LAB 6: GROOVY: METAPROGRAMMING AND WORKING WITH XML & JSON

In this lab you will explore some interesting capabilities of Groovy. You will use runtime metaprogamming to manipulate the properties and methods of an object. You will also use builders to create XML and JSON representations of an object, and create an object using data from a web API.

For this lab you will need to create and run Groovy scripts. You can do this using the Groovy command line tools or GroovyConsole, or you can use IntelliJ (recommended). If you are using IntelliJ, you should create a project lab6project.

Task 1.Metaprogramming

Creating an object to manipulate

1. Create a new Groovy class, in your project's src folder, called *Person*. Edit the class so that it contains the following code:

   ```
   class Person {
       String name
       String password

       def resetPassword(String password) {
           this.password = password
       }
   }
   ```

2. Create a new Groovy script called *task1.groovy*. In this task you will make some modifications to the Person class. Everything else you are asked to do will involve adding further lines of code to the *task1* script.

3. Add code to the script to create an instance *p* of Person as follows:

   ```
   def p = new Person()
   ```

   and add code to set the values of the properties of *p* to *"Alice"* and *"aSecret"*. You will use this object throughout this task.

4. Add code to print the property values, and run your script.

Missing methods and properties

1. Continue your script by adding the following code, which tries to access a method *someMethod*

   ```
   p.someMethod(3, "hello")
   ```

   Run the script. What happened? Review the method resolution order for Groovy in your lecture notes to see the steps that led to this outcome.

2. Add methods *methodMissing* and *invokeMethod* to the Person class. Each should print a message indicating the name of the method called, the arguments supplied, and which method was called in response. Run the script again. Which method was invoked? Is this what you would expect?

3. Remove or comment out the call to *someMethod* in your script. Add the following call (to a method that does exist):

   ```
   p.resetPassword("newpwd")
   println(p.password)
   ```

   The *println* statement will let you see whether the call to *resetPassword* worked. Run the script. Was the *resetPassword* method called?

4. Make the following modifications to your Person class:

   - Make Person implement the interface GroovyInterceptable
   - Use *System.out.println* instead of *println* in *invokeMethod*[1]

   Run the script again. What happened? Was the *resetPassword* method called? Note that if a class implements GroovyInterceptable then an *invokeMethod* method will intercept all method calls

5. Add the following line to *invokeMethod* – this uses the object's metaclass to find and invoke the method which was called:

   ```
   metaClass.getMetaMethod(name, args).invoke(this, args)
   ```

   Run the script again. What happened? Was the *resetPassword* method called?

   Based on the results of the above steps, what do you think are the intended purposes of *invokeMethod* and *methodMissing* in Groovy?

6. Add a *propertyMissing* method to the Person class and add code to your script to test this.

[1] See http://stackoverflow.com/questions/30646899/groovy-println-vs-system-out-println-with-groovyinterceptable

Adding properties and methods at runtime

1. Add the following to your script, immediately after the existing code:

   ```
   p.metaClass.age = 21
   ```

   This adds a property *age* to your Person object through its metaclass. Test that you can print the value of the property (add a print statement and run the script).

2. Add the following code to your script.

   ```
   Scanner input = new Scanner(System.in)
   print("new property name: ")
   while ((newPropName = input.next()) != "done") {
       print("new property value: ")
       newPropVal = input.next()
       p.metaClass."${newPropName}" = newPropVal

       print("new property name: ")
   }

   println("Properties of object-")
   for (prop in p.properties) {
       println(prop)
   }
   ```

   This allows multiple properties to be added to the Person object at runtime with names and values input by the user, until the user enters *“done”* for the property name. The for loop iterates through and prints the properties of the object. Run the script and test that you can add properties to your Person object and see their names and values, along with those of the static properties.

3. You can also add methods at runtime as closures. Add the following code to your script to add a method *prettyPrint* to your person object which will list the properties and values in a tidier format:

   ```
   p.metaClass.prettyPrint = {
       println "Person object has following properties"
       p.properties.each {
           if (it.key != 'prettyPrint')
               println " " + it.key + ": " + it.value
       }
   }
   ```

   Test that you can call the method *prettyPrint* on your Person object (add a call and run the script).

4. You can add methods dynamically to any type of object, not just instances of your own types. This includes instances of Java classes. Add code to your script to do the following:

- Create a new instance of the Java Date class called *d* (you can use the Date class in Groovy without any imports)
- Define a variable *clos1* and assign to it a closure containing the same code as used for *prettyPrint* above (except that the output shouldn't refer to a "Person")
- Assign the *clos1* to your Date object using:
  `d.metaClass.prettyPrint = clos1`

Test that you can call the method *prettyPrint* on your Date object.

Closure - delegating to an object

1. Add the following code to your script to create a closure and call that closure:

   ```
   def clos2 = {
       resetPassword "aReallyBigSecret"
   }

   clos()
   ```

   (Note that the code inside the closure is equivalent to

   ```
   resetPassword("aReallyBigSecret")
   ```

   as Groovy allows parentheses to be omitted here)

   What happened? Note that the closure has no access to a method or closure called *resetPassword*.

2. Assign your Person object to be the delegate for the closure using the following code before the call to clos:

   ```
   clos.delegate = p
   ```

   Add a call to *p.prettyPrint* so that you can see any change to the Person object's properties. Run the script and check that the closure *clos2* has successfully changed the Person object's password.

Task 2. Building representations of an object

In this task you will create XML, HTML and JSON representations of a Person object, using the same Person class as in Task 1.

1. Create a new Groovy script *task2.groovy*. Download the file *task2.txt* from GCULearn and copy the code into your script. This code is essentially a part of the code for Task 1 which creates a Person object *p*, allows you to add some properties at runtime to it, and prints the property names and values. Run this script to make sure it works correctly.

   Review the documentation for the Groovy MarkupBuilder class:

   http://docs.groovy-lang.org/latest/html/api/groovy/xml/MarkupBuilder.html

2. Add code to the script to do the following:

   - Create an instance of StringWriter and assign it to a variable *writer*
   - Create an instance of MarkupBuilder, with writer as the single constructor parameter, and assign it to a variable *xml* (you will need to import groovy.xml.MarkupBuilder)

3. Use the following code to build and print an XML representation of your Person object:

```
xml.person() {
    name(p.name)
    password(p.password)
    dynamicproperties {
        for (prop in p.properties) {
            if (!
['class','name','password'].contains(prop.key))
                "${prop.key}"("${prop.value}")
        }
    }
}
println writer
```

   What do you think the purpose of the if statement is?

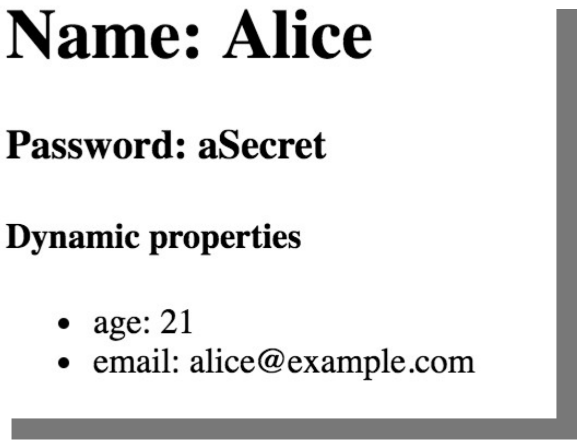
   Run the script, adding a single property names *email*, at runtime. You should get a representation of your object similar to the following – note how the code and properties correspond to the XML.

```
<person>
  <name>Alice</name>
  <password>aSecret</password>
  <dynamicproperties>
    <age>21</age>
    <email>alice@example.com</email>
  </dynamicproperties>
</person>
```

Note that the call *xml.person()* produces an XML document with a root element called *person*. The code in {} after this is a closure. Each call in the closure, e.g *name(p.name)* produces an XML element containing the parameter value, e.g. <name>Alice</name>. The *dynamicproperties* element has other elements nested inside it.

4. Add a similar block of code to create an HTML representation of the Person and run the script. You will need new StringWriter and MarkupBuilder objects, and should name the root element *html*. You should write calls inside the closure to produce HTML elements, e.g. *h1*. The result should look like this:

```
<html>
  <head>
    <title>Person object</title>
  </head>
  <body>
    <h1>Name: Alice</h1>
    <h3>Password: aSecret</h3>
    <h4>Dynamic properties</h4>
    <ul>
      <li>age: 21</li>
      <li>email: alice@example.com</li>
    </ul>
  </body>
</html>
```



Name: Alice

Password: aSecret

Dynamic properties

- age: 21
- email: alice@example.com

Create a new HTML file in your project called *person.html*, and copy the HTML from the console output into the file. Open the file in your browser (IntelliJ should give you an option to do this).

5. Now you will create a JSON (JavaScript Object Notation) representation of the Person object. Add the following code to your script. You will need to import groovy.json.JsonBuilder.

```
def json = new JsonBuilder()
json.person() {
    for (prop in p.properties) {
        "${prop.key}"("${prop.value}")
    }
}
println json.toString()
```

This creates a simple JSON representation with all properties at the same level (no nesting). Run the script to see the JSON.

6. Finally, you can easily use JSON to create a Groovy object using a JsonSlurper object. Add the following code to your script to do this and print the properties of

the object, which is actually a Map instance. You will need to import groovy .json.JsonSlurper.

```
def slurper = new JsonSlurper()
def result = slurper.parseText(json.toString())

result.each { key, value ->
    println "Object of type $key "
    value.each { k, v ->
        println "$k : $v"
    }
}
```

Run the script and check that the properties (keys) have the same values as the original Person object.

Task 3. Getting data from a REST API

XML and JSON formats are widely used for making data available from a REST web API. A REST API allows programs to retrieve data by making HTTP requests. The URL of the request specifies what data the program wants to retrieve. Groovy makes it easy to work with REST APIs.

In this task you will use a “dummy” API which is available for testing and prototyping[2]. While the data is not meaningful, this is a “live” service which is accessed over the internet.

1. Enter the following URL in your browser address bar.

   https://jsonplaceholder.typicode.com/photos/1

   You should see the following data, in JSON format. This is the data you will consume from a Groovy script.

   ```
   {
     "albumId": 1,
     "id": 1,
     "title": "accusamus beatae ad facilis cum similique qui sunt",
     "url": "http://placehold.it/600/92c952",
     "thumbnailUrl": "http://placehold.it/150/30ac17"
   }
   ```

2. This data represents a photo. We want to use the data to create a Groovy object which is an instance of a class Photo with properties matching those in the JSON

[2] https://jsonplaceholder.typicode.com/

data. Create a new Groovy class Photo in the *src* folder in your project, with the properties *albumId*, *id* (both ints), *title*, *url* and *thumbnailUrl* (all Strings)

3. Create a new Groovy script *task3.groovy* in your project. Add the following code to retrieve the data:

```
URL url =
    new
URL("https://jsonplaceholder.typicode.com/photos/4")
def data = url.getText(requestProperties: [
    'User-Agent': 'Groovy Sample Script'])
```

4. Add the following code to parse the JSON data and use it to create a Groovy object, as you did in Task 2. This object is, as before, a Map instance.

```
def slurper = new JsonSlurper()
def objectFromJson = slurper.parseText(data)
```

5. Groovy allows you to provide a Map as a constructor parameter for a Groovy class and it will match the map keys to the class properties and create an instance with the appropriate property values taken from the Map. Add the following code to do this and print the properties of the Photo object.

```
Photo ph = new Photo(objectFromJson)
ph.properties.each {
    println " " + it.key + ": " + it.value
}
```

   Run the script and check that the properties match the data from the API. In the IntelliJ console you should be able to click on the *url* and *thumbnailUrl* property values and see the relevant photos (they're not very exciting, actually) in your browser.