Meta-programming

Software Design and Programming

KLM

Department of Computer Science and Information Systems
Birkbeck, University of London

keith@dcs.bbk.ac.uk

Spring term 2016

A Meta Object Protocol (MOP) is a mechanism that makes the semantics of a program extensible

Why do it?

- It is state of the art
- It allows you to write more expressive code
- It allows you to write more succinct code

But...

- You may need to change the way you think about things
- The laws are different in the "duck typing" universe

Types

POJO Plain Old Java Object
You interact with an object based upon what it is (class)

POGO Plain Old Groovy Object
You interact with an object based on which properties and methods are available on the object (meta-class)

Typing

Static the type checking is performed during compile-time
e.g., C++, C#, Eiffel, F#, Go, Fortran, Java, Objective-C, etc.

Dynamic the majority of its type checking is performed at run-time
e.g., Erlang, Groovy, Lisp, Python, Ruby, Smalltalk

"Duck" the type checking occurs only on demand at runtime
"if it walks like a duck, and quacks like a duck, then it is a duck"

Do not confuse this with **strong** and **weak** typing
One can still have a strong and dynamically typed language

Modifying class behaviour at runtime I

Also known as

- "Monkey Patch",
- "Guerrilla patch", and
- "Duck punching"

(see definitions on the website), e.g., "cat into a running kernel"

Modifying class behaviour at runtime II

Is used to:

- Replace methods, fields, functions at runtime
  e.g., to "stub" out a method during testing ("mocking")
- Modify or extend the behaviour of a third party entity without requiring access to the source code

Modifying class behaviour at runtime III

- Apply a patch at runtime to the objects in memory instead of the source code on disk (or maybe you dont have the source code)
- Distribute security fixes
- Behavioural changes (e.g., add logging) that live alongside the original source code

How does groovy implement a MOP?

The `MetaClass`

- Every Groovy object has one
- Java has reflection – this goes one stage further, providing a richer introspection and invocation interface
- You can set the `MetaClass` for an object to change the rules for how it can behave

Meta-Programming Hooks

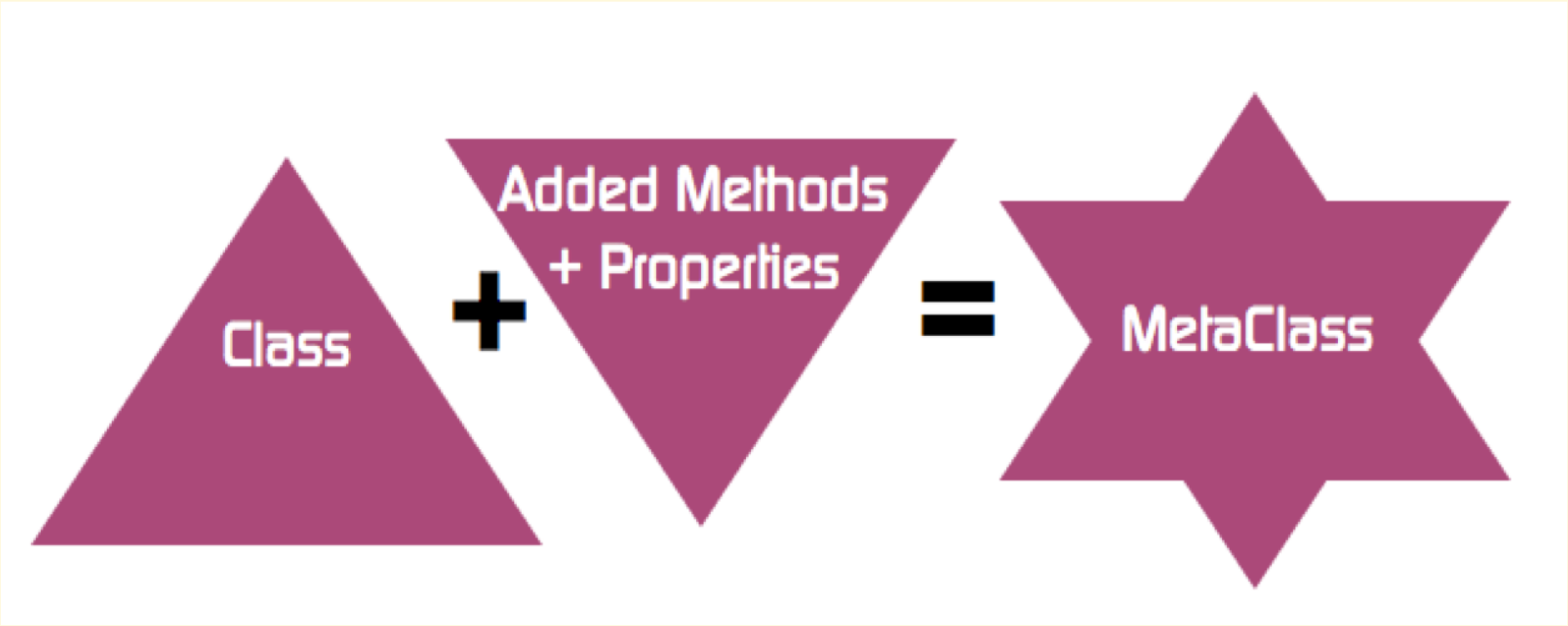
invokeMethod
: intercepts method dispatch to a given class

methodMissing
: intercepts only failed method dispatch for a given class

getProperty/setProperty
: intercepts all property access to a given class

propertyMissing
: intercepts only failed property access for a given class

The final shape of an object



The `ExpandoMetaClass`

- The core of Groovy meta-programming
- Allows one to customise behaviour on the fly
- When you ask a newly created Groovy object for its `metaClass`, this is what you get back

Testing the ExpandoMetaClass

```
class ExpandoMetaClassTest extends GroovyTestCase {
    void testExpandoMetaClass(){
        String message = "Hello"
        shouldFail(MissingMethodException){
            message.shout()
        }

        String.metaClass.shout = { ->
            delegate.toUpperCase()
        }

        assertEquals "HELLO", message.shout()

        String.metaClass = null

        shouldFail{
            message.shout()
        }
    }
}
```

What can you do with it?

- Borrow methods from other classes
- Add or override constructors
- Dynamically create methods
- Add methods to interfaces
- Add or override instance methods
- Add or override properties
- Add or override static methods

`http://docs.groovy-lang.org/latest/html/documentation/#metaprogramming_emc`

Adding a class method

```
String.metaClass.americanise = {
    matcher = (delegate =~ /ise/)
    matcher.replaceAll('ize')
}

println 'finalise'.americanise()
```

Adding a property

```
Integer.metaClass.getSausages << { ->
    def stringOfSausages = []
    delegate.times {
        stringOfSausages << 'sausage'
    }
    return stringOfSausages
}

println 3.sausages
```

Static methods

```
3.times {
    println Math.random()
}

Math.metaClass.static.random = { ->
    return 0.5
}

3.times {
    println Math.random()
}
```

Many methods at once

```
class FilesTest extends GroovyTestCase {
    void testFiles() {
        File f = new File("nonexistent.file")
        f.metaClass {
            exists { -> true }
            getAbsolutePath { -> "/opt/some/dir/${delegate.name}" }
            isFile { -> true }
            getText { -> "This is the text in the file." }
        }

        assertTrue f.exists()
        assertTrue f.isFile()
        assertEquals "/opt/some/dir/nonexistent.file", f.absolutePath
        assertTrue f.text.startsWith("This is")
    }
}
```

Overriding an instance method I

```
class Marine {
    def yell() {
        return """
${toString()} is my toString.
There are many like it but this is mine.
"""
    }
}

def m = new Marine()
println m.yell()
```

Overriding an instance method II

```
def marines = []

3.times { marines << new Marine() }

def newMetaClass = new ExpandoMetaClass(Marine.class)
newMetaClass.yell = { "I'm an individual"}
newMetaClass.initialize()
marines[1].metaClass = newMetaClass

for (x in marines)
    println x.yell()
```

Intercept, cache, invoke

- When using `methodMissing` a typical pattern is:
  - Intercept method call
  - Dynamically create a new method, wrapping and caching the new behaviour
  - Invoke the new behaviour
- This pattern allows the next invocation to be more performant (like memoisation)
- We can also use `invokeMethod`, as we shall see now ...

AOP with `invokeMethod` I

```
class Employee {
    def stareIntoSpace() {
        println "Employee: I'm staring into space"
    }

    def addValue() {
        println "Employee: I'm adding value"
    }
}
```

AOP with `invokeMethod` II

```
Employee.metaClass.invokeMethod = {
    String methodName, arguments ->
        def method = Employee.metaClass.getMetaMethod(methodName, args)
        println "Before"
        method.invoke delegate, args
        println "After"
        if (methodName == 'addValue')
            println "Boss: That there is some nice ${methodName}'ing"
        else
            println "Boss: Hey! ${methodName} on your own time!"
        println()
}

Employee e = new Employee()
e.stareIntoSpace()
e.addValue()
```

Flexible methods with `methodMissing`

```
class Librarian {
    def methodMissing(String name, args) {
        println "Method $name with args $args"
        def match = name =~ /findBookBy(.*)/
        if (match) {
            def mission = "Find a book with a "
            mission += "${match.group(1).toLowerCase()}" +
                    " of ${args[0]}"
            println mission
        } else {
            throw new Exception("Method ${name} not found")
        }
    }
}

def lib = new Librarian()
lib.findBookByISBN("123456")
lib.findBookByTitle("oh...")
lib.findBook("ouch!")
```

Categories

- a feature borrowed from Objective-C
- for adding functionality to classes that make them more usable within the Groovy environment
- Allows one to add methods to some instances
- EMC allows either a single instance or all instances
- Reusable class
- Best shown with an example

Categories

```
class MetaTest extends GroovyTestCase {
    void testCategory(){
        String message = "Hello"
        use(StringHelper){
            assertEquals "HELLO", message.shout()
            assertEquals "GOODBYE", "goodbye".shout()
        }

        shouldFail {
            message.shout()
            "foo".shout()
        }
    }
}

class StringHelper {
    static String shout(String self){
        self.toUpperCase()
    }
}
```

Examples

Syntactic Sugar

```
Integer.metaClass.getSeconds << { ->
    delegate * 1000
}

Integer.metaClass.getDays << { ->
    delegate.seconds * 60 * 60 * 24
}

Integer.metaClass.getWeeks << { ->
    delegate.days * 7
}

def waitFor(Integer[] mills) {
    println "Well, I'm not really going to wait " +
            "for ${GroovyCollections.sum(mills)} milliseconds"
}

waitFor(2.weeks, 5.days, 3.seconds)
```

Mocking/Stubbing behaviour

```
File.metaClass.getText = { ->
    "my canned data"
}

def readFileContents(fileName){
    new File(fileName).getText()
}

println readFileContents('blah.txt')
```

Our major example

Ice Cream example... I

```
def icecream = new IceCream()

icecream.order {
    serve_in_waffle_cone

    add(3.scoops_of_chocolate_chip)
    add(2.scoops_of_pistachio)
    add(1.scoop_of_marmite)

    make_it_snappy

    use_a_left_handed_icecream_scoop
}

println icecream
```

Ice Cream example... II

```
String.metaClass.getDescore = { ->
    delegate.replace('_', ' ')
}

Integer.metaClass.propertyMissing = { name ->
    def match = name =~ /(scoop(s)?)_of_(.*)/
    if (match)
        return [(match.group(3).descore): delegate]
}
```

Ice Cream example... III

```
class IceCream {
    def scoops = [:], notes = [], cone = ''

    def order(Closure closure) {
        closure.delegate = this
        closure()
    }

    String toString() {
        def string = "I am a delicious ice cream served in a $cone with \n"
        scoops.each {
            string += " - ${it.value} scoop${it.value > 1 ? 's' : ''}"
            string += " of ${it.key}\n"
        }
        string += "Additional Notes:\n"
        notes.each { string += " - ${it}\n" }
        return string
    }
```

Ice Cream example... IV

```
    def propertyMissing(String name) {
        def match = name =~ /(serve_in)_(.*)/
        if (match)
            cone = match.group(2).descore
        else
            notes << name.descore
    }

    def add(newScoops) {
        scoops << newScoops
    }
}
```

How is the MOP used?

- In various frameworks and design patterns
  - Spring
  - EasyMock/JMock
  - Patterns like Builder
- To circumvent the rigidity of the Java language
- Dependency Injection

Finally

The Pros and Cons

- Pros
  - Flexibility
  - Power
  - Extensibility
- Cons
  - Sometimes clunky syntax
  - Slower performance

With Great Power...

- Don't get wowed by the coolness factor
- Remember to KISS
- The end result should be more understandable rather than less!
- Keep pushing the boundaries but exercise judgement

The End