



### 클래스 정의와 스크립트

#### Table of Contents

- 클래스 정의와 스크립트
  - 그루비에서 필드 변수의 기본 접근 영역은 특별하게 다루어진다.
    - 1. 필드를 정의할 때 접근 제한자를 지정하지 않으면 이름에 맞는 '프로퍼티'가 만들어진다 (.getter, .setter)
    - 2. 그루비는 자바의 접근제한자를 쓸수있다?
    - 3. 배열 첨자 연산자로 필드 변수 사용하기
    - 4. 필드 접근 메커니즘 확장하기
  - 메서드와 인자
    - 5. 메서드 정의하기
    - 6. 인자리스트 정의 인자에 따른 메서드 호출. 다르게
    - 7. 고급인자 사용 기술
    - 8. 안전한 참조연산자
  - 생성자들
    - 9. 위치기반 인자를 이용한 생성자
    - 이름기반 이자로 생성자 호출하기
  - 클래스와 스크립트 구성하기
    - 패키지로 구조화하기
    - 클래스패스
      - 14. 클래스 별칭
    - 클래스 패스에 관련된 추가 사항
  - 고급 객체지향 기능
    - 16. 멀티메서드
    - 17. equals를 선택적으로 재정의하는 메서드
    - 18. 그루비빈 사용하기
    - 19. getter, setter 메서드 만들기 만해도 그루비 프로퍼티 접근형식으로 접근가능
    - 20. 프로퍼티 함수를 통하여 접근하지 않고! @기호로 필드 변수에 바로 접근하기.
  - 자바빈 스타일 이벤트처리
  - 필드, 접근자, 맵, Expando
    - 21. 프러퍼티 접근 및 get, set 메서드 활용 예제
  - 그루비 특징점 활용하자
    - 22. GPath는 객체들의 구조를 탐색하는 강력한 도구이다. XPath이름을 따서 왔다나..?
  - 23. 확산 연산자 사용하기
    - 23. 확산연산자 예제.
  - 24. 키워드 use를 이용한 카테고리 섞기
    - 25. use 메서드 사용하기
  - 그루비 메타 프로그래밍
    - 메타 프로그래밍 이해

<http://goo.gl/rRzINX>

크고 복잡하고 요란하게 만드는 건 헛똑똑이들이 하는 짓이다. 그렇게 하지 않기 위해서는 천재적인 솜씨와 용기가 필요하다. - 알베르트 아인슈타인

여기까지는 자바보다 문법이 좀 편할 뿐이라는 느낌이 들것이다.  
이제 그루비 답게 객체나 클래스를 실행중에 수정, 추가 하고, 메서드 호출을 가로채는등 일을 해보자.

## 그루비에서 필드 변수의 기본 접근 영역은 특별하게 다루어진다.

### 1. 필드를 정의할 때 접근 제한자를 지정하지 않으면 이름에 맞는 '프로퍼티'가 만들어진다.(getter, setter)

```
1 class SomeClass {
2
3     public    public_var='old_public_var_val'
4     String    string_var='old_string_var_val'
5     def        def_var = 'old_def_var_val' //필드를 정의할 때 접근 제한자를 지정하지 않으면 이름에 맞는 '프로퍼티'가 만들어진다.(getter, setter)
6     static    static_var='old_static_var_val'
7     protected protected_var1, protected_var2, protected_var3
8     private    assignedField = new Date()
9     public static final String CONSTA = 'a', CONSTB = 'b'
10
11 }
12
13 SomeClass somclass = new SomeClass();
14 println 'b_somclass.public_var : '+somclass.public_var;
15 println 'b_somclass.string_var : '+somclass.string_var;
16 println 'b_somclass.def_var : '+somclass.def_var;
17
18 println ";
19 somclass.public_var = 'new_public_var_val';
20 somclass.string_var = 'new_string_var_val';
21 somclass.def_var = 'new_def_var_val';
22 println ";
23
24
25 println 'a_somclass.public_var : '+somclass.public_var;
26 println 'a_somclass.string_var : '+somclass.string_var;
27 println 'a_somclass.def_var : '+somclass.def_var;
28
29
30 println ";
31 println SomeClass.class.methods.name.grep(~/[get].*/); //get으로 시작하는 메서드이름 가져와보자
32 println SomeClass.class.methods.name.grep(~/[set].*/); //set으로 시작하는 메서드이름 가져와보자
33
34 /* 결과
35     b_somclass.public_var : old_public_var_val
36     b_somclass.string_var : old_string_var_val
37     b_somclass.def_var : old_def_var_val
38
39
40     a_somclass.public_var : new_public_var_val
41     a_somclass.string_var : new_string_var_val
42     a_somclass.def_var : new_def_var_val
43
44     [getMetaClass, this$dist$invoke$2, this$dist$set$2, this$dist$get$2, getString_var, getDef_var, getStatic_var, getProperty, equals, toString,
45     [setMetaClass, this$dist$invoke$2, this$dist$set$2, this$dist$get$2, super$1$wait, super$1$wait, super$1$wait, super$1$toString]
46 */
```

위결과를 보면 자동으로 생성된 프로퍼티는 `getString_var`, `getDef_var`, `getStatic_var` `setString_var`, `setDef_var`, `setStatic_var` 가생성된걸 볼수 있다.

### 2. 그루비는 자바의 접근제한자를 쓸수있다?

```
1 package _2.sub
2
3 class GClass {
4     private private_var='private_var';
5     public public_var='public_var';
6     protected protected_var='protected_var';
7     def def_var = 'def_var';
8 }
```

```
1 package _2
2
3 import _2.sub.GClass;
4 GClass g = new GClass();
5 println "${g.private_var.class} : ${g.private_var}";
6 println "${g.public_var.class} : ${g.public_var}"; //자바에서는 이것만 허용이 된것이다.
```

```

7 println "${g.protected_var.class} : ${g.protected_var}";
8 println "${g.def_var.class} : ${g.def_var}";
9 /*
10 결과
11 class java.lang.String : private_var
12 class java.lang.String : public_var
13 class java.lang.String : protected_var
14 class java.lang.String : def_var
15 */

```

접근제한자를 썼지만 자유롭게 접근이 가능하다.

### 3. 배열 첨자 연산자로 필드 변수 사용하기

```

1 package _3
2
3 class SomeClass {
4
5     public public_var='old_public_var_val'
6     String string_var='old_string_var_val'
7     def def_var = 'old_def_var_val' //필드를 정의할 때 접근 제한자를 지정하지 않으면 이름에 맞는 '프로퍼티'가 만들어진다.(getter,setter)
8     static static_var='old_static_var_val'
9     protected protected_var1, protected_var2, protected_var3
10    private assignedField = new Date()
11    public static final String CONSTA = 'a', CONSTB = 'b'
12
13 }
14
15 SomeClass somclass = new SomeClass();
16 println 'b_somclass.public_var : '+somclass['public_var'];
17 println 'b_somclass.string_var : '+somclass['string_var'];
18 println 'b_somclass.def_var : '+somclass['def_var'];
19
20 println ";
21 somclass['public_var'] = 'new_public_var_val';
22 somclass['string_var'] = 'new_string_var_val';
23 somclass['def_var'] = 'new_def_var_val';
24 println ";
25
26
27 println 'a_somclass.public_var : '+somclass['public_var'];
28 println 'a_somclass.string_var : '+somclass['string_var'];
29 println 'a_somclass.def_var : '+somclass['def_var'];
30
31
32 println ";
33
34 /* 결과
35 b_somclass.public_var : old_public_var_val
36 b_somclass.string_var : old_string_var_val
37 b_somclass.def_var : old_def_var_val
38
39
40 a_somclass.public_var : new_public_var_val
41 a_somclass.string_var : new_string_var_val
42 a_somclass.def_var : new_def_var_val
43 */

```

### 4. 필드 접근 메커니즘 확장하기

```

1 package _4
2
3 class SomeClass {
4
5     public public_var='old_public_var_val'
6     String string_var='old_string_var_val'
7     def def_var = 'old_def_var_val' //필드를 정의할 때 접근 제한자를 지정하지 않으면 이름에 맞는 '프로퍼티'가 만들어진다.(getter,setter)
8     static static_var='old_static_var_val'
9     protected protected_var1, protected_var2, protected_var3
10    private assignedField = new Date()
11    public static final String CONSTA = 'a', CONSTB = 'b'
12
13    def notVar_GetCallCount=0;
14    def notVar_SetCallCount=0;
15
16    Object get (String name) {
17        notVar_GetCallCount++;
18        println "get : ${name}";
19        return 'pretend value'
20    }
21    void set (String name, Object value) {
22        notVar_SetCallCount++;
23        println "set name : ${name}, value : ${value}";
24    }
25
26 }

```

```

27 SomeClass somclass = new SomeClass();
28 println 'b_somclass.public_var : '+somclass.public_var;
29 println 'b_somclass.string_var : '+somclass.string_var;
30 println 'b_somclass.def_var : '+somclass.def_var;
31 println 'b_somclass.not_var: '+somclass.not_var;
32
33 println "";
34 somclass.public_var = 'new_public_var_val';
35 somclass.string_var = 'new_string_var_val';
36 somclass.def_var = 'new_def_var_val';
37 somclass.not_var = 'new_not_var_val';
38 println "";
39
40
41 println 'a_somclass.public_var : '+somclass.public_var;
42 println 'a_somclass.string_var : '+somclass.string_var;
43 println 'a_somclass.def_var : '+somclass.def_var;
44 println 'a_somclass.not_var : '+somclass.not_var;
45
46
47
48 println "";
49 println 'somclass.notVar_GetCallCount : ' + somclass.notVar_GetCallCount;
50 println 'somclass.notVar_SetCallCount : ' + somclass.notVar_SetCallCount;
51
52 /* 결과
53 b_somclass.public_var : old_public_var_val
54 b_somclass.string_var : old_string_var_val
55 b_somclass.def_var : old_def_var_val
56 get : not_var
57 b_somclass.not_var: pretend value
58
59 set name : not_var, value : new_not_var_val
60
61 a_somclass.public_var : new_public_var_val
62 a_somclass.string_var : new_string_var_val
63 a_somclass.def_var : new_def_var_val
64 get : not_var
65 a_somclass.not_var : pretend value
66
67 somclass.notVar_GetCallCount : 2
68 somclass.notVar_SetCallCount : 1
69 */

```

여기서 중요하게 볼것이 존재하지 않는 프로퍼티를 접근할 때 get 이라는 메서드를 통하여 접근한다  
존재하지 않는 프로퍼티에 변수값을 지정할 때 set('name',Object)로 접근한다는것도 잊지말여라.

## 메서드와인자

자바의 접근제한자를 쓸수 있으며 리턴형은 생략가능하다. 접근제한자나 리턴형을 지정하지 않을 때는 키워드 def를 사용한다.  
def를 쓰면 리턴값의 자료형이 지정되지 않았다고 생각할수 있다 (물론, 리턴이 없는 void메서드일수도 있다)  
내부적으로 java.lang.Object가 리턴된다. 접근제한자는 def로 선언시 public선언된다.

## 5.메서드정의하기

```

1 package _5
2
3 class SomeClass {
4     static void main (args){                // #1 자동으로 public 접근제한자 없으므로 자바 메인메서드가 된다.
5         println "call main"
6         def some = new SomeClass()
7         some.publicVoidMethod()
8         assert 'hi' == some.publicUntypedMethod()//true
9         assert 'ho' == some.publicTypedMethod()//true
10        combinedMethod()                    // #2 현재 클래스의 정적 메서드 호출
11    }
12    static void main2 (args){
13        args+'before';
14        args+'after';
15    }
16    static main3 (args){
17        args+'before';
18        args+'after';
19    }
20    void publicVoidMethod(){
21    }
22    def publicUntypedMethod(){
23        return 'hi'
24    }
25    String publicTypedMethod() {
26        return 'ho'
27    }
28    protected static final void combinedMethod(){
29    }
30 }
31 //println '-----'; //이부분 주석을 풀게되면 이클래스는 Script를 상속받게된다 따라서 run메서드를 실행시키게된다.
32 /* 결과
33 call main

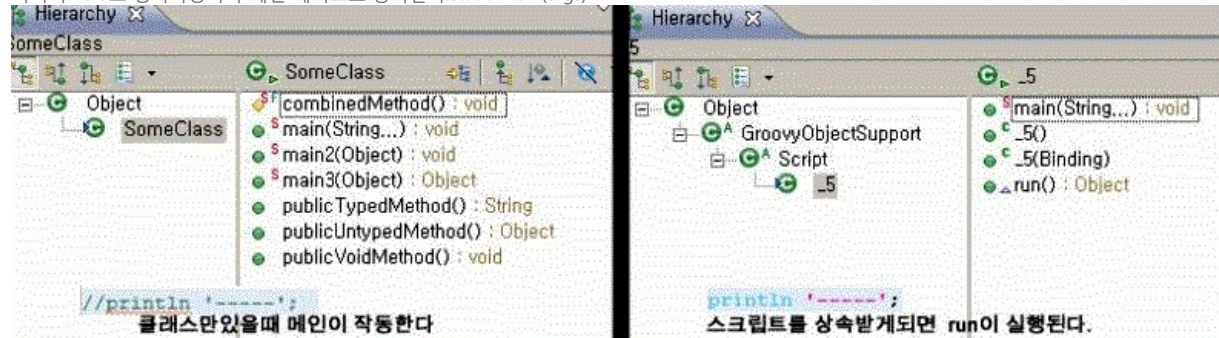
```

main 메서드에 흥로운 것이 있다

첫째는 : public 디폴트이기 때문에 생략

둘째는 : 실행할 수 있는 클래스의 메인메서드는 인자가 String[] 이어야 한다. 여기서 arg가 암묵적으로 Object 가 되는데도 그루비의 메서드 디스패치 덕분에 메인 메서드로 동작한다.

여기서 void도 생략가능하다 메인 메서드로 동작한다.static main(args)



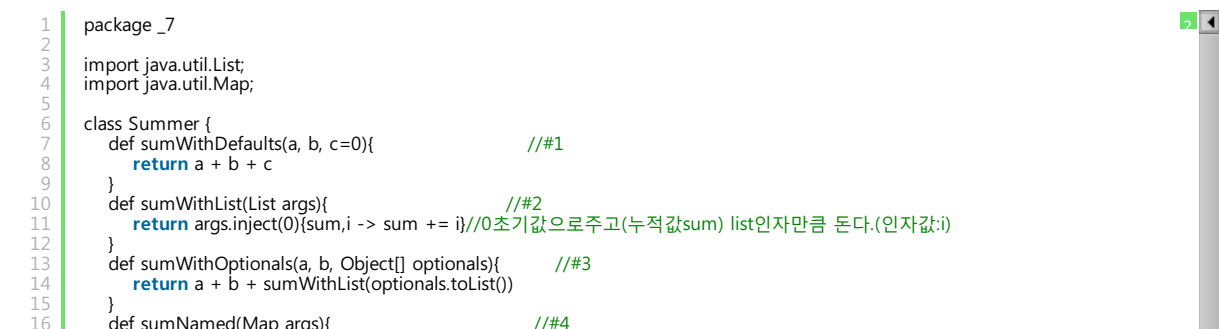
## 6. 인자리스트 정의 인자에 따른 메서드 호출 다르게



### 오버로딩 같은개념

여기서 맨마지막 method(1,2,3) 호출이 오류난건 자바에서처럼 메인메서드에서 호출하려는 메서드는 static이어야한다 그걸 위반했다.

## 7. 고급인자 사용 기술



```

17     [a,b,c].each{args.get(it,0)} //초기값부여
18     return args.a + args.b + args.c
19 }
20 def sumNamed(Map args,Map args1,Map args2){ // #4
21     return 3;
22 }
23 }
24
25 def summer = new Summer()
26
27 // #1호출
28 assert 2 == summer.sumWithDefaults(1,1)
29 assert 3 == summer.sumWithDefaults(1,1,1)
30 println 'summer.sumWithDefaults(1,1) :'+summer.sumWithDefaults(1,1)
31 println 'summer.sumWithDefaults(1,1,1) :'+summer.sumWithDefaults(1,1,1)
32
33 // #2호출
34 assert 2 == summer.sumWithList([1,1])
35 assert 3 == summer.sumWithList([1,1,1])
36 println 'summer.sumWithList(1,1) :'+summer.sumWithList([1,1])
37 println 'summer.sumWithList([1,1,1]) :'+summer.sumWithList([1,1,1])
38
39 // #3호출
40 assert 2 == summer.sumWithOptionals(1,1)
41 assert 3 == summer.sumWithOptionals(1,1,1)
42 assert 6 == summer.sumWithOptionals(1,1,1,1,1,1)
43 println 'summer.sumWithOptionals(1,1) :'+summer.sumWithOptionals(1,1)
44 println 'summer.sumWithOptionals(1,1,1) :'+summer.sumWithOptionals(1,1,1)
45 println 'summer.sumWithOptionals(1,1,1,1,1,1) :'+summer.sumWithOptionals(1,1,1,1,1,1)
46
47 // map호출
48
49 assert 2 == summer.sumNamed(a:1, b:1)
50 assert 3 == summer.sumNamed(a:1, b:1, c:1)
51 assert 1 == summer.sumNamed(c:1)
52 assert 3 == summer.sumNamed([a:1, b:1, c:1])
53 assert 3 == summer.sumNamed([a:1, b:1, c:1],[a:1, b:1, c:1],[a:1, b:1, c:1])
54 println 'summer.sumNamed(a:1, b:1) :'+summer.sumNamed(a:1, b:1)
55 println 'summer.sumNamed(a:1, b:1, c:1) :'+summer.sumNamed(a:1, b:1, c:1)
56 println 'summer.sumNamed(c:1) :'+summer.sumNamed(c:1)
57 println 'summer.sumNamed([a:1, b:1, c:1]) :'+summer.sumNamed([a:1, b:1, c:1])
58 println 'summer.sumNamed([a:1, b:1, c:1],[a:1, b:1, c:1],[a:1, b:1, c:1]):'+summer.sumNamed([a:1, b:1, c:1],[a:1, b:1, c:1],[a:1, b:1, c:1])
59
60 /*
61 결과
62 summer.sumWithDefaults(1,1) :2
63 summer.sumWithDefaults(1,1,1) :3
64 summer.sumWithList(1,1) : 2
65 summer.sumWithList([1,1,1]) : 3
66 summer.sumWithOptionals(1,1) :2
67 summer.sumWithOptionals(1,1,1) :3
68 summer.sumWithOptionals(1,1,1,1,1,1) :6
69 summer.sumNamed(a:1, b:1) :2
70 summer.sumNamed(a:1, b:1, c:1) :3
71 summer.sumNamed(c:1) :1
72 summer.sumNamed([a:1, b:1, c:1]) :3
73 summer.sumNamed([a:1, b:1, c:1],[a:1, b:1, c:1],[a:1, b:1, c:1]):3
74
75 */
76

```

여기서 눈여겨볼것... 3가지있는데.

1. 디폴트값

```
def sumWithDefaults(a, b, c=0){}
```

2. 파라미터 집합형자료형 ,로 바로넣기(동적 메시지 디스패처가 넘치는 인자들을 배열에 담아서 전달)

```
def sumWithOptionals(a, b, Object[] optional){}
```

```
assert 6 == summer.sumWithOptionals(1,1,1,1,1,1)
```

3. 집합자료형 , 바로넣기 2 (동적 메시지 디스패처가 넘치는 인자들을 배열에 담아서 전달)

```
def sumNamed(Map args){}
```

```
sumNamed(Map args,Map args1,Map args2){}
```

```
assert 3 == summer.sumNamed(a:1, b:1, c:1)
```

```
assert 3 == summer.sumNamed([a:1, b:1, c:1])
```

```
assert 3 == summer.sumNamed([a:1, b:1, c:1], [a:1, b:1, c:1], [a:1, b:1, c:1])
```

#### 고급명명기법

```

1 class g{
2     def g=55;
3 }
4 Map args = [ a:1,b:2,c:3,d:4]
5 println args.'size'()
6 println new g().'a';

```

역시 여기서도 스크립트의 파워풀한기능을 볼수있다.

## 8. 안전한 참조연산자

```
1 package _8
2
3 def map = [a:[b:[c:1]]]
4
5 assert map.a.b.c == 1 //일반적인접근
6
7 if (map && map.a && map.a.x){ // #1 평가단축기법
8     assert false;
9     assert map.a.x.c == null
10 }
11
12 //널포인트 발생에 따른 예외처리를 해줘야한다.. 일반적인처리.. a안에는x가 없으니..
13 try {
14     assert map.a.x.c == null
15 } catch (NullPointerException npe){ // #2
16     println "NullPointerException : ${npe}"
17 }
18
19 //안전하게 참조하는 ? 연산자를 제공한다 이연산자는 앞에 있는 참조 변수가 null이면 현재 해석중인 표현식을 중지하고 null리턴한다.
20 assert map?.a?.x?.c == null // #3
21 println 'map?.a?.x?.c '+map?.a?.x?.c
22 /*
23 결과
24 NullPointerException : java.lang.NullPointerException: Cannot get property 'c' on null object
25 map?.a?.x?.c null
26 */
```

괜찮은 기능이기긴 하지만 저 안전한 참조연산자를 쓸정도로 애매모호한 구조를 설계하지말아야 될것같은 느낌이다.

## 생성자들

생성자는 세가지 방법으로 호출할수 있다

1. 자바에서 하던방식
2. 키워드 as, asType를 이용한 강제형변환
3. 암묵적 형변환 방식

## 9. 위치기반 인자를 이용한 생성자

```
1 package _9
2
3 import java.util.Map;
4
5 class VendorWithCtor {
6     String name, product
7     VendorWithCtor(){
8         println "디폴트 생성자"
9     }
10
11     VendorWithCtor(Map map){
12         println "Map 파라미터1개 생성자호출 : ${map}, ${map.class.name}"
13         this.name = name;
14     }
15     VendorWithCtor(name){
16         println "파라미터1개 생성자호출 : ${name}, ${name.class.name}"
17         this.name = name;
18     }
19     VendorWithCtor(name, product) { // #1
20         println "파라미터2개 생성자호출 : ${name}, ${product}"
21         this.name = name
22         this.product = product
23     }
24 }
25
26 def first = new VendorWithCtor('Canoo','ULC_일반적인 생성') // #2 일반적인 생성자호출
27 def first2 = new VendorWithCtor() // #2 일반적인 생성자호출
28
29 def second = ['Canoo','ULC_강제 형변환'] as VendorWithCtor // #3 강제 형변환
30 // def second = ['Canoo','ULC'].asType(VendorWithCtor) // #3 강제 형변환
31 def second2 = [] as VendorWithCtor // #3 강제 형변환
32 Map map = [a:1,b:2,c:3];
33 def second3 = map as VendorWithCtor // #3 강제 형변환 이걸안된다.. 디폴트가탄다.. Map이있지만도..
34
35 VendorWithCtor third = ['Canoo','ULC_암묵적 형변환'] // #4 암묵적 형변환
36 VendorWithCtor third2 = ['Canoo_암묵적 형변환'] // #4 암묵적 형변환
37
38
39 /*
40 결과
41 파라미터2개 생성자호출 : Canoo, ULC_일반적인 생성
42 디폴트 생성자
43 파라미터1개 생성자호출 : [Canoo, ULC_강제 형변환], java.util.ArrayList
```

```

44 파라미터1개 생성자호출 : [], java.util.ArrayList
45 디폴트 생성자
46 파라미터2개 생성자호출 : Canoo, ULC_암묵적 형변환
47 파라미터1개 생성자호출 : Canoo 암묵적 형변환, java.lang.String
48 */

```

역시 자바와다르게 형변환할때도 생성자 호출이된다... 하지만 저 해쉬맵은 왜안될까...

역시 자바와다르게 스크립트언어이기 때문에 개발시 부분부분 테스트가 더욱더 세밀하게 이뤄져야할것같다.

## 이름기반 이자로 생성자 호출하기

```

1 package _10
2
3 class Vendor {
4     //String name, product
5     def name, product
6     //private String name, product //private 해도 적용됨
7 }
8
9 new Vendor()
10 new Vendor(name: 'Canoo')
11 new Vendor(product:'ULC')
12 new Vendor(name: 'Canoo', product:'ULC')
13 new Vendor(name: 'Canoo', product:1) //자동캐스팅
14 //new Vendor(name: 'Canoo', product2:1) //오류남
15
16 def vendor = new Vendor(name: 'Canoo')
17 assert 'Canoo' == vendor.name
18 println vendor.name;
19
20
21 //암묵적 생성
22 java.awt.Dimension area;
23 area = [20,100];
24 assert area.width ==20;
25 assert area.height == 100;
26 println 'area.width : '+area.width ;
27 println 'area.height : '+area.height ;
28
29 /*
30 결과
31 Canoo
32 area.width : 20.0
33 area.height : 100.0
34 */
35

```

위처럼 사용할수도있다.. 하지만 위 같은 상황을 쓸일이 많이 생길지 의문이다..

## 클래스와 스크립트 구성하기

그루비 클래스 생성을 할수 있다.

1. 그루비 파일에 클래스 정의가 '하나도 없다면' 그파일은 스크립트로 동작한다.

다시말해 자동으로 Script 클래스를 상속받는다.

자동으로 생성된 클래스의 이름은 확장자를 뺀 소스 파일의 이름과 같다.

파일의 코드는run메서드에 들어가고, 실행할 수 있겠끔 main 메서드도 만들어진다.

2. 그루비 파일에 클래스가 '하나' 만 있고, 그 클래스 이름이 확장자를 제외한 소스 파일의 이름과 같다면 자바와 같은 일대일 대응 관계가 된다.

3. 그루비는 그루비 파일에 클래스가 '몇개' 정의되어있든 그것들이 public이든 private이든 소스파일의 이름과 같은 클래스가 있던 없던 다수용할수 있다. 그루비 컴파일러인 groovyc는 파일에 정의된 클래스 각각에 대해 \*.class 파일을 생성한다. 이파일을 프롬프트에서 goovy를 통해서 실행하거나 IDE에서 호출하는 식으로 스크립트로서 사용할 때는 파일에서 첫번째로 정의된 클래스에 main 메서드가 있어야한다.

4. 그루비 파일에 클래스 정의와 스크립트 코드를 섞어서 쓸수도 있다 이렇게 하면 실행시에 주 클래스는 스크립트 코드가 된다 따로 소스 파일이 이름과 같은 이름으로 클래스를 만들 필요는 없다.

명시적으로 컴파일 하지 않았다면 그루비에서 클래스를 찾을 때는 클래스 이름 과 같은 \*.groovy소스 파일을 검색한다. 이때 이름이 중요하다 그루비는 찾고 있는 클래스와 이름이 같은 파일만 검색한다 , 파일을 발견하면 그파일 안에 모든 클래스를 분석해서 로딩한다.

## 패키지로 구조화하기

\*.groovy소스 파일들은 \*.class파일로 컴파일하지 않아도 쓸수있기때문에 클래스를 찾을때 \*.groovy 파일도 함께 찾는다. 이때도 같은 방식으로 검색한다 즉 그루비는 business/Vendor.groovy파일에서 business 패키지에 속한 Vendor 클래스를 검색할것이다.



## 클래스패스

그루비가 \*.groovy 파일을 찾을때 클래스 패스를 사용한다.

주어진 클래스를 찾다가 \*.class 와 \*.groovy 모두 발견됐다면 둘중 최근에 변경된 파일을 사용한다 즉 \*.groovy 가 \*.class보다 최근에 변경됐다면 \*.groovy 컴파일한후 \*.class를 사용한다.

### 패키지예제

```
1 package business
2
3 class Vendor {
4     public String name
5     public String product
6     public Address address = new Address()
7 }
8
9 class Address {
10    public String street, town, state
11    public int zip
12 }
```

### 임포트예제

```
1 import business.*
2
3 def canoo = new Vendor()
4 canoo.name = 'Canoo Engineering AG'
5 canoo.product = 'UltraLightClient (ULC)'
6
7 assert canoo.dump() =~ /ULC/
```

몇몇 스크립트 언어와 달리 임포트를 해도 클래스나 파일을 실제로 포함하는것이 아니다 다만 클래스 이름을 해석할 때 참고할 정보를 주는것뿐이다.

그루비는 패키지 6개와 클래스2개를 자동 임포트한다.

```
java.lang.*
java.util.*
java.io.*
java.net.*
groovy.lang.*
groovy.util.*
java.math.BigInteger
java.math.BigDecimal
```

## 14.클래스 별칭

as 를통하여 클래스 별칭(type aliasing)을 만들수 있다.

이것은 클래스 이름 충돌을 해결하거나 지역적인 수정 혹은 써드파트 라이브러리 버그를 수정하는데 사용한다.

```
1 package _14.oldpack
2
3 public class _14_A {
4     public def calc(def a, def b){
5         return a+b;
6     }
7 }
```

```
1 package _14
2
3 import _14.oldpack._14_A as GO;
4
5 class A_14_A extends GO {
6     public def calc(def a, def b){
7         return a*b;
8     }
9 }
10 def f = new A_14_A();
11 println f.calc(4,4);
```

## 클래스 패스에 관련된 추가 사항

그루비가 \*.class 파일과 \*.groovy 파일에서 클래스를 찾아낸다는 점은 그루비를 다룰때 이해하고 있어야 하는 중요한 부분이다. 안타깝게도 여기서

문제가 종종 발생하는데...

그루비의 클래스패스는 %GROOVY\_HOME%\conf 디렉터리의 특별한 설정파일이 있다.  
groovy-starter.conf

맨마지막줄 #지워서 활성화시키면 좋은 기능이 살아난다.

user.home으로 상정되는 사용자의 홈 디렉터리에서 서브디렉터리로 .groovy/lib를 만들고 \*.class나 \*.jar 파일 넣어두면 그루비가 쓸때마다 로딩되도록 한다.

user.home찾기힘들다면

groovy -e "println System.properties['user.home']"

설제목	설제목	설제목
구분	정의	목적과 사용법
JDK/JRE	%JAVA_HOME%/lib,%JAVA_HOME%/lib/ext	JRE부트 클래스패스와 확장 라이브러리들
OS설정	CLASSPATH 변수	일반적인 기본설정
커맨드라인	CLASSPATH 변수	특수설정
java	-cp,--classpath,option	실행시 설정
Groovy	%GROOVY_HOME%/lib	그루비 실행환경
Groovy	-cp	그루비 실행시 설정
Groovy	.	현재 디렉터리 클래스패스로 하는 디폴트 클래스 패스

## 고급 객체지향 기능

### 1. 상속하기

그루비에서는 그루비와 자바의 클래스나 인터페이스를 상속받아서 확장할 수 있다. 자바 쪽에서도 그루비 클래스나 인터페이스를 상속받을 수 있다.

### 2. 인터페이스

자바의 인터페이스를 완벽하게 지원한다.

자바의 추상 메서드도 지원한다.

그루비는 더 동적으로 인터페이스를 사용할 수 있는 기능을 제공한다. 메서드가 하나만 있는 인터페이스인 MyInterface와 클로저 myClosure가 있다면 이 myClosure를 키워드 as 이용하여 MyInterface로 강제 형변환할 수도 있다

## 16.멀티메서드

자바에 메서드를 호출하면 명시한 자료형을 참조해서 메서드를 찾는다 그에 반해 그루비에서는 인자의 동적 자료형을 고려해서 적절한 메서드를 찾아낸다 그루비의 이런 기능을 멀티메서드(multimethod)라고 한다

```
1 package _16
2
3 def oracle(Object o) { return 'object' }
4 def oracle(String o) { return 'string' }
5
6 Object x = 1
7 Object y = 'foo'
8
9 assert 'object' == oracle(x)
10 assert 'string' == oracle(y) // #1 자바라면 'object'를 호출할것이다.
11 println oracle(x);
12 println oracle(y);
13 println oracle(y as Object);
14 /* 결과
15 object
16 string
17 object
18 */
```

인자x는 Object로 표시됐지만 동적으로는 Integer이다.

인자y는 Object로 표시됐지만 동적으로는 String이다.

명시적으로 가도록 선언하고 싶다면 명시적으로 형변환을 해주면된다.

그루비에서는 인자의 자료형을 동적으로 검사한다.

## 17.equals를 선택적으로 재정의하는 메서드

```
1 package _17
2
3 class Equalizer {
4     boolean equals(Equalizer e) { //이건 equals의 오버라이딩 된것이 아니라. 오버로딩된것이다.
5         println 'equals Equalizer : '+e.class
6         return true
7     }
8 }
```

```

8 // boolean equals(e){ //이건 equals의 오버라이딩 된것이 아니라. 오버로딩된것이다.
9 //     println 'equals Object : '+e.class
10 //     return true
11 // }
12 }
13
14 Object same = new Equalizer()
15 Object other = new Object()
16 //여기 객체를 Object로 받았다.
17
18 assert new Equalizer().equals( same ) //Equalizer에서 정의한 equals로 간다. //그루비는 자동으로 클래스형대로 찾아서간다
19 assert ! new Equalizer().equals( other ) // Object의 equals 메서드를 호출한다. //그루비는 자동으로 클래스형대로 찾아서간다
20 /*결과
21 equals Equalizer : class _17.Equalizer
22 */

```

뭐가 다른지 자바를 보자

```

1 import GG;
2 import java.util.Date;
3
4
5 class GG{
6
7     public boolean equals(String obj) {
8         System.out.println("equals String");
9         return true;
10    }
11 }
12
13 public class TestJava {
14     public static void main(String[] args) {
15         GG g = new GG();
16
17         Object s = new String("----");
18         Object d = new Date(0);
19
20         System.out.println( g.equals(s) );
21         System.out.println( g.equals(d) );
22
23         System.out.println( g.equals((String)s) );
24     }
25 }
26
27 /*결과
28 false
29 false
30 equals String
31 true
32 */

```

## 18.그루비빈 사용하기

```

1 package _18
2
3 import java.io.Serializable
4
5 class MyBean implements Serializable {
6     def untyped
7     String typed
8     def item1, item2
9     def assigned = 'default value'
10    //접근제한자가 붙은건 자동프로퍼티가 생기지 않는다.
11 }
12
13 def bean = new MyBean()
14 assert 'default value' == bean.getAssigned()
15 println bean.getAssigned();
16
17 bean.setUntyped('some value')
18 assert 'some value' == bean.getUntyped()
19 println bean.getUntyped();
20
21 bean = new MyBean(typed:'another value',untyped:'untyped---',item1:'item1---')
22 assert 'another value' == bean.getTyped()
23 assert 'untyped---' == bean.getUntyped()
24 assert 'item1---' == bean.getItem1()
25 println bean.getTyped();
26 println bean.getUntyped();
27 println bean.getItem1();
28
29 /*
30 결과
31 default value
32 some value
33 another value
34 untyped---
35 item1---
36 */

```

여기서 중요하게 볼것이 생성할때 프로퍼티를 통하여 초기값을 줄수 있다는것이다.

#### 읽기전용 프로퍼티

```
1 package _18
2 import java.io.Serializable
3
4 class MyBean implements Serializable {
5     final def untyped='untyped'
6 }
7
8 def bean = new MyBean()
9 assert 'untyped' == bean.getUntyped()
10 println bean['untyped']
11 bean['untyped']='tttttttt'; //error
12 bean.setUntyped('-----untyped');//error
13
14 //프로퍼티 변수에 final을 붙이면 읽기전용이 된다
```

프로퍼티 name에 접근하는 방법 비교

자바	그루비
getName()	name
setName('a')	name='a'

#### 19.getter, setter 메서드 만들기만해도 그루비 프로퍼티 접근형식으로 접근가능

```
1 package _19
2
3 class MrBean {
4     String firstname, lastname    //#1
5
6     String getName(){            //#2 가상 프로퍼티의 생성자 가상의name이생긴거다/
7         return "$firstname $lastname"
8     }
9 }
10
11 def bean = new MrBean(firstname: 'Rowan')    //#3
12 bean.lastname = 'Atkinson'                  //#4
13
14 assert 'Rowan Atkinson' == bean.name        //#5 자동으로 생긴name 읽고 쓰고한다.
15 println bean.name ;
16 /*결과
17 Rowan Atkinson
18 */
```

위에서 보는것처럼 프로퍼티접근하는거에대한 일관성이 유지된다. 오우.~

#### 20.프로퍼티 함수를 통하여 접근하지 않고! @기호로 필드 변수에 바로 접근하기.

```
1 package _20
2
3 class DoublerBean {
4     public value                //#1
5
6     void setValue(value){
7         this.value = value      //#2
8     }
9
10    def getValue(){
11        value * 2                //#3
12    }
13 }
14
15 def bean = new DoublerBean(value: 100)
16
17 assert 200 == bean.value        //#4
18 assert 100 == bean.@value       //#5
19
20 println 'bean.value:  : '+bean.value;
21 println 'bean.@value:  : '+bean.@value;
```

```

22  /* 결과
23  bean.value;   : 200
24  bean.@value;  : 100
25  */

```

필드 변수와 같은 영역에서는 `fieldname` 이나 `this.fieldname`을 필드 변수에 대한 접근으로 해석하며 프로퍼티에 대한 접근으로 보지 않습니다. 영역 밖에서는 `referenc.@fieldname`문법을 이용해야 동일한 효과를 얻는다. 주의할 것이 하나 있는데 정적인 영역 (static context)에서 `@`를 사용하거나 `def x = this;x.@fieldname` 같은 식으로 사용하면 이상한 현상이 발생한다. 권장하지 않는다.

## 자바빈 스타일 이벤트처리

그루비는 빈의 내부를 검사(bean introspection) 해서 할당문의 필드 변수가 리스너 등록 메서드 인지 확인한다 등록 메서드로 판단되면 `ClosureListener` 클래스가 자동으로 붙고 이벤트가 발생하면 클로저를 호출해준다.

그루비

```

1  def button = new JButton("push");
2  button.actionPerformed = {event->
3    println button.text;
4  }

```

자바

```

1  final JButton button = new JButton("push");
2  button.addActionListener(new ActionListener() {
3    public void actionPerformed(ActionEvent e) {
4      System.out.println(button.getText());
5    }
6  });

```

## 필드, 접근자, 맵, Expando

그루비 코드에는 `object.name`같은 표현이 자주 눈에 띈다 그루비가 이런 표현식을 만났을 때 하는 일을 나열해 보면 다음과 같다.

1. `object`가 맵이면 `object.name`은 맵의 키에서 `name`을 찾아 대응하는 값을 가르킨다.

```

1  def map = [a:1,b:2];
2  println map.a;

```

2. `name`이 `object`의 프로퍼티일 때는 프로퍼티를 찾아서 가리킨다

```

1  class gogo{
2    def oneProperty='oneProperty_value'
3    def towProperty='towProperty_value'
4    public def getOneProperty() {
5      println 'call getOneProperty'
6      return oneProperty;
7    }
8  }
9  def g = new gogo();
10 println g.oneProperty;
11 /*결과
12 call getOneProperty
13 oneProperty_value
14 */

```

3. 모든 그루비 객체는 자신만의 `getProperty(name)` 과 `setProperty(name,value)` 메서드를 정의할수 있다. 맵(map)객체도 이방법으로 키를 프로퍼티처럼 제공한다.

```

1  class gogo{
2    def oneProperty='oneProperty_value'
3    def towProperty='towProperty_value'
4    public def getOneProperty() {
5      println 'call getOneProperty'
6      return oneProperty;
7    }
8  }
9  def g = new gogo();
10 println g.getProperty('towProperty');
11 /*결과
12 towProperty_value
13 */

```

4. `object.get(name)`메서드를 만들면 필드 변수 접근을 가로챌수 있다 이부분은 그루비 실행환경의 최전방에 속한다 자바빈에 적당한 프로퍼티도 없고 `getProperty`메서드도 구현되지 않았을때 사용된다.  
그리고 당연히 `name`에 특수 문자가 있어서 식별자(identifier)로 사용할수 없을 때도 문자열로 만들면 사용할수 있다 즉 `object.my-name`과같이 만들면된다 또 `GString` 이용할수도있다. `def name = 'my-name';Object.$name` 처럼 써도된다 `Object`에는 `getAt`이라는 메서드가 있어서 `object[name]` 형식으로 프로퍼티에 접근하도록 위임할수 있다.

## 21.프러퍼티 접근 및 get,set메서드 활용 예제

```
1 package _21
2
3 class Go{
4     def g;
5     public void setG(def g) {
6         println 'setG '+g;
7         this.g = g;
8     }
9
10
11     public void set(def name,def value) {
12         println 'No such property Name -> Set Name:'+name+', value:'+value;
13     }
14     public def get(String name) {
15         return 'No such property Name -> '+name;
16     }
17 }
18
19
20 def g = new Go([g:'1']);
21 println 'g.g; : ' + g.g;
22 println 'g.gg; : ' + g.gg;
23
24 println "
25 g.g='g_value'
26 g.gg='gg_value';
27 println "
28
29 println 'g.g; : ' + g.getAt("g");
30 println 'g.gg; : ' + g.getAt("gg");
31
32
33
34 /*결과
35 setG 1
36 g.g; : 1
37 g.gg; : No such property Name -> gg
38
39 setG g_value
40 No such property Name -> Set Name:gg, value:gg_value
41
42 g.g; : g_value
43 g.gg; : No such property Name -> gg
44 */
```

여기서 주의할것은 `get,set` 메서드는 프로퍼티가 없을경우에만 탄다는것이다.  
프로퍼티가 있는것은 자신의 접근 메서드로 바로 호출된다.

## 그루비 특징점 활용하자

그루비의 강력한 세가지 기능을 설명한다

1. GPath
2. 확산연산자(spread operator)
3. use

## 22.GPath는 객체들의 구조를 탐색하는 강력한 도구이다. XPath이름을 따서 왔으나..?

### GPath예제

```
1 package _22
2
3 //현재 객체를 문자열로 표현한것이다
4 println 'this : '+this;
5
6 //이객체의 클래스가 무엇인지 알려면 this.getClass라고하면되지만 그루비에서는 다음처럼 가능하다.
7 println 'this class : '+this.class;
```



```

60 [0]+range;      : [0, 1, 2, 3]
61 [0,range].flatten(); : [0, 1, 2, 3]
62 -----
63 [c:3,*:map];   : [c:3, a:1, b:2]
64 [c:3,*:map];   : [c:3, a:1, b:2]
65 [c:3]+[*:map]; : [c:3, a:1, b:2]
66
67 */

```

여기를 보면 확산연산자의 진짜능력을 알수있다.

```

[*list] : [1, 2, 3, 4]
[]+list : [1, 2, 3, 4]
[list] : [[1, 2, 3, 4]]

```

## 24.키워드 use를 이용한 카테고리 섞기

'숫자형 문자열' + '숫자형 문자열' = 산술계산값

처럼 결과값을 얻고 싶다고 한다면 '1'+1' 하면 안될것이다.

이문제를 해결하기위해 그루비는 use키워드를 제공한다 이키워드는 카테고리를 이용해 클래스에 인스턴스 메서드를 추가할 수 있다.

```

1 use (StringCalculationCategory) {
2   write(read()+read())
3 }

```

여기서 카테고리는 정적 메서드 (카테고리 메서드) 가 포함된 클래스다

키워드 use는 이 메서드의 첫째 인자로 문자열의 인스턴스를 전달해서, 인스턴스 메서드처럼 동작하게 해준다.

```

1 class StringCalculationCategory {
2   static def plus(String self, String operand) {
3     //구현
4   }
5 }

```

```

1 package _24
2
3 class StringCalculationCategory {
4   static def plus(String self, String operand) {
5     println 'plus String'
6     try {
7       return self.toInteger() + operand.toInteger()
8     }
9     catch (NumberFormatException fallback){
10      return (self << operand).toString()
11    }
12  }
13
14  static def plus(def self, def operand) {
15    println 'plus Def'
16    return (self + operand)
17  }
18
19  static def plus(ArrayList self, ArrayList operand) {
20    println 'plus ArrayList'
21    //(self + operand) 이렇게하게되면 재귀호출이된다. 오류남
22    //self.plus(operand) 이렇게하게되면 재귀호출이된다. 오류남
23    int cnt = self.size;
24    for (int i = 0; i < operand.size; i++) {
25      self[cnt++] = operand[i]
26    }
27    return self;
28  }
29
30
31  static def minus(String self, String operand) {
32    println 'minus String'
33    try {
34      return self.toInteger() - operand.toInteger()
35    }
36    catch (NumberFormatException fallback){
37      return (self << operand).toString()
38    }
39  }
40
41 }
42
43 //여기서부터 StringCalculationCategory 클래스의 정의를 따르겠다.
44 use (StringCalculationCategory.class) {
45   assert 1 == '1' + '0'
46   assert 2 == '1' + '1'
47   assert 'x1' == 'x' + '1'
48   assert 0 == '1' - '1'
49   println '-----';
50   println "'1' + '0' : ${('1' + '0')}"
51   println "'1' + '1' : ${('1' + '1')}"
52   println "'x' + '1' : ${('x' + '1')}"
53   println "'1' - '1' : ${('1' - '1')}"

```



```

54     println ([1,2,3,4]+[5,6,7,8]);
55
56 }
57
58 /*결과
59 plus String
60 plus String
61 plus String
62 plus String
63 minus String
64 -----
65 plus String
66 '1' + '0' : 1
67 plus String
68 '1' + '1' : 2
69 plus String
70 'x' + '1' : x1
71 minus String
72 '1' - '1' : 0
73 plus ArrayList
74 [1, 2, 3, 4, 5, 6, 7, 8]
75
76 */

```

**카테고리는 주어진 클로저와 현재 스레드에서만 동작한다** 이런식의 변경이 전역적으로 반영되면 부작용이 발생되기 때문이다.

1. 특수목적의 메서드를 제공한다 StringCalculation Category에서 본 것처럼 연산 메서드의 대상이 동일한 클래스이고 기존 동작을 재정의해야 할 때 사용한다 예제처럼 연산자를 재정의하는 특수한 경우에도 쓸 수 있다.
2. 라이브러리 클래스에 메서드를 추가한다 '불안전 라이브러리 클래스'가 의심될때 사용하면 효과적이다.
3. 함께 동작하는 서로 다른 수신자 클래스를 위한 메서드 모음을 제공한다. 예를 들어 java.io.OutputStream을 위한 encryptedWrite 메서드와 java.io.InputStream을 위한 encryptedWrite 메서드와 java.io.InputStream을 위한 decryptedRead 메서드를 제공할 수 있다.
3. 자바에서 Decorator 패턴을 써야 하는 경우에는 사용한다 하지만 메서드들을 많이 만들어야 하는 불편이 더는 겪지 않아도 된다.
4. 클래스가 너무 커졌을 때 이를 한개의 핵심 클래스와 상황에 맞는 여러개의 카테고리 나눌 수 있다 그리고 use에는 카테고리 클래스를 여러개 줄 수 있으므로 이 카테고리들을 핵심 클래스와 함께 사용한다.

## 25.use 메서드 사용하기

```

1 package _25
2
3 class PCategory{
4     static void gogoSave(Object self){
5         //self 지정하는 함수
6         println 'gogoSave Call '+self.gogo;
7     }
8 }
9 class GG{
10
11     def gogo=[1,3]
12     GG(){
13         use (PCategory){
14             gogoSave();
15         }
16         gogoSave();
17     }
18     public void gogoSave(){
19         println 'GG Class gogoSave()';
20     }
21 }
22
23 }
24
25 def gg = new GG();
26
27 /*결과
28 gogoSave Call [1, 3]
29 GG Class gogoSave()
30 */

```

**카테고리 메서드를 Object에 할당하면 그 메서드는 모든 객체에서 사용할 수 있다.**

Object보다 더 작은 영역에 적용하는 법에도 관심이 있을 것이다 모든 Collection 클래스나 여러분이 작성한 클래스들 중 특정 인터페이스를 공유하는 비즈니스 객체들 전부에 대해 적용할 수도 있다.

use에는 카테고리 클래스 여러개를 줄 수 있다 카테고리들은 쉼표로 분리하거나 리스트로 줘도 된다.

use(a,b,c,d) use([a,b,c,d])..

## 그루비 메타 프로그래밍

그루비에는 가로챌 수 있는 지점이 셀 수 없이 많다

그중 어떤것을 고르는지에 따라 그루비가 내부적으로 제공하는 기능을 이용할 수도 있고 재정의할 코드의 양이 달라지기도 한다.

특징들이 모여서 그루비의 메타오브젝트 프로토콜(MOP,Meta-Object Protocol)을 이룬다

그루비의 모든것은 GroovyObject 인터페이스에서 시작된다.

```
1 package groovy.lang;
2
3 public interface GroovyObject {
4     Object invokeMethod(String name, Object args);
5     Object getProperty(String propertyName);
6     void setProperty(String propertyName, Object newValue);
7     MetaClass getMetaClass();
8     void setMetaClass(MetaClass metaClass);
9 }
```

그루비에서 당신이 작성한 모든 클래스는 GroovyClassGenerator에 의해서 생성된다 덕분에  
그 클래스들은 GroovyObject 인터페이스를 구현하여 각 메서드의 디폴트 구현을 포함하게된다.  
(재정의하지 않았다면 말이다.)

보통의 자바 클래스를 그루비 클래스로 인식시키려면 GroovyObject 인터페이스를 구현하면된다 더 간편하게는 디폴트 메서드를 구현되어있는  
GroovyObjectSupport 추상클래스를 상속해도된다.

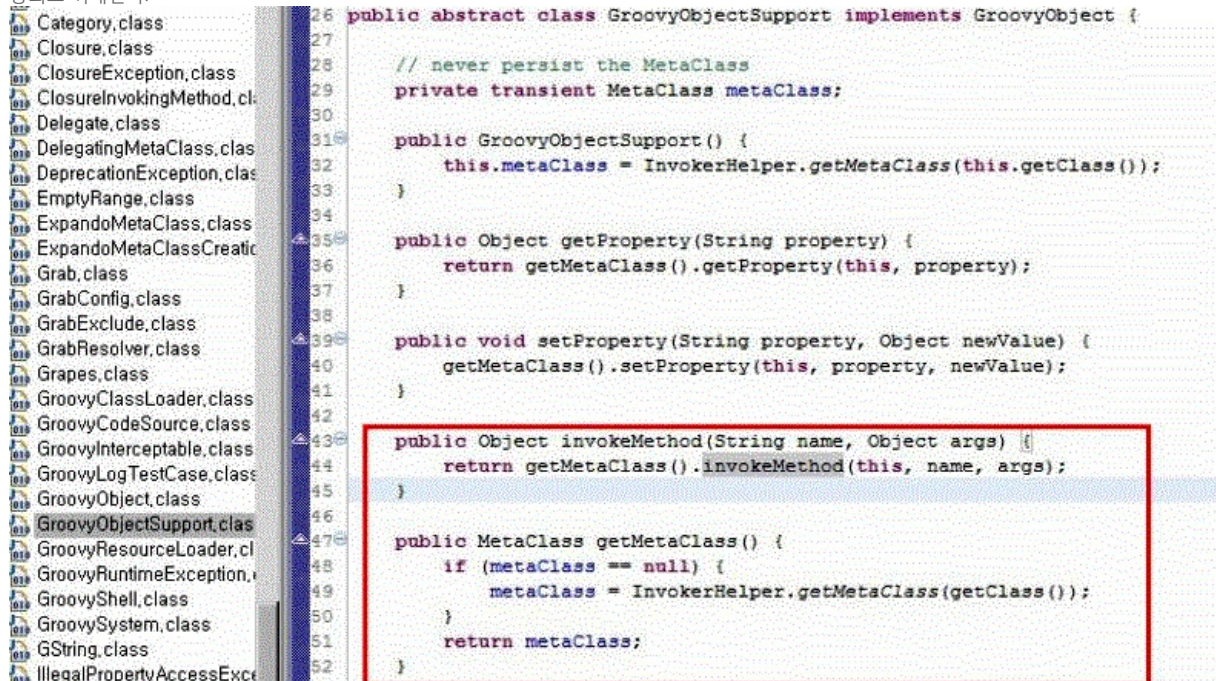
### MetaClass

GroovyObject는 메타 오브젝트 프로토콜의 핵심인 MetaClass와 밀접한 관련이 있다. 이 클래스가 Groovy 클래스에 대한 모든 메타 정보를 제공  
한다 여기서 제공되는 메타정보들은 사용가능한 메서드, 필드, 변수, 프로퍼티등이다. 그리고 다음 메서드들도 구현되어있다.

```
1 Object invokeMethod(Object obj, String methodName, Object args)
2 Object invokeMethod(Object obj, String methodName, Object[] args)
3 Object invokeStaticMethod(Object obj, String methodName, Object args)
4 Object invokeStaticMethod(Object obj, String methodName, Object[] args)
```

위 메서드들이 실제로 메서드 호출을 담당하는 메서드들이다. 이 메서드들은 자바의 리플렉션 API나 자동으로 생성된 리플렉터(reflector) 클래스  
를 통해서 메서드를 호출한다 성능을 생각한다면 기본적으로는 리플렉터 메서드를 이용한다.

Groovy-Object.invokeMethod의 디폴트 구현은 모든 호출을 자신의 MetaClass에게 전달한다. MetaClass는 MetaClassRegistry라는 중앙 저장소에 저  
장되고 꺼내진다.



```
26 public abstract class GroovyObjectSupport implements GroovyObject {
27
28     // never persist the MetaClass
29     private transient MetaClass metaClass;
30
31     public GroovyObjectSupport() {
32         this.metaClass = InvokerHelper.getMetaClass(this.getClass());
33     }
34
35     public Object getProperty(String property) {
36         return getMetaClass().getProperty(this, property);
37     }
38
39     public void setProperty(String property, Object newValue) {
40         getMetaClass().setProperty(this, property, newValue);
41     }
42
43     public Object invokeMethod(String name, Object args) {
44         return getMetaClass().invokeMethod(this, name, args);
45     }
46
47     public MetaClass getMetaClass() {
48         if (metaClass == null) {
49             metaClass = InvokerHelper.getMetaClass(getClass());
50         }
51         return metaClass;
52     }
53 }
```



```

org.codehaus.groovy.runtime
  ArrayUtil.class
  ClassExtender.class
  ConversionHandler.class
  ConvertedClosure.class
  ConvertedMap.class
  CurriedClosure.class
  DateGroovyMethods.class
  DefaultCachedMethodKey.class
  DefaultGroovyMethods.class
  DefaultGroovyMethodsSupport.class
  DefaultGroovyStaticMethods.class
  DefaultMethodKey.class
  EncodingGroovyMethods.class
  FlushingStreamWriter.class
  GeneratedClosure.class
  GroovyCategorySupport.class
  GStringImpl.class
  HandleMetaClass.class
  InvokerHelper.class
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
}
public static MetaClassRegistry getMetaRegistry() {
    return metaRegistry;
}

public static MetaClass getMetaClass(Object object) {
    if (object instanceof GroovyObject)
        return ((GroovyObject) object).getMetaClass();
    else
        return ((MetaClassRegistryImpl) GroovySystem.getMetaRegistry()).getMetaClass(object);
}

public static MetaClass getMetaClass(Class cls) {
    return metaRegistry.getMetaClass(cls);
}

/**
 * Invokes the given method on the object.
 */

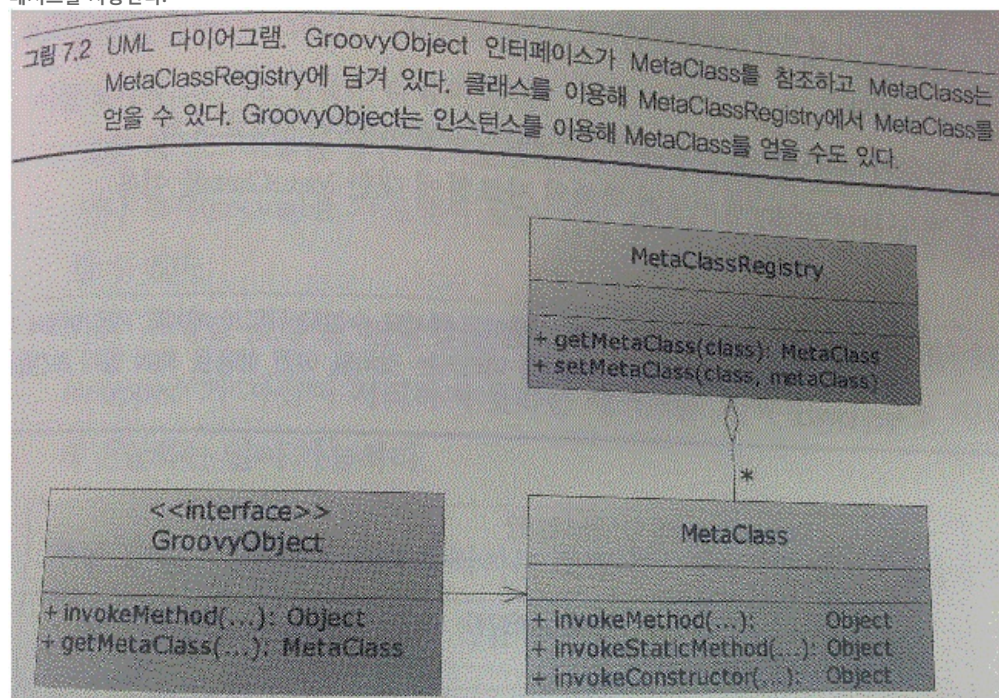
```

```

1 package groovy.lang;
2 //디폴트 메서드를 구현되어있는 GroovyObjectSupport
3 import org.codehaus.groovy.runtime.InvokerHelper;
4 public abstract class GroovyObjectSupport implements GroovyObject {
5
6     // never persist the MetaClass
7     private transient MetaClass metaClass;
8
9     public GroovyObjectSupport() {
10         this.metaClass = InvokerHelper.getMetaClass(this.getClass());
11     }
12
13     public Object getProperty(String property) {
14         return getMetaClass().getProperty(this, property);
15     }
16
17     public void setProperty(String property, Object newValue) {
18         getMetaClass().setProperty(this, property, newValue);
19     }
20
21     public Object invokeMethod(String name, Object args) {
22         return getMetaClass().invokeMethod(this, name, args);
23     }
24
25     public MetaClass getMetaClass() {
26         if (metaClass == null) {
27             metaClass = InvokerHelper.getMetaClass(getClass());
28         }
29         return metaClass;
30     }
31     public void setMetaClass(MetaClass metaClass) {
32         this.metaClass = metaClass;
33     }
34 }

```

MetaClassRegistry는 싱글톤 이어야하지만 아직은 아니다. 어쨌든 실질적으로 한개뿐인 이객체의 인스턴스를 얻을때는 InvokerHelper의 팩토리 메서드를 사용한다.



---

#### 링크 목록

- <http://goo.gl/rRzINX> - <http://goo.gl/rRzINX>
-