritos", "Ramen", "Milk"])
groceries.myCollect{i -> i.length()}
```

Results in:

```
[4, 4, 6, 5, 4]
```

Currying

- Currying allows you to take part of a function and re-purpose for other uses
- It transforms a function that take two arguments into the two functions that take one argument.

```
def multiply3 = {a, b, c -> a * b * c}
def multiplyBy6 = multiply3.curry(3, 2)
def items = 1..10
items.collect{multiplyBy6(it)}
```

Results in:

```
[6, 12, 18, 24, 30, 36, 42, 48, 54, 60]
```

Closures and Java 8 Lambdas

- Groovy has seamless support with Java 8 Lambdas.
- Use a closure in place of a Java Lambda.

```
import java.util.stream.Collectors

[1,2,3,4,5].stream()
    .filter{it % 2 == 0}
    .map{it * 2}
    .collect() //Groovy collect from Object
```

Results in:

```
[4,8]
```

Converting a method into a Closure

```
class Foo {
    private int bar

    def process(int baz) {
        bar + baz
    }
}

def foo = new Foo(bar:30)
def fun1 = foo.&process

fun1(3) //33
```

Decorating Collections

List decorators in Groovy

```
def list = [1,2,3,4,5]
println list //Print the list
println list.getClass().name //java.util.ArrayList
```

Extracting from a List

```
def list = [1,2,3,4,5]
println list[0]      // 1
println list[1..4]   // [2,3,4,5]
println list[-2]     // 4
println list[-1..-3] // [5,4,3]
```

Important List operations

- `collect`
- `find`, `findAll`, `grep`
- `inject`

collect

```
def list = [1, 2, 3, 4, 5]
list.collect {i -> i * 2} //[2, 4, 6, 8, 10]
```

or

```
def list = [1, 2, 3, 4, 5]
list.collect {it * 2} //[2, 4, 6, 8, 10]
```

Spread Operator

The previous slide can also be reworked with a spread operator:

```
def list = [1, 2, 3, 4, 5]
list*.multiply //[2, 4, 6, 8, 10]
```

Another Example:

```
def list = ["Red", "Orange", "Yellow", "Green", "Blue", "Indigo", "Violet"]
list*.size() //[3, 6, 6, 5, 4, 6, 6]
```

Object#isCase

```

class NFLTeam {
    String name
    Conference conference
}

class Conference {
    String name

    boolean isCase(Object val) {
        val.getClass() == NFLTeam.class && val.conference.name == this.name
    }
}

def afc = new Conference(name : "AFC")
def nfc = new Conference(name : "NFC")

```

Using Object#isCase

```

String result = null
switch(patriots) {
    case afc : result = "An AFC Team"; break;
    case nfc : result = "An NFC Team"; break;
    default : result = "Unknown"; break;
}
println result

```

Results In:

```
"An AFC Team"
```

find, findAll, grep

find

`find` finds a single object using a closure.

```
[1,2,3,4,5].find{it > 3}
```

Results In:

```
4
```

findAll

`findAll` captures more than one.

```
[1,2,3,4,5].find{it > 3}
```

Results In:

```
[4,5]
```

find and findAll truthiness

"Truthiness!" is anything that is "like a positive"

- For numbers, it is "truthy" if it is a positive integer or floating point.
- For `boolean`, `true`
- For a `Collection`, a collection with 1 or more elements

```
[0,0.0,false,[],30.0,10,[1,2,3]].find() //30.0  
[0,0.0,false,[],30.0,10,[1,2,3]].find() //[30.0, 10, [1, 2, 3]]
```

grep

`grep` take an `Object` and can be anything that can be a case of something (see `Object#isCase` next) In general you can do class types, regex, closures (if applicable to the elements)

```
[0,0.0,false,[],30.0,10,[1,2,3]].grep(BigDecimal) //Object  
["Do", "Re", "Me", "Fa", "So", "La", "Ti", "Do"].grep(/.o/)
```

inject

Inject is a functional reduction:

```
[1,2,3,4,5].inject {total, next -> total * next}
```

Results In:

```
120
```

Question: What mathematical property is this?

Object#isCase?

```
class Conference {
  String name

  boolean isCase(Object val) {
    val.getClass() == NFLTeam.class && val.conference.name == this.name
  }
}

class NFLTeam {
  String name
  Conference conference
}

def afc = new Conference(name : "AFC")
def nfc = new Conference(name : "NFC")
```

grep with Object#isCase

```
def patriots = new NFLTeam(name : "New England Patriots", conference: afc)
def cowboys = new NFLTeam(name : "Dallas Cowboys", conference: nfc)
def saints = new NFLTeam(name : "New Orleans Saints", conference: nfc)
def broncos = new NFLTeam(name : "Denver Broncos", conference: afc)

[cowboys, saints, broncos, patriots].grep(afc) == [broncos, patriots]
```

Chaining static calls

We can use the word `this` to chain calls in a `static` method!

```
class Foo {
  private static int count
  def static bar(item) {
    println("Running ${++count} time with item: $item")
    this
  }
}

Foo.bar("Eeeny").bar("Meeny").bar("Miney")
```

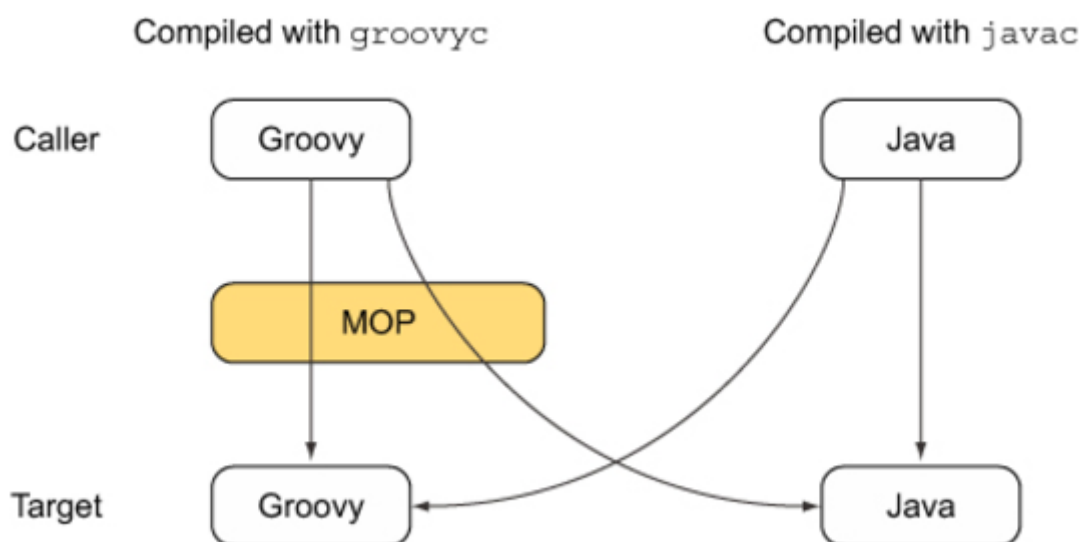
Mopping and Metaprogramming

- Metaprogramming is about programming your code.
- Done to the MetaObject Protocol (MOP)
- We can:
 - Manipulate Objects that already exist
 - Intercept calls and mock
 - Create DSLs

About MetaClass

- For every class in the class loader, Groovy has a metaclass!
- Maintains a collection of all methods and properties
- All instances share the same `metaClass`, but there is support for per-instance `metaClass`
- Calling a method means calling the `metaClass`
- This includes `methodMissing` and `propertyMissing`
- A `metaClass` can change at runtime and an object can change a `metaClass`
- If you do not declare a `metaClass`, one will be looked up in `MetaClassRegistry`
- The `MetaClassRegistry` maintains a map of classes and their `metaClass`

How MOP Works



Source: Groovy In Action[GinA2015]

Asking for the MetaMethod

```
def str = "Hello"
def methodName = "toUpperCase"
def metaClass = str.metaClass
def method = metaClass.getMetaMethod(methodName)
```

Implementing the Method Missing, Property Missing Hooks

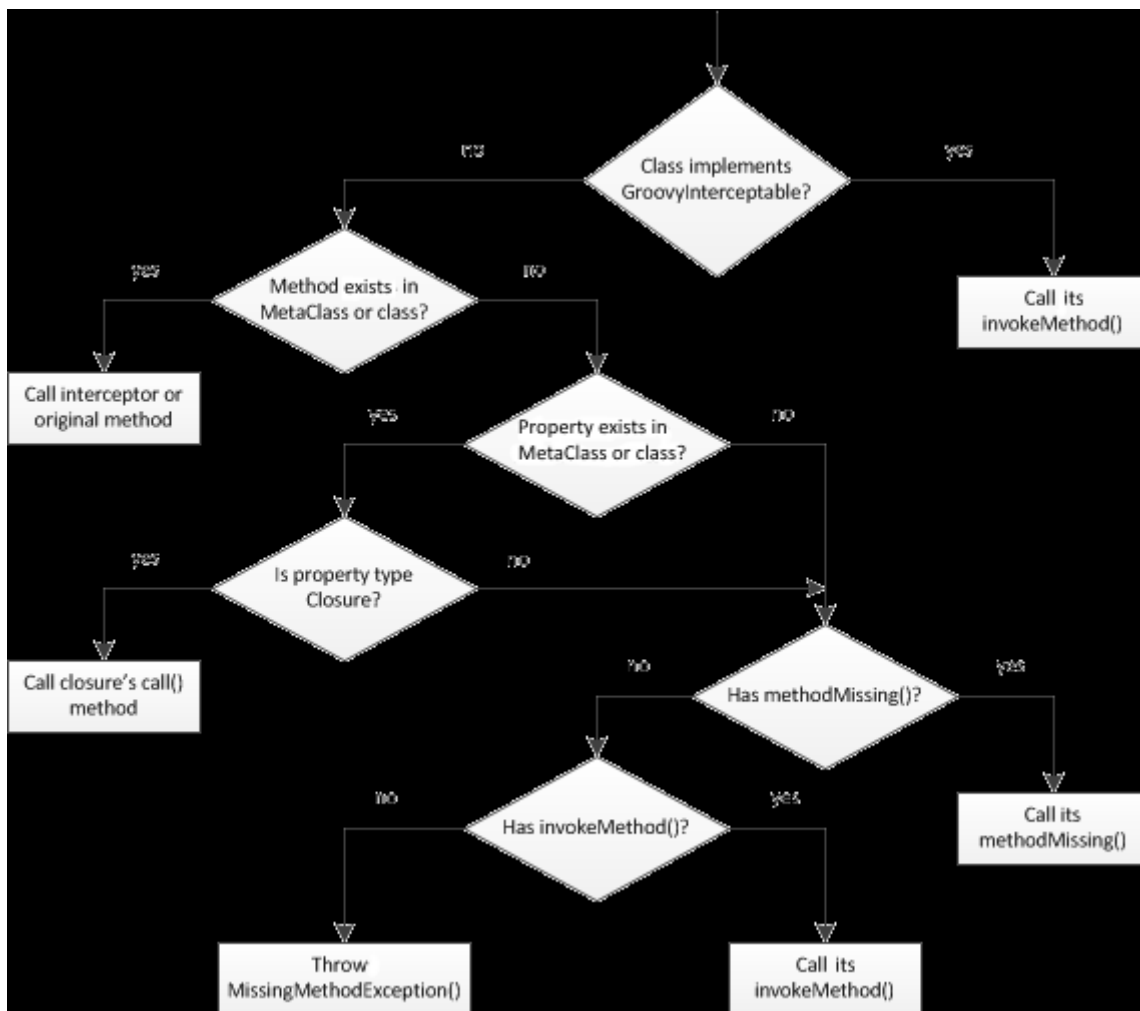
```
class Dynamical {
    private map = [:]

    def propertyMissing(String name, args) {
        map[name] = args
    }

    def propertyMissing(String name) {
        def value = map[name]
        if (value == null) throw new MissingPropertyException("name", this.class)
        value
    }

    def methodMissing(String name, args) {
        def value = map[name]
        System.out.println("value is $value")
        if (value instanceof Closure) {
            return value.call(args)
        } else {
            throw new MissingMethodException(name, this.getClass(), args)
        }
    }
}
```

How Methods are found to be invoked



Interceptions

Interception of methods with `GroovyInterceptable`

If a Groovy Object implements `GroovyInterceptable` then `invokeMethod` is called when *any* of it's methods are called, if not then just the methods that don't exist.

```

package com.xyzcorp;

import java.time.LocalDateTime

class Stock implements GroovyInterceptable {
    String name
    private int sharesLeft;

    Stock(String name, int initialShares) {
        this.name = name
        this.sharesLeft = initialShares
    }

    Stock(String name) {
        this.name = name
        this.sharesLeft = 0
    }

    def buy(int shares) { sharesLeft -= shares }

    def add(int shares) { sharesLeft += shares }

    def invokeMethod(String name, Object args) {
        System.out.println("${LocalDateTime.now()}: method name $name is invoked with
the arguments, $args")
        if (metaClass.respondsTo(this, name, *args)) {
            def metaMethod = metaClass.getMetaMethod(name, *args)
            def result = metaMethod.invoke(this, *args)
            System.out.println("${LocalDateTime.now()}: method name $name has been
invoked with the arguments, $args")
            return result;
        }
        throw new MissingMethodException(name, this.class, *args)
    }
}

```

NOTE

We require `System.out.println` not `println` since `println` is a member of `Object`, and will intercept!

Lab: Create a Threshold

Take the previous example. Ensure that only 100 shares can be bought at a time.

Interceptions of methods using metaClass

- Useful if we are not the owners of the code.

```
class Stock {
    String name
    private int sharesLeft

    public Stock(String name, int initialShares) {
        this.sharesLeft = initialShares
    }

    public Stock(String name) {
        this.sharesLeft = 0
    }

    def buy(int shares) { sharesLeft -= shares }

    def add(int shares) { sharesLeft += shares }
}

Stock.metaClass.invokeMethod = {String name, Object args ->
    System.out.println("${LocalDateTime.now()}: method name $name is invoked with the
arguments, $args")
    if (Stock.metaClass.respondsTo(this, name, *args)) {
        def metaMethod = Stock.metaClass.getMetaMethod(name, *args)
        def result = metaMethod.invoke(delegate, *args)
        System.out.println("${LocalDateTime.now()}: method name $name has been invoked
with the arguments, $args")
        return result;
    }
    throw new MissingMethodException(name, this.class, *args)
}

def stock = new Stock("AAPL")
stock.add(4000)
```

NOTE | `metaMethod.invoke(delegate, *args)`

Intercepting Java Classes

```
Integer.metaClass.invokeMethod = {String name, Object args ->
  System.out.println("Intercepting $name with $args")
  def metaMethod = Integer.metaClass.getMetaMethod(name, args)
  if (metaMethod != null) {
    metaMethod.invoke(delegate, args)
  } else {
    Integer.metaClass.invokeMissingMethod(delegate, name, args)
  }
}
```

NOTE

The type of `metaClass` might be `ExpandoMetaClass` or perhaps `HandleMetaClass` depending on circumstances

Categories

- A category is an object with the ability to alter a class's `metaClass`
- Does so within the block and executing `Thread` (which is highly preferable in case you do not want permanence)

```
class NumberDecorator {
  def static isOdd(self) {
    return self % 2 != 0;
  }

  def static isEven(self) {
    return !isOdd(self)
  }
}

use (NumberDecorator) {
  assert 10.isOdd() == false
  assert 11.isOdd() == true
  assert 15L.isOdd() == true //That's a long!
}
```

NOTE

In order for the category to be in effect it has to be in the scope of the `use` NOTE: For this approach you will need it to be `static`

Categories Using Instances

As an alternate route you can also ask the compiler to do the work of create `static` calls from an instance, so you can avoid the `static` calls in your own code.

```

@Category(Integer)
class NumberDecorator {
    def isOdd(self) {
        return self % 2 != 0;
    }

    def isEven(self) {
        return !isOdd(self)
    }
}

use (NumberDecorator) {
    assert 10.isOdd() == false
    assert 11.isOdd() == true
}

```

NOTE | We can only use on class type in the `@Category` annotation

ExpandoMetaClasses

- Used to add and manipulate
 - Methods
 - Properties
 - Constructors
 - Static methods
- Can be used on Java and Groovy Objects

Using ExpandoMetaClass to create new methods

```

Integer.metaClass.divMod = {divisor -> [ delegate.intdiv(divisor), delegate % divisor
] }

10.divMod(2) //[5, 0]
12.divMod(5) //[2, 2]

```

Using ExpandoMetaClass to add functionality to call one another


```
Integer.metaClass.isOdd = { -> delegate % 2 != 0 }
Integer.metaClass.isEven = { -> !isOdd() }
```

Using `ExpandoMetaClass` to add static functionality

```
Integer.metaClass.'static'.matrix = {x, y, item ->
    def result = []
    (1..x).each {i ->
        def row = []
        (1..y).each {j -> row << item}
        result << row
    };
    result
}

assert Integer.matrix(2, 3, 'x') == [['x', 'x', 'x'],
                                       ['x', 'x', 'x']]
```

Using `ExpandoMetaClass` to add constructors

```
Integer.metaClass.constructor << {Float x -> Math.floor(x)}
Integer.metaClass.constructor << {Double x -> Math.floor(x)}

assert new Integer(90.00) == 90
assert new Integer(4) == 16
```

Unified `Expando Class`

We can bring it all in, and make it concise with under one block

```
Integer.metaClass {
  divMod = { divisor -> [ delegate.intdiv(divisor), delegate % divisor ] }

  isOdd = { -> delegate % 2 != 0 } //No parameters

  isEven = { -> !isOdd() }

  'static' {
    matrix = {x, y, item ->
      def result = []
      (1..x).each {i ->
        def row = []
        (1..y).each {j -> row << item}
        result << row
      };
      result
    }
  }

  constructor = {Float x -> Math.floor(x)}

  constructor = {Double x -> Math.floor(x)}

  constructor = {int x -> x * 4}
}
```

Lab: Take an existing class and add functionality.

The Scala Programming Language has a cool method called `mkString` that goes like this:

```
List(1,2,3,4).mkString(",") // "1,2,3,4"
List("A", "B", "C").mkString("[", ":", "]") // "[A, B, C,]"
```

Create a `mkString` for Groovy for any Collection use any of the techniques we've learned so far

Injecting functionality to a specific object not a class

```

class Employee {
  String firstName
  String lastName
}

def governmentClearance = new ExpandoMetaClass(Employee)

governmentClearance.giveInformation = { ->
  'There were ufos'
}

governmentClearance.initialize()

def stan = new Employee(firstName: 'Stan', lastName: 'Lee')
def fox = new Employee(firstName: 'Fox', lastName: 'Mulder')

fox.metaClass = governmentClearance

assert fox.giveInformation() == 'There were ufos'

```

Inject specific functionality to an object concisely

```

class Employee {
  String firstName
  String lastName
}

def stan = new Employee(firstName: 'Stan', lastName: 'Lee')
def fox = new Employee(firstName: 'Fox', lastName: 'Mulder')

fox.metaClass.giveInformation = { ->
  'There were ufos'
}

assert fox.giveInformation() == 'There were ufos'

```

Two or more add methods to one object

```

class Employee {
  String firstName
  String lastName
}

def fox = new Employee(firstName: 'Fox' , lastName: 'Mulder')
def dana = new Employee(firstName: 'Dana', lastName: 'Scully')

fox.metaClass {
  giveInformation = { -> 'There were ufos' }
  partner = { -> dana }
}

assert fox.giveInformation() == 'There were ufos'
assert fox.partner() == dana

```

Traits

Traits allow a mixing in of functionality and state to a class or object

```

trait Published {
  private def books = []
  def numberBooks() { books.size() }
  def addTitle(title) { books << title }
}

class Author implements Published { // implementing trait
  String firstName
  String lastName
}

def king = new Author(firstName: 'Stephen', lastName: 'King')

king.addTitle('It')
king.addTitle('Pet Sematary')

assert king.numberBooks() == 2

```

Mixing in a Trait at Runtime

A trait can be applied at runtime with `as`

```

trait Published {
  private def books = []
  def numberBooks() { books.size() }
  def addTitle(title) { books << title }
}

class Person {
  String firstName
  String lastName
}

def toni = new Person(firstName: "Toni", lastName: "Morrison") as Published
toni.addTitle('The Bluest Eye')
toni.addTitle('BeLoved')
assert toni.numberBooks() == 2

```

Multiple Traits at Compile Time

You can implement multiple **traits** with **implements** in the class definition

```

trait Published {
  private def books = []
  def numberBooks() { books.size() }
  def addTitle(title) { books << title }
}

trait Awarded {
  private def awards = []
  def numberAwards() { awards.size }
  def addAward(award) { awards << award }
}

class Author implements Published, Awarded {
  String firstName
  String lastName
}

def king = new Author(firstName: 'Stephen', lastName: 'King')

king.addTitle('It')
king.addTitle('Pet Sematary')
king.addAward(['O. Henry', 1996])
king.addAward(['Bram Stoker Award', 2011])
assert king.numberBooks() == 2
assert king.numberAwards() == 2

```

Multiple Traits at Runtime

To apply multiple traits to an object we must use the method `withTraits` on an object.

```
trait Published {
  private def books = []
  def numberBooks() { books.size() }
  def addTitle(title) { books << title }
}

trait Awarded {
  private def awards = []
  def numberAwards() { awards.size }
  def addAward(award) { awards << award }
}

class Person {
  String firstName
  String lastName
}

def toni = (new Person(firstName: "Toni", lastName: "Morrison")).withTraits Published,
Awarded
toni.addTitle('The Bluest Eye')
toni.addTitle('Beloved')
toni.addAward(['Pulitzer', 1988])
toni.addAward(['Noble Prize', 1993])
assert toni.numberBooks() == 2
assert toni.numberAwards() == 2
```

trait Conflicts

When a trait conflict happens, the last trait declared is the winner. Oftentimes the `trait` requires an anchor from which the rest of the traits can attach.

```

trait Stamped { //anchor
  def logn(log) { log } //Simple
}

trait TimeStamped {
  def logn(log) { "2015-01-01 13:00:00: " + log }
}

trait ThreadStamped{
  def logn(log) { "[Thread-1] " + log }
}

trait UserStamped {
  def logn(log) { "(User: Randall) " + log }
}

class Logger implements TimeStamped, ThreadStamped, UserStamped {
  def info(log) { logn("INFO: " + log) }
  def warn(log) { logn("INFO: " + log) }
  def fatal(log) { logn("FATAL: " + log) }
}

def logger = new Logger()
logger.info("Hello this is a test")

```

Chaining trait

- Traits can be changed much like the decorator pattern, all is required is a call to `super` with the method to go to the next trait

We want to create a Log Decorator. Create one a Log for the time

```

trait TimeStamped {
  def logn(log) { super.logn("2015-01-01 13:00:00: " + log)}
}

```

Lab: Add some other filters to the chained trait

- Apply `super.logn` to all the filters except the base trait `Stamped`
- Ensure the stack works
- Let's get real
 - Replace `TreadStamped` with the real thread name using `Thread.currentThread().getName()`

- Replace `UserStamped` with the real user name using `System.getProperty('user.name')`
- Replace `TimeStamped` with the real date time using `LocalDateTime.now`

Expando

Expando allows us to create classes dynamically

```
def author = new Expando(firstName:'Rudolph', lastName: 'Anaya')

author.firstName = 'Rudolfo'
author.awards = [['National Medal of Arts', 2001], ['American Book Awards', 1980]]

assert author.awards.size() == 2
```

Expando with Closure

You can reference any property or method within in an `Expando` to create things on the fly.

```
def author = new Expando(firstName:'Rudolph', lastName: 'Anaya')

author.firstName = 'Rudolfo'
author.awards = [['National Medal of Arts', 2001], ['American Book Awards', 1980]]

assert author.awards.size() == 2

author.awardTitles = {-> awards.collect{it[0]}}

author.awardTitles()
```

Agenda: Day 2

- XML/JSON
- Database Development
- Testing

Working with XML

XML Markup Builder

To create XML one of the great things that we can do is use `MarkupBuilder` to create our XML.


```
def builder = new groovy.xml.MarkupBuilder();
builder.html {
    head {
        title 'Sample Strict Style Webpage!'
    }
    body {
        span(id:'welcome') { h2 'Welcome to our page' }
    }
}
```

XML Markup Builder with `mkp`

There is a special `mkp` that can be used to create specialized XML constructs.

Per definition in GroovyDoc:

`mkp` is a special namespace used to escape away from the normal building mode of the builder and get access to helper markup methods `yield`, `pi`, `comment`, `xmlDeclaration` and `yieldUnescaped`.

`mkp` Methods

<code>comment(String value)</code>	Produce a comment in the output.
<code>pi(Map<String,Map<String,Object>> args)</code>	Produce an XML processing instruction in the output.
<code>xmlDeclaration(Map<String,Object> args)</code>	Produce an XML declaration in the output.
<code>yield(Object value)</code>	Prints data in the body of the current tag, escaping XML entities.
<code>yield(String value)</code>	Prints data in the body of the current tag, escaping XML entities.
<code>yieldUnescaped(Object value)</code>	Print data in the body of the current tag.
<code>yieldUnescaped(String value)</code>	Print data in the body of the current tag.

XML Markup Builder with `mkp`

To create XML one of the great things that we can do is use `MarkupBuilder` to create our XML.

```

import java.time.LocalDateTime
def builder = new groovy.xml.MarkupBuilder();
builder.html {
    head {
        mkp.comment("The title should go here")
        title 'Sample Strict Style Webpage!'
    }
    body {
        span(id:'welcome') {
            h2 'Welcome to our page'
            span (id:'date_time_msg') {
                mkp.yield('The date and time is ')
                mkp.yield(LocalDateTime.now())
            }
        }
    }
}

```

XML Markup Builder with a Writer

```

def writer = new StringWriter()
def builder = new groovy.xml.MarkupBuilder(writer);
builder.html {
    head {
        title 'Sample Strict Style Webpage!'
    }
    body {
        span(id:'welcome') { h2 'Welcome to our page' }
    }
}
writer.flush()
writer.toString() //Renders XML

```

XML Markup Builder

While the previous examples, deal with small data set for rendering XML, if we have larger streams of data, then use [StreamingMarkupBuilder](#)

```
import java.time.LocalDateTime
def xml = new groovy.xml.StreamingMarkupBuilder().bind {
    html {
        head {
            mkp.comment("The title should go here")
            title 'Sample Strict Style Webpage!'
        }
        body {
            span(id:'welcome') {
                h2 'Welcome to our page'
                span (id:'date_time_msg') {
                    mkp.yield('The date and time is ')
                    mkp.yield(LocalDateTime.now())
                }
            }
        }
    }
}
```

Create Stock Information From Streaming Data in XML

Pair Program the Following

Using the following URL:
http://ichart.finance.yahoo.com/table.csv?s=<STOCK_SYMBOL>&a=00&b=01&c=2017

- Read each line from the website and create an XML based on your interpretation of the data based on the incoming stream
- Use `StreamMarkupBuilder`
- Replace `<STOCK_SYMBOL>` in the above URL with your favorite stock symbol

Hints: * Use `toURL` in a `String` to convert from `String` to `URL` * `URL` has an `each` method that takes a `Closure`

GPath vs. XPath

- GPath is a groovy path type of schematic to look up values
- Works nearly the same as navigating POJOs

<code>a.b.c</code>	yields all the <code><c></code> elements inside <code></code> inside <code><a></code>
<code>a["@href"]</code>	the href attribute of all the a elements
<code>a.'@href'</code>	an alternative way of expressing this

`a.@href`

an alternative way of expressing this when using `XmlSlurper`

Parsing XML with `XmlParser`

- Use `XmlParser` if you want to update and parse at the same time
- `XmlParser` will return a `Node` object

```
String xml = '''<person id=\'49912\'>
    <firstName>Frank</firstName>
    <lastName>Zappa</lastName>
</person>'''

def person = new XmlParser().parseText(xml)
person.firstName.text() == 'Frank' //GPath
person['@id'].toInteger() == 49912 //GPath
```

Slurping XML with `XmlSlurper`

- `XmlSlurper` evaluates the structure lazily. So if you update the xml you'll have to evaluate the whole tree again.
- `XmlSlurper` returns `GPathResult` instances when parsing XML

Updating XML at the same time as reading it

children traversal

Given:

```

<a>
  <b>
    <d>
      <h>Ache</h>
    </d>
    <e>
      <i>Eye</i>
      <j>Jay</j>
    </e>
  </b>
  <c>
    <f>
      <k>Kay</k>
    </f>
    <g>Gee</g>
  </c>
</a>

```

```

a. '<strong>'</strong>.name() // ['b', 'c']
a.children()*.name() // ['b', 'c']

```

breadthFirst

Given:

```

<a>
  <b>
    <d>
      <h>Ache</h>
    </d>
    <e>
      <i>Eye</i>
      <j>Jay</j>
    </e>
  </b>
  <c>
    <f>
      <k>Kay</k>
    </f>
    <g>Gee</g>
  </c>
</a>

```

```

a.breadthFirst()*.name() // ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']

```

depthFirst

Given:

```
<a>
  <b>
    <d>
      <h>Ache</h>
    </d>
    <e>
      <i>Eye</i>
      <j>Jay</j>
    </e>
  </b>
  <c>
    <f>
      <k>Kay</k>
    </f>
    <g>Gee</g>
  </c>
</a>
```

```
a.'<strong>*</strong>.name() // ['a', 'b', 'd', 'h', 'e', 'i', 'j', 'c', 'f', 'k', 'g']
a.depthFirst().name() // ['a', 'b', 'd', 'h', 'e', 'i', 'j', 'c', 'f', 'k', 'g']
```

DOMCategory

DOMCategory is used for bringing a DOM from any source like java, and decorating with GPath

```
import groovy.xml.dom.DOMCategory
import org.w3c.dom.Document
import javax.xml.parsers.DocumentBuilder
import javax.xml.parsers.DocumentBuilderFactory

String xml = ''' <a>...</a> '''

def inputStream = new ByteArrayInputStream(xml.bytes)
DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
Document doc = dBuilder.parse(inputStream);

use(DOMCategory) {
    println(doc.getDocumentElement().'<strong>*</strong>.name())
}
```

Results In:

```
[a, b, d, h, e, i, j, c, f, k, g]
```

Lab: Find all the counties in your state

1. Use http://api.sba.gov/geodata/city_county_links_for_state_of/<STATE_NAME>.xml
2. Parse the Data and Group Major Cities by County

JSON and Groovy

Parsing with JSONSlurper

There is a `JSONParser` but is an `interface` and no a concrete option

```
import groovy.json.JsonSlurper
def jsonSlurper = new JsonSlurper()
def map = jsonSlurper.parseText('{ "firstName": "John", "lastName" : "Goodman"}')
map instanceof Map //true
(map as Map).size() // 2
map['firstName'] == 'John'
```

JSON with Lists

JSON respects Lists

```
import groovy.json.JsonSlurper
def jsonSlurper = new JsonSlurper()
def map = jsonSlurper.parseText('{ "firstName": "John", "lastName" : "Goodman",
  "awards" : ["Golden Globe Best Actor 1993"]}')
map instanceof Map //true
map['awards'] instanceof List //true
map['awards'] == ["Golden Globe Best Actor 1993"] //true
```

JSONOutput to create JSON from Map

```
import groovy.json.JsonOutput
def statesCaps = ['Phoenix':'Arizona', 'Sacramento':'California', 'Helena':'Montana']
JsonOutput.toJson(statesCaps)
```

Results In:

```
{"Phoenix":"Arizona","Sacramento":"California","Helena":"Montana"}
```

JSONOutput from a Groovy Object

```
import groovy.json.JsonOutput
def helenMirren = new Employee(firstName:'Helen', lastName:'Mirren', awards: ['Academy Award Best Actress 2007'])
JsonOutput.toJson(helenMirren)
```

```
{"awards":["Academy Award Best Actress 2007"],"firstName":"Helen","lastName":"Mirren"}
```

Database Development

Initializing a SQL Connection

Assuming we are using HSQLDB. Use Appropriate URL, username, password, and driver Ensure that the driver is in the classpath

```
def sql = groovy.sql.Sql.newInstance('jdbc:hsqldb:hsqldb://localhost/mydb', 'SA', '', 'org.hsqldb.jdbc.JDBCdriver')
```

Inserting Data

```
def sql = groovy.sql.Sql.newInstance('jdbc:hsqldb:hsqldb://localhost/mydb', 'SA', '', 'org.hsqldb.jdbc.JDBCdriver')
def city = 'Albuquerque'
def county = 'Bernalillo'
sql.executeInsert("INSERT INTO counties (name, city) values (${city}, ${county})")
```

Selecting Data

There are various ways to query databases. `eachRow` uses a closure to analyze each row `rows` returns full rows


```
def sql = groovy.sql.Sql.newInstance('jdbc:hsqldb:hsqldb://localhost/mydb', 'SA', '',
'org.hsqldb.jdbc.JDBCdriver')
sql.eachRow("SELECT id, name, city FROM COUNTIES WHERE city = ${city}") { id,
col_name, col_city -> "Found record ${id}"}
sql.rows('SELECT * FROM COUNTIES').size() > 0
```

Updating Data

```
def city = 'Los Lunas'
def county = 'Bernalillo'
sql.executeUpdate("UPDATE counties set CITY = ${city} WHERE name = ${county}")
sql.eachRow("SELECT city from counties where name = $county"){c -> assert c == city}
```

Object-rational database development

In the Grails Space there is GORM, which is fun, here are some quick samples from <https://docs.grails.org/latest/guide/GORM.html>

Create

```
def p = new Person(name: "Fred", age: 40, lastVisit: new Date())
p.save()
```

Read

```
def p = Person.get(1)
assert 1 == p.id
```

Update

```
def p = Person.get(1)
p.name = "Bob"
p.save()
```

GORM Querying Using MetaProgramming

```
def books = Book.findAllByTitleLikeOrReleaseDateGreaterThan(
"%Java%", new Date() - 30)
```

```
def books = Book.findAllByTitleLike("Harry Pot%",
    [max: 3, offset: 2, sort: "title", order: "desc"])
```

Lab: Connect to Database

- Include in **build.gradle** dependencies: `compile 'org.hsqldb:hsqldb:2.3.4'`
- Also download the jar file from <http://search.maven.org/remotecontent?filepath=org/hsqldb/hsqldb/2.3.4/hsqldb-2.3.4.jar>
- Run a server at the command prompt: `java -cp lib/hsqldb.jar org.hsqldb.Server --database.0 file:mydb --dbname.0 mydb`
- Run the console: `java -cp lib/hsqldb.jar org.hsqldb.util.DatabaseManagerSwing`
- Create table 'counties' with 'CREATE TABLE counties (id integer identity primary key, name varchar(250), city varchar(250))'
- Store the information from the counties that you retrieved from the webservice earlier in whatever format you'd like.
- Check the database with the console

Testing

Review of JUnit and testing concepts

- Red/Bar Green Bar Test
- Recently JUnit 4, closer to JUnit 5
- Extension to JUnit in Groovy by extending `GroovyTestCase`

Integrating Groovy with JUnit

- Here is a simple test that just runs `GroovyTestCase`
- This

```
class MyTest extends GroovyTestCase {
    void testAddition() {
        def result = 1 + 2
        assertEquals(3, result)
    }
}
```

Determining code coverage with Cobertura

Working with Mock Objects

Any Mocking Framework that works with Java can work with Groovy including but not limited to: * JMock * EasyMock * Mockito

Mocking with Maps

```
class Die {
    private Random random;
    int pips;

    Die(Random random) {
        this.random = random
    }

    void roll() {
        this.pips = random.nextInt(6) + 1
    }
}

Random random = [nextInt : {int bound -> 4}] as java.util.Random

Die die = new Die(random)
die.roll()
assert die.pips == 5
```

Mocking with Expando

This works well if the item that is to be mocked is not bound by type.

```

class Die {
    private def random;
    int pips;

    Die(random) {
        this.random = random
    }

    void roll() {
        this.pips = random.nextInt(6) + 1
    }
}

def random = new Expando(nextInt: {int bound -> 4})

Die die = new Die(random)
die.roll()
assert die.pips == 5

```

Mocking with Groovy Mock **mockFor**

Groovy has **mockFor** that can create a mock.

```

import groovy.mock.interceptor.*;
import java.util.Random

class Die {
    private Random random
    int pips

    Die(Random random) {
        this.random = random
    }

    def roll() {
        this.pips = random.nextInt(6) + 1
    }
}

def randomMock = new MockFor(Random)
randomMock.demand.nextInt(1..1) {int x -> 4} //run once

randomMock.use {
    def die = new Die(new Random())
    die.roll()
    assert die.pips == 5
}

```

Mocking with Groovy Stub `stubFor`

```
import groovy.mock.interceptor.*;
import java.util.Random

class Die {
    private Random random
    int pips

    Die(Random random) {
        this.random = random
    }

    def roll() {
        this.pips = random.nextInt(6) + 1
    }
}

def randomStub = new StubFor(Random)
randomStub.demand.nextInt(1..1) {int x -> 4}

randomStub.use {
    def die = new Die(new Random())
    die.roll()
    assert die.pips == 5
}
```

Mocking with Categories

```

public class InterestService {
    private float balance;

    protected final Float lookupLatestInterestRate() throws IOException {
        //Calling a web service
        Thread.sleep(5000);
        return 0.04;
    }

    public synchronized void addToAccount(int amount) throws IOException {
        this.balance += (this.lookupLatestInterestRate() * amount) + amount;
    }

    public synchronized float getBalance() {
        return balance;
    }
}

class MaskCategory {
    static Float lookupLatestInterestRate(InterestService is) throws IOException {
        return 0.01
    }
}

use(MaskCategory) {
    def intServ = new InterestService()
    intServ.addToAccount(10)
    assert (intServ.balance - 10.1) < 0.01
}

```

Testing with Spock

- Specification Based Testing Framework
- Each Specification describes features
- Can either be a class or an entire system, known as *System Under Specification* (SUS)

Address of Spock

```
'org.spockframework:spock-core:1.1-groovy-2.4-rc-3'
```

Imports

```
import spock.lang.*
```

Specification

Under the hood there is an engine called *Sputnik* that runs as a JUnit Test

```
class MyFirstSpecification extends Specification {  
    // fields  
    // fixture methods  
    // feature methods  
    // helper methods  
}
```

Instance Fields

- Fixture Elements used throughout the test
- These objects *are not* shared between methods

Sharing Fields

If you wish to share the field, then use the `@Shared` annotation

```
@Shared WebDriver webdriver = new FirefoxWebDriver()
```

NOTE | Per documentation it is best to initialize at the point of declaration

WARNING | `static` fields should only be used for constants

Fixture Methods

Fixture Methods are lifecycles for that trigger before or after each test method is run.

```
def setup() {}           // run before every feature method  
def cleanup() {}         // run after every feature method  
def setupSpec() {}       // run before the first feature method  
def cleanupSpec() {}     // run after the last feature method
```

WARNING | `setupSpec` and `cleanupSpec` may not share instance fields unless annotated with `@Shared`

Fixture Methods with Inheritance

- `setup()` in the superclass will run before the `setup` methods of the subclass

- `cleanup()` in the *subclass* will run before the `cleanup()` methods of the *superclass*
- `setupSpec()` in the *superclass* will run before the `setupSpec()` methods of the *subclass*
- `cleanupSpec()` in the *subclass* will run before the `cleanupSpec()` methods of the *superclass*

WARNING | Never call `super.setup()` or `setup.cleanup`, Spock will take care of that for you.

Creating a Feature

- Inside the *Specification* comes the feature methods
- Each features is usually a *String*

```
def "full names is the first name and last name combined" {  
    //blocks here  
}
```

The phases of a feature

- Set up the feature's fixture
- Provide a stimulus to the system under specification
- Describe the response expected from the system
- Clean up the feature's fixture

NOTE | The first and last phases are optional

The blocks of phases

- `setup`
- `when`
- `then`
- `expect`
- `cleanup`
- `where`

About blocks

- Any statements between the beginning of the method and the first block are an implicit `setup`
- A feature must have one explicit block
- Blocks cannot be nested

setup

- Place work that you are setting up here
- May not be preceded
- May not be repeated
- May be omitted whereas the it will take on the implicit **setup** (see previous slide)
- **given** is analogous to **setup**

```
setup:  
def c = new Calculator()  
def num1 = 30  
def num2 = 60
```

when and then

- **when** and **then** always occur together.
- **when** is usually an stimulus
- **then** are the conditions, exceptions, and expected interactions
- A feature may contain one or more pairs of **when-then**

Conditions

- Inside of **then** are the conditions
- Represented as **boolean** or Groovy-truths

Example Specification

```
class CalculatorSpecification extends Specification {  
    Calculator calculator;  
  
    def "A calculator should add two items"() {  
        given:  
            calculator = new Calculator();  
        when:  
            def result = calculator.add(40, 10)  
        then:  
            result == 60 //Oh oh  
    }  
}
```

Verification Example

Any failure will provide a full break down

Condition not satisfied:

```
result == 60
|         |
50      false
```

Expected :60

Actual :50

Exception Handling

If there are exception, you can assert one was thrown using `thrown()`

```
def "A calculator shouldn't be allow to divide two numbers"() {
  given:
    calculator = new Calculator();
  when:
    calculator.divide(40, 0)
  then:
    thrown(IllegalArgumentException)
}
```

Introspection Exception Handling

Exceptions are better handled when you get to see the cause and message. This is usually because sometimes Exceptions can be caused for varying reasons.

```
def "A calculator shouldn't be allow to divide two numbers"() {
  given:
    calculator = new Calculator();
  when:
    calculator.divide(40, 0)
  then:
    def e = thrown(IllegalArgumentException)
    e.message == "Cannot divide by zero"
}
```

Asserting something is not thrown

```
def "A calculator should be able to divide by a number other than 0 just fine"() {  
  given:  
    calculator = new Calculator();  
  when:  
    def result = calculator.divide(40, 2)  
  then:  
    notThrown(IllegalArgumentException)  
    result == 20  
}
```

Data Driven Testing

If you need to run the test with varying inputs and expected results, you can use data style tables for your tests

In it's naive form we can use the **expect** block to list out all the expectations from a function call

```
class CalcStatsSpecification extends Specification{  
  Calculator calculator = new Calculator();  
  
  def "the addition of two numbers"() {  
    expect :  
    calculator.add(1, 3) == 4  
    calculator.add(5, 5) == 10  
    calculator.add(0, 0) == 0  
    calculator.add(12, 55) == 67  
  }  
}
```

WARNING

The problem is that this is too brittle, repeated, and tough to find which line will fail.

Using the **where** block and data driven tables

We can now break this down in a more visible approach.

```
def "the addition of two numbers using data values"(int a, int b, int total) {
  expect :
  calculator.add(a, b) == total

  where:
  a | b | total
  1 | 3 | 4
  5 | 5 | 10
  0 | 0 | 0
  12 | 55 | 67
}
```

Cleanup

Blocks can be cleaned up using `cleanup` and can only be followed by a `where:` block

```
setup:
def webDriver = FirefoxWebDriver()

cleanup:
if (webDriver != null) webDriver.close()
```

Dependencies Required with Mocking Concrete Classes

```
'cglib:cglib-nodep:3.2.5'
```

Mocking

Mocking with Spock

To create mock objects with Spock it is a matter of the following signatures

```
def random = Mock(Random)
```

```
Random random = Mock()
```

About Spock Mocks

- Spock Mocking is lenient
- All mocks return a default value: `0`, `false`, `[]`, unless called upon. Falsiness
- Mocks will return an appropriate `equals`, `toString`, and `hashCode` to discriminate between mocks

Mocking By Example

```
class AccountServiceSpecification extends Specification {  
  
    def logService = Mock(LogService)  
    def accountService = new AccountService(logService)  
  
    def "a message should be sent the remote system that we logged in"() {  
        when:  
        def success = accountService.login("sam@foo.com", "myPassw0rd".bytes)  
        then:  
        success  
        1 * logService.send("Account sam@foo.com logged in") //testing behavior  
    }  
}
```

NOTE

If none of the interactions ever gets called an `Interaction Not Satisfied Error` will occur

Cardinality

<code>1 * logService.send(..)</code>	exactly one call
<code>0 * logService.send(..)</code>	zero calls
<code>(1..3) * logService.send(..)</code>	between one and three calls (inclusive)
<code>(1.._) * logService.send(..)</code>	at least one call
<code>(_..3) * logService.send(..)</code>	at most three calls
<code>_ * logService.send(..)</code>	any number of calls, including zero

Target Constraint

To be general with any mock you can replace the target with an underscore `_`.

```
1 * _.send("Account sam@foo.com logged in")
```

This will check that any mock will receive the method send.

Method Constraint

Methods can also become wildcarded like targets

```
1 * logService./s..d/("Account sam@foo.com logged in")
```

Argument Constraints

Arguments can also take on wildcard patterns

```
1 * logService.send("Account sam@foo.com logged in")
1 * logService.send(!("Account jill@foo.com logged in"))
1 * logService.send() //No arguments
1 * logService.send(_) //Any single argument
1 * logService.send(*_) //Zero or more argument list
1 * logService.send(!null) //Not Null
1 * logService.send(_ as String) //Anything that is a String
1 * logService.send({it.contains("sam@foo.com")}) // A Closure Predicate
```

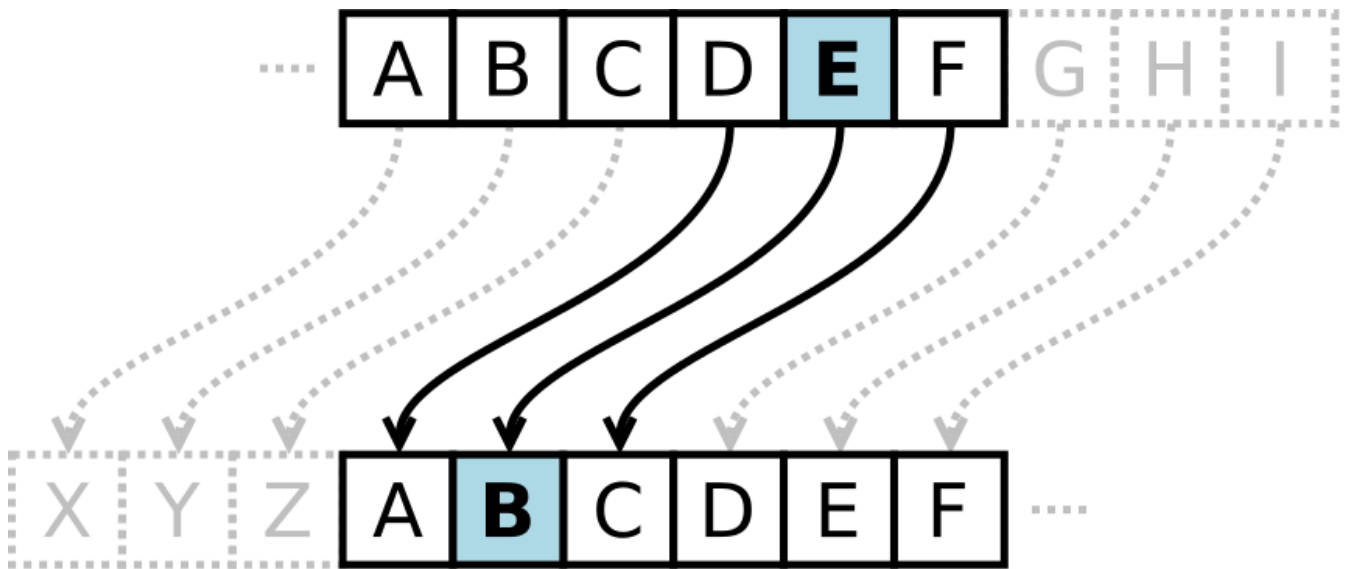
NOTE

You can also do: `1 * logService._(*_)` or `1 * _._` for all wildcards, but may not be useful

Lab: If time remaining, or do it for homework!

Caesar Cipher!

The Caesar cipher, also known as a shift cipher, is one of the simplest forms of encryption. It is a substitution cipher where each letter in the original message (called the plaintext) is replaced with a letter corresponding to a certain number of letters up or down in the alphabet.



- Create a Caesar Cipher with Spock and Test Driven Development
- Ensure Code Coverage Using Cobertura

Source: <https://learncryptography.com/classical-encryption/caesar-cipher>