Metaprogramming

What is Metaprogramming?

Metaprogramming is when you write programs that write other programs. It's at a higher level, where you don't write code but code writes code.

- **Code Generation** - you write code that will read something and will generate other pieces of code that you can compile or run.
- **Code Synthesis** - Even more powerful than Code Generation, you don't have code at all, code becomes to life in memory, it's used and then it goes away.
    - Ruby on Rails is an example of Code Synthesis, certain functions don't exist until you query it.

> Metaprogramming allows you to insert behavior dynamically.

1. Metaprogramming in Groovy

> In languages like Groovy and Ruby, classes are **open**. Meaning that you take any class and add behavior to it. You can take objects and insert behavior into these objects.

> One way to extend behavior is through inheritance.

An example of **code injection**:

```
str = "hello"
println str

prinntln str.class

str.shout()
```

There is an error when we call the `shout` method call. But Groovy allows us to do something like this...

```
str = "hello"
println str

println str.class

String.metaClass.shout = {->
```

```
        delegate.toUpperCase()
}

println str.shout()
```

We are able to add methods to classes and objects. Now our code works as we've added the function `shout` through **code injection**.

```
class Person {
        def work() { println "working..."}
}

joe = new Person()
joe.work()
joe.sing()
```

Again we have an error. This time it's because `sing` is not a method in the `Person` class.

What most dynamic languages do is, you can write a special method for missing methods. It works as a catch all for when a method does not exist. We can then reject these missing methods.

```
class Person {
        def work() { println "working..."}

        def methodMissing(String name, args)
                println "You called methodMissing with $name"
}

joe = new Person()
joe.work()
joe.sing()
```

Instead of failing this time, we invoked the `methodMissing` function because of the `sing` function call.

> Most dynamic languages allow a similar functionality.

```
def method_missing(m, *args, &block)
end
```

```
def __missing__(self, key):
        return 'finxter'
```

Domain Specific Languages

A **Domain Specific Language (DSL)** is a language that we create with a very specific purpose within a particular domain.

- Extremely **narrow** and **focused** which can be **restrictive**
- Not a general purpose language.
  - You wouldn't use a DSL to create any arbitrary program.
- Typically very **small**, and because it's small you can design it fairly effectively. Examples of DSLs are pavement for Python and Rake for Ruby.

Key Characteristics of DSLs

1. **Fluent** - DSLs are very fluent
2. **Context** - DSLs are heavily context driven.

External vs. Internal DSLs

External DSLs are where you get to define your own language all by yourself. But the burden is that you'll have to parse it yourself.

- You have to create a parser.
- The parser provides **great validation**. Your parser will parse through your code and reject invalid syntax right at the door.

> You would typically use languages such as C#, Java, C++, or Scala. One of the heavyweight static languages to write a DSL. Because these languages come with libraries that can be used to build parsers for you.

With an **External DSL** you are in control because it is the language of your making. However to create such a language takes a lot of work.

Internal DSLs are DSLs that write on an existing language. The compiler/interpreter becomes a tool that you can leverage. Rather than creating a parser.

- The syntax that you create is a subset of the syntax of the host language.
  - For example Rake is a sub-syntax of Ruby.
- The language that hosts your internal DSL does all of the hardwork.
  - This can be a bit restrictive.
- You're able to bend the language.

    - If you want to be able to express a certain syntax you need to make the language work that way.

> For **Internal DSLs** you would like to find a language that is very fluent and very low ceremony. You want a language that forgos `.`'s, `;`'s and `{}`'s.

Take this example from Scala where we take ceremonious code and reduce its noise.

```
class Car {
        def drive(dist: Int)
        println("drive called")
}

val = new Car

car.drive(10)
```

Line number 10 can be rewritten with less ceremony.

```
class Car {
        def drive(dist: Int)
        println("drive called")
}

val = new Car

car.drive(10)

car drive 10
```

Metaprogramming is also important in creating Internal DSLs.

- In our example from Scala, we have a very expressive syntax.
    - But Scala does not provide that completely dynamic behavior.
- Metaprogramming and dynamic typing can help provide flexibility.

So to recap, for an Internal DSL you want:

1. An **expressive** language with **low ceremony**
2. **Metaprogramming**

2. DSLs in Groovy

Example of a DSL: Game Scores

```
//Name this file scores.dsl
joe 12
bob 14
jim 8
winner
```

If this was an external DSL you would write a parser for this. However, we're not going to do that. We're going to write this as if it was code.

Groovy is treating `joe` as a function call. As far as Groovy goes, it's treating this line as:

```
joe(12);
```

So what we're going to do is write a new method `methodMissing`. And let's add a hash table called `playersAndScores`.

Now pay attention to the `process` function. The `process` function takes a `dsl` as the parameter, it comes in a **closure**.

> A closure is a lexically scoped name binding.

We create a `processor`, and the code that we receive in the function will run in the context of this `Processor` object. All the methods received will be in the context of this object.

```
//Name this file Processor.groovy
def playersAndScores = {:}

def methodMissing(String name, args) {
        if (args.size() == 1) {
                playersAndScores[name] = args[0]
        }
}

def getWinner() {
        def theWinner = ""
        def maxScore = 0
        playersAndScores.each { name, score ->
                if (maxScore < score) {
                        maxScore = score
                        theWinner = name
                }
        }
        println "Winner is $theWinner with score #maxScore"
}

def process(dsl) {
        def processor = new Processor()
        processor.with dsl
}
```

Put in the DSL, put in the code, and then evaluate it. And now you'll be able to run the DSL code we showed earlier. It's magic! ~~Except it's called Groovy~~

```
dsl = new File('scores.dsl').text
code = new File('Processor.groovy').text
evaluate(code + dsl)
```

3. Parsing XML

Can you think of an example of an XML Parser?
DOM Parser - completely parses the document, validate it, and gives an object model. Then you can query and navigate.

- **DOM** stands for **Document Object Model**
  SAX Parser - as it goes through the document it fires events at you. It tells you each element and attribute it comes across. When it realizes that the document isn't well formed, it says "oops forget everything I told you".
- You have to be ready to undo all the things you did.

XPath is an alternative that we can use in Groovy and in Scala.

4. Parsing XML in Groovy

Here is the xml file that we're going to parse today.

```
<language>
        <language name="C++">
                <author>Stroustrup</author>
        </language>
        <language name="Java">
                <author>Gosling</author>
        </language>
        <language name="Lisp">
                <author>McCarthy</author>
        </language>
        <language name="Modula-2">
                <author>Wirth</author>
        </language>
        <language name="Oberon-2">
                <author>Wirth</author>
        </language>
        <language name="Pascal">
                <author>Wirth</author>
        </language>
        <language name="Ruby">
                <author>Matz</author>
        </language>
```

In Groovy, we're going to use what's called an `XmlSlurper` to "slurp up" or better yet parse through the xml above.

```
languages =
        new XmlSlurper().parse('languages.xml')
println languages.language.each {
        println it
}
```

Remember that you're using a dynamic language, so what does that mean we can use?
Metaprogramming

```
languages =
        new XmlSlurper().parse('languages.xml')
println languages.language.each {
        println it.@name
}
```

We reference the attribute in XML by using `@name`.
Notice as we parse we didn't do any ceremonious things here.

```
languages =
        new XmlSlurper().parse('languages.xml')
println languages.language.each {
        println "${it.@name} was written by ${it.author[0]}"
}
```

That's how simply it is to navigate down in a dynamic language. You can call methods directly.

- It's harder to do this in languages like Java, C# or Scala. Because of static binding.

Notice one more thing. The difference between parsing a local file versus a remote one is hardly anything. That is how simple it is in Groovy.

```
languages =
        new XmlSlurper().parse('http://www.cs.uh.edu/~svenkat/languages.xml')
println languages.language.each {
        println "${it.@name} was written by ${it.author[0]}"
}
```

4. XML Parsing in Scala

```
val str = scala.io.Source.fromFile(
        "languages.xml").mkString

val xml = scala.xml.XML.fromString(str)

println(xml \\ "language")
```

5. Creating XML

Creating XML in Groovy

Groovy gives you a DSL for building XML.

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy', 'Modula-2'
: 'Wirth', 'Oberon-2' : 'Wirth', 'Pascal' : 'Wirth', 'Ruby' : 'Matz']

bldr = new groovy.xml.MarkupBuilder()

bldr.languages{
        langs.each { k, v ->
                language(name:k) { author(v) }
        }
}
```

We'll come back to this code when we talk about pattern matching below.

Creating XML in Scala

```
val xml = <hello></hello>
println(xml.getCllass())
```

Scala says that XML should be a first class citizen. In Scala, any XML syntax is valid. That's because Scala is a superset of XML.

- You don't need to hide XML behind strings in Scala.

So let's go ahead an create XML in Scala.

```
val langs = Map(
        "C++" -> "Stroustrup",
        "Ruby" -> "Matz")

val xml = <languages>{
        langs.map { (lang, author) =>
                <language name={entry._1}>
                        <author>{entry._2</author>
                </language>
        }
}</languages>

print(xml)
```

We don't have to do that much work here since XML is treated as a first class citizen in Scala.

6. Pattern Matching

What is a pattern? Let's see for ourselves...
So you're going to get some data and take actions based on that data.

```
def process(msg: AnyRef) = {
        msg match {
                catch _ => println("Whatever")
        }
}
```

> The underscore `_` here in Scala is the same as what you see in Python. It's a convention of naming a variable that is temporary or insignificant. We don't care to name it.

```
def process(msg: AnyRef) = {
        msg match {
                catch _ => println("Whatever")
        }
}

process(5)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

Scala catches a type mismatch error from the above code.

```
found : Int(5)
required : AnyRef
Note: primitive types are not implicitly converted to AnyRef.
You can safely force boxing by casting x.asInstanceOf[AnyRef].process(5)
```

So instead let's use `Any`

```
def process(msg: Any) = {
        msg match {
                catch _ => println("Whatever")
        }
}

process(5)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

Let's add some literal based matching...

```
def process(msg: AnyRef) = {
        msg match {
                case 5 => println("high five")
                catch _ => println("Whatever")
        }
}

process(5)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

You can even add some type based matching as well

```
def process(msg: AnyRef) = {
        msg match {
                case 5 => println("high five")
                case x : Int => println("I got int " + x)
                catch _ => println("Whatever")
        }
}

process(5)
process(10)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

We can have a case for a tuple as well

```
def process(msg: AnyRef) = {
        msg match {
                case 5 => println("high five")
                case x : Int => println("I got int " + x)
                case (a, b) => println("we have " + a + " " + b)
                catch _ => println("Whatever")
        }
}

process(5)
process(10)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

There's also something called a **guarded match**, these come with an added condition for the matching.

- An example of that is below for when b equals 10.

```
def process(msg: AnyRef) = {
        msg match {
                case 5 => println("high five")
                case x : Int => println("I got int " + x)
                case (a, b) if b == 10 => println("we have " + a + " " + b)
                case (a, b) => println("we have " + a + " " + b)
                catch _ => println("Whatever")
        }
}

process(5)
process(10)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

So far we've looked at:

1. literal matching
2. type matching
3. guarded matching

```
def process(msg: AnyRef) = {
        msg match {
                // literal matching
                case 5 => println("high five")
                // type matching
                case x : Int => println("I got int " + x)
                case (a, b) => println("we have " + a + " " + b)
                case x : Double => println("double value " + x)
                case x : String => println("string " + x)
                // guarded matching
                case (a, b) if b == 10 => println("we have " + a + " " + b)
        }
}

process(5)
process(10)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

We can create a case for our XML content.

- Our XML has a root element of `<languages/>`
  Between the root element is A LOT of stuff, which is our `langs`
  `langs` has the child elements.
  When we want to process the languages, we can use **list comprehension**. Which we can see below.
- What we wrote reads as: `for each language in our root element: print it's contents`.
  Here we have a pattern matching at work, extracting data from our XML

```
def process(msg: AnyRef) = {
        msg match {
                // literal matching
                case 5 => println("high five")
                // type matching
                case x : Int => println("I got int " + x)
                case (a, b) => println("we have " + a + " " + b)
                case x : Double => println("double value " + x)
                case x : String => println("string " + x)
                // guarded matching
                case (a, b) if b == 10 => println("we have " + a + " " + b)

                // List comprehension:
                case <languages>{langs @ _*}</languages> =>
                        for (language @ <language>{_*}</language> <- langs) {
                                println((language \\ "@name") + " was written by "
                                        + (language \\ "author").text)
                        }

                catch _ => println("Whatever")
        }
}

process(5)
process(10)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)

val str2 = scala.io.Source.fromURL(
        new Java.net.URL("http://www.cs.uh.edu/~svenkat/languages.xml")).mkString
```

```
val xml2 = scala.xml.SML.loadString(str2)
process(xml2)
```

> Note: The sequence of the pattern matching cases matters. Don't look at what you see here and think that it's doing it sequential. Scala behind the scenes. It takes each of these cases and convert them into ***partially applied functions***. Each case turns into one of these functions. Behind each case there's partially applied functions.

> In short, ***partially applied functions***, you apply some parameters and leave the others out, so then the other parameters can be given later on.

The order in which Scala does that is in the order that you specify. Say we swapped the cases above to look like what we have below:

```
def process(msg: AnyRef) = {
        msg match {
                // case 5 is swapped below the type matching.
                case x : Int => println("I got int " + x)
                case 5 => println("high five")

                case (a, b) => println("we have " + a + " " + b)
                case x : Double => println("double value " + x)
                case x : String => println("string " + x)
                case (a, b) if b == 10 => println("we have " + a + " " + b)

                // List comprehension:
                case <languages>{langs @ _*}</languages> =>
                        for (language @ <language>{_*}</language> <- langs) {
                                println((language \\ "@name") + " was written by "
                                        + (language \\ "author").text)
                        }

                catch _ => println("Whatever")
        }
}
```

We will never be able to reach case 5 if we mix the order like above.