01.1주차1-1Groovy기초

그루비에 대한 간단한 소개

Table of Contents

위키피디아 -그루비 (http://ko.wikipedia.org/wiki/%EA%B7%B8%EB%A3%A8%EB%B9%84)

그루비의 기초

일반적인 코드 형태

그루비를 처음 시작할 때는 자바와 구분할 수 없을 정도로 비슷한 형태로 코딩할 수도 있다.
나중에 더 알게 될수록 가볍고 함축적인, 관용어를 활용하는 형태로 바꿔 나가면 된다.
이제부터 코드에 코멘트를 다는법, 자바와 그루비가 다른 점, 비슷한 점, 문법 요소들을 생략해 자바보다 더 짧게 코딩하는 법 등
기초적인 사항들을 알아보자.

첫번째로 그루비는 들여쓰기에 상관없이 동작한다.
두번재로 그루비는 문장 끝의 줄 바꿈과 한 줄 코멘트의 줄 바꿈을 제외하고는 공백 문자에 상관하지 않는다.

코멘트 달기

한 줄 코멘트와 여러 줄 코멘트는 자바와 문법이 똑같다. 한가지 추가된 사항은 스크립트의 맨 첫줄에 사용하는 코멘트이다.

```
1  #! /usr/bin/groovy
2  // 한줄 짜리 코멘트
3  /*
4   여러줄 짜리 코멘트
5   */
```

코멘트를 작성할때 참고할 사항은 다음과 같다.

- 맨 첫줄을 #! 로 시작하면 유닉스 셸에서 그루비를 로딩하고 스크립트를 실행할 수 있다.(이런 코멘트를 shebang 이라고 한다.)
- 한 줄 코멘트는 // 으로 시작하고, 그 줄의 끝까지 코멘트가 된다.
- 여러줄 코멘트는 /* .... */ 으로 감싼다.
- javadoc에서 사용하는 / **** / 스타일의 코멘트도 지원한다.(그루비 1.5.7 이후 버전부터는 javadoc의 문법 대부분을 지원한다.)

그루비와 자바의 문법 비교

그루비 코드중에 '일부분은'(전부가 아니다) 자바와 똑같이 생겼다. 때문에 그루비 문법이 자바 문법을 모두 수용한다고 생각하기쉽다.
문법이 비슷하지만 두 언어는 어느 쪽도 상대방 언어의 부분 집합이 아니다.
미묘한 차이점들을 제외하면 자바 문법에서 상당히 많은 부분이 그루비 문법에 포함되어 있다.
다음 항목들이 이에 속한다.

- 일반적인 패키지 구성방식
- 문장들 (package와 import 포함)
- 클래스와 메서드 정의(중첩 클래스는 제외)
- 제어문(고전적인 for(init; test; inc) 루프는 제외)
- 연산자, 표현식, 할당문
- 예외 처리
- 문자열 정의(약간 변형됨)
- 객체 생성, 참조, 삭제, 메서드 호출

그루비에서 추가된 문법은 다음과 같다.

- 자바 객체에 쉽게 접근하게 해주는 표현식과 연산자
- 객체를 생성하는 다양한 문법
- 실행 흐름을 세련되게 제어하는 새로운 구조
- 새로운 자료형에 관련된 연산자와 표현식
- 모든 것을 객체로 다루기

짧은 것이 아름답다.

자바에서 꼭 필요한 문법 요소도 그루비에서는 생략할 수 있다.
다음은 자바와 그루비에서 URL에 사용하기 위해 문자열을 인코딩하는 코드다. 비교해보자.

```
자바 : java.net.URLEncoder.encode("a b");
그루비: URLEncoder.encode 'a b'
```

그루비 코드는 짧기만 한게 아니라, 코드의 목적을 가장 간단한 형태로 보여준다.
패키지를 나타내는 java.net을 생략하고, 괄호와 세미콜론도 생략해서 코드가 가장 작은 형태로 줄여준다.
그루비 언어 규격(GLS, Groovy Language Specification) 에는 괄호를 생략하면 코드가 모호해지거나, 인수나 호출의 우선순위를
지정하고 싶을 때는 괄호를 사용해도 된다고 나와 있다.
그루비는 자동으로 groovy.lang.*, groovy.util.*, java.lang.*, java.util.*, java.net.*, java.io.* 패키지와 java.math.BigInteger. java.math.BigDecimal 클래스를 임포트 해준다.
이 패키지에 소속된 클래스를 사용할 때에는 패키지 이름을 생략할 수 있다.(참고로 자바에서는 java.lang.* 패키지만 자동으로 임포트해준다.)

assert로 언어를 검증한다.

java 1.4나 더 상위 버전을 사용해 봤다면 "검증(assertion)" 에 대해 잘알고 있을것이다.
이 기능은 프로그램 내의 모든것이 올바르게 동작하는지 검사한다. 테스트 주도 개발에서는 어떤 코드가 수행해야 하는 역활을 최종적으로 증명하려 할 때 이 테스트를 사용하지만, 우리는 코드의 실행 가능 여부만이 아닌 실행 결과를 확인하기 위해서도 검증을 사용했다. 그루비에서 검증이 어떻게 동작하는지, 어떻게 사용해야 하는지 설명하여 추후의 예제들을 이해할 수 있도록 하겠다.
그루비는 assert 키워드를 통해서 검증 기능을 제공한다.

```
assert(true)
assert 1 == 1
def x = 1
assert x == 1
def v = 1; assert v == 1
```

한줄씩 들여다 보자
assert(true)
이것이 assert 키워다. 검증할 표현식을 인자로 주어야함을 알 수 있다.(그루비에서 '참'은 간단한 boolean 값보다 더 넓은 개념이다)
assert 1 == 1
문자나 변수가 아닌 표현식도 assert의 인자로 사용할 수 있다. 루비와는 똑같고 자바와는 다른데 '==' 연산자는 같은 '객체'가 아니라, 같은 '값'을 의미한다.
def x = 1
assert x == 1
변수 x를 정의하고 여기에 정수 1을 대입한 후에, 이를 assert의 표현식에 사용했다.
눈여겨볼 부분은 변수 x의 '자료형'을 지정하지 않았다는 점이다. 키워드 def는 '동적인 자료형'이라는 의미다.
def y = 1; assert y == 1
이는 현재 행의 실행 결과를 검증하는 전형적인 방식이다. 두 문장을 한 줄에 표시하고, 세미콜론으로 둘을 나눈다.

구성 요소들로 나눈 복잡한 다른 예제를 살펴보자.

```
assert('text' * 3 << 'hello').size() == 4 * 3 + 5
```

assert 키워드
('text' * 3 << 'hello').size() 설명중인 주제
== 비교 연산자
4 * 3 + 5 아주 쉬운 부분

그루비 훑어보기

그루비에도 문장이나 표현식 따위로 코드를 분석하는 문법 체계가 있다. 하지만 문법으로 언어를 배우는것은 재미가 없다.
대신 우리는 그루비에서 자주 쓰는 것들(클래스, 스크립트, 자비빈, 문자열, 정규 표현식, 숫자, 리스트, 맵, 범위, 클로저, 반복, 조건문)의 전형적인 사용법을 보여줄것이다.
다만 넓은 영역을 둘러볼뿐 깊게 들어가지 않는다

클래스 만들기

객체지향 프로그래밍에서 클래스는 토대에 해당한다. 객체가 동작하는 방식이 클래스에 설계되어 있기 때문이다.
다음은 간단한 그루비 클래스인 Book을 보여준다. 클래스에는 변수인 title과 title을 저장하는 생성자 그리고 title의 값을 얻는 메소드가 하나 있다.
자바와 너무 비슷하지만, 메서드에 접근 권한 제한자(public, private) 가 없다.

```
class Book {
    private String title

    Book( String theTitle) {
        title = theTitle
    }

    String getTitle() {
        return title
    }
}
```

스크립트로 실행하기

그루비의 스크립트는 확장자가 .groovy인 텍스트 파일이며, 커맨드라인에서 아래와 같은 명령으로 실행 할 수 있다.

```
groovy myfile.groovy
```

이런건 자바로는 할 수 없다. 그루비는 소스코드를 실행시킬수 있다. 내부적으로는 자동으로 클래스 파일이 생성된 후에 실행되지만, 사용자의 눈에는 그냥 그루비 소스코드를 실행시킨 것처럼 보인다.
(약간의 조건을 갖추기만 하면, 모든 그루비 코드는 스크립트로 동작할 수 있다. 약간의 조건이란, 스크립트로 작성되었거나(즉, 클래스에 포함되지 않은 코드가 있거나) 클래스가 있거나 Runnable 클래스이거나, GroovyTestCase 클래스이어야 한다는것이다.)

다음은 Book 클래스를 사용하는 스크립트다. 새로운 객체를 생성하고 자바와 비슷한 문법으로 get 메서드를 호출했다.
그다음에는 title에 저장된 문자열을 거꾸러 읽는 메소드를 정의했다.

```
Book gina = new Book(' Groovy in Action ')

assert gina.getTitle() == ' Groovy in Action '
assert getTitleBackwards(gina) == ' noitcA ni yvoorG '

String getTitleBackwards(book) {
    title = book.getTitle()
    return title.reverse()
}
```

주의해서 볼 부분은 어떻게 getTitleBackwards를 정의하기 전에 이 메서드를 호출했는가 하는 점이다. 여기에는 사실 루비 등 다른 스크립트 언어와 그루비의 근본적인 차이가 숨어 있다.
그루비 스크립트는 실행하기 전에 '분석, 컴파일, 클래스 생성'의 단계를 거친다.
또 하나 중요하게 관찰할 부분은 우리가 Book 클래스를 컴파일한 적이 없다는 점이다. book.groovy 파일 클래스패스에 있기만 하면 된다. 그루비가 알아서 파일을 찾고 컴파일하고, Book 객체를 생성한다. 그루비는 스크립트 언어가 지진 편리함과 객체지향 언어의 유익함을 하나로 묶어 주었다.

이러한 특징을 이용하면 스크립트 기반 프로그램이 거대해질 경우에 구성하는 방법을 이끌어낼 수 있다. 좋은 구성 방법은 바로 수많은 스크립트 파일을 그냥 쌓아 두지 않고 Book 클래스와 같이 재사용 가능한 형태로 모아두는것이다. 이와 같이 클래스로 분리해도 여전히 스크립트의 특징이 있기 때문에 소스코드를 수정 하기만 하면 실행할 때 그 내용이 곧바로 반영될 것이다.

그루비빈

자바빈을 그루비로 구현한것이 그루비빈 이다. 그루비는 자바보다 쉽게 빈을 다룰 수 있도록 다음 세 가지 기능을 제공한다.

- 접근자 자동 생성
- 자바빈과 그루비빈에 대한 간략한 접근법
- 이벤트 핸들러의 간단한 등록

다음 예제는 Book 클래스를 그루비빈으로 정의하기이다.

```
class Book{
    String title  // 프로퍼티 정의
}
def groovyBook = new Book()
```

```
// 명시적 메서드 호출로 프로퍼티 접근
groovyBook.setTitle(' Groovy conquers the world ')
assert groovyBook.getTitle() == ' Groovy conquers the world '

// 그루비의 단축 문법으로 프로퍼티 접근
groovyBook.title = ' Groovy in Action '
assert groovyBook.title == ' Groovy in Action '
```

문자열 다루기

그루비 문자열은 자바와 같이 java.lang.String 클래스를 이용하지만 간편한 사용을위해 몇가지 가능과 연산자를 추가 했다.

GStrings

그루비에서는 문자열을 표시할 때 작은따옴표와 큰따옴표를 모두 쓸 수 있다.
큰 따옴표를 사용해서 문자열을 만들면 문자열 내부의 변수 이름이 그 변수의 값으로 치환된다. 이는 GString이 제공하는 기능이다.

```
def nick = 'Gina'
def book = 'Groovy in Action'
assert "$nick is $book" == 'Gina is Groovy in Action'
```

숫자는 객체다

그루비에서 숫자는 자바와 비슷해 보이지만, 사실은 원시형이 아니라 기본 객체다.
자바에서 원시형은 메서드를 가질수 없다. 만약 x가 int 형이면 x.toString() 같은 메서드 호출은 불가능하다. 반대로 변수 y가 객체라면 2*y와 같은 표현식도 불가능하다.
그루비에서는 숫자로 사칙 연산도 할 수 있고, 메서드도 호출할 수 있다.

```
def x = 1
def y = 2
assert x + y == 3
assert x.plus(y) == 3
assert x instanceof Integer
```

리스트, 맵, 범위 사용하기

리스트

자바에서는 대괄호([]) 로 배열의 요소에 접근하는데, 여기에서는 이 연산자를 배열 기호 연산자라고 부르겠다.
그루비에서는 이 연산자로 리스트(java.util.list)를 다룬다. 즉 배열에 접근하는 것처럼 요소를 추가, 삭제하고, 크기를 변경하고, 자료형이 다른 변수를 저장할 수 있다. 그리고 리스트의 현재 크기를 넘어가는 요소에 접근 할 수도 있는데 그러면 리스트 크기가 자동으로 커진다.

```
def roman = [ '', '1', '2', '3', '4', '5', '6', '7'  ]
assert roman[4] == '4'

roman[8] = '8'
assert roman.size() == 9
```

맵

자바와 달리 그루비는 맵을 언어 수준에서 지원하기 때문에 맵을 위한 문법과 연산자가 있다. 맵과 관련한 문법은 배열과 비슷한데 요소들에는 '키-값'이 저장된다.

```
def http = [
    100 : 'continue',
    200 : 'ok',
    400 : 'bad request'
    ]

assert http[200] == 'ok'
http[500] == 'internal server error'
assert http.size() == 4
```

범위

자바 표준 라이브러리에는 범위(range)의 개념이 없지만,대부분 범위가 무엇인지 직관적으로 알 것이다.

```
def x = 1..10
```

```
assert x.contains(5)
assert x.contains(15) == false
assert x.size() == 10
assert x.from == 1
assert x.to == 10
assert x.reverse() == 10..1
```

코드를 객체로 - 클로저

클로저는 어딘가에 정의된 임의의 코드가 다른 코드에 전달되고 실행되는 것을 말한다.
객체지향 언어에서는 이런 행동을 흉내 낼때 매서드-객체 패턴을 사용하곤 한다.
그루비에서 클로저는 다른 코드 블록처럼 중괄호로 둘러싸인 문장들이다. 인자가 있는 경우에는 먼저 인자들이 나오고 화살표(->) 뒤에 문장들이 나오기도 한다.

```
[1, 2, 3].each { entry -> println entry }
```

[1, 2, 3] 리스트
each 반복 메서드
{ entry -> println entry } 중괄호 내의 클로저
entry 인자
println entry 문장

예제를 보면 리스트[1,2,3] 에 List.each()를 호출하면서 인자로 클로저를 전달했다. List.each 메서드는 클로저를 인자로 받아서 리스트의 각 요소로 주어진 클로저를 실행한다. 실행되는 클로저에는 요소의 값이 인자로 전달된다. 예제에서 사용된 클로저는 entry 라는 이름으로 전달된 인자를 출력한다.

더 자세한 내용은 후에 다시 설명하겠다.

그루비의 제어문들

```
if ( false ) assert false //한줄짜리 if

// if else 문
if ( null ) { assert false }
else { assert true }

// 고전적인 while
def i = 0
while ( i < 10) { i++ }
assert i == 10

// 범위의 for 루프
def clinks = 0
for ( remainingGuests in 0..9 ) { clinks += remainingGuests }
assert clinks == (10*9)/2

// 리스트의 for
def list = [0,1,2,3,4,5,6,7,8,9]
for ( j in list ) { assert j == list[j] }

// 클로저와 each 메서드
list.each() { item -> assert item == list[item] }

// 고전적인 switch
switch (3) {
    case 1 : assert false; break
    case 3 : assert true; break
    default : assert false
}
```

더 자세한 내용은 후에 다시 설명하겠다.

자바 환경에서 그루비의 위치

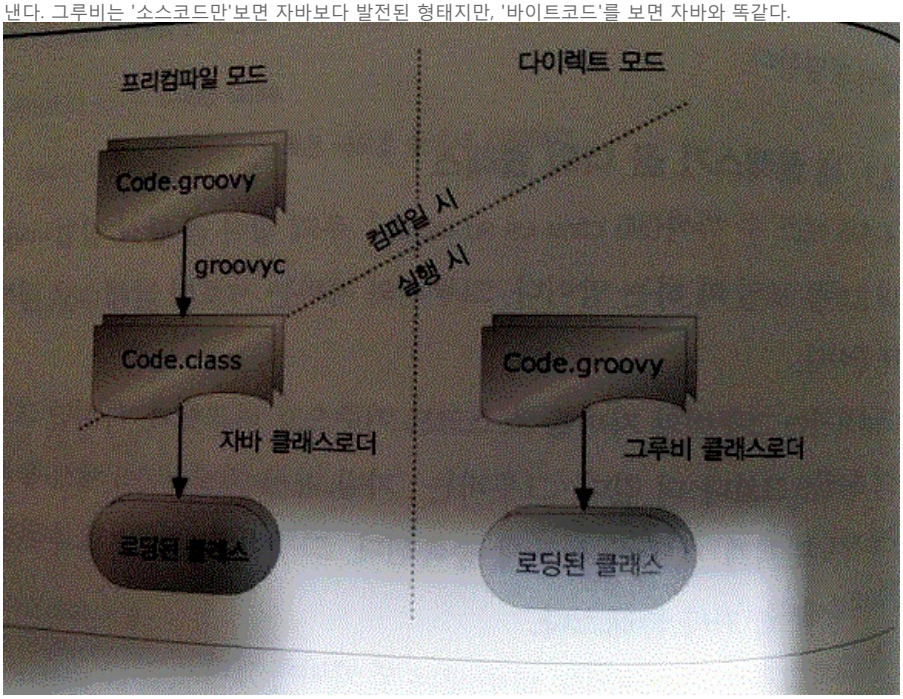
내 클래스가 곧 너의 클래스

그루비는 'JVM(Java Virtual Machine) 에서 동작하기 때문에 자바의 객체 모델에 강하게 종속된다. 그루비의 클래스와 스크립트 모두 jvm 내부에서는 자바 클래스다.
JVM으로 그루비를 실행하는 방법에는 두 가지가 있다.

- groovyc로 *.groovy 파일을 *.class로 컴파일한 후 자바의 classpath에 넣고, 자바의 클래스로더를 이용해서 객체를 생성한다.
- *.groovy 파일을 그래도 사용하고 그루비 클래스로더를 통해 이 파일들로부터 객체를 얻는다. 이렇게 하면 *.class 파일은 생성되지 않는다. 대신 java.lang.Class의 인스턴스인 '클래스 객체;를 얻는다. 달리 말하면 그루비 코드에 new MyClass()라는 문장이 있고 MyClass.groovy 파일이

있다면 그 파일이 파싱되고, MyClass 클래스가 생성된 다음 클래스로더에 로딩된다. 그리고 나서 새로운 MyClass 객체가 할당될 것이다. 이 객체는 마치 *.class 파일에서 생성된 것처럼 쓸 수 있다.

다음 그림은 *.groovy파일을 자바 클래스로 변환하는 방법을 나타냈다. 두가지 방법 모두 자바의 클래스 구조와 동일한 포맷으로 클래스를 만들어낸다. 그루비는 '소스코드만'보면 자바보다 발전된 형태지만, '바이트코드'를 보면 자바와 똑같다.



GDK - 그루비의 라이브러리

그루비가 제공하는 라이브러리는 JDK 라이브러리를 확장한 것이다. 새로운 클래스도 있지만, 기존 JDK 클래스를 확장하는 부분도 있다. 이러한 확장을 GDK라고 부른다.

JDK에서 객체 크기를 얻는 여러 가지 방법

타입	JDK	그루비
Array	length 필드	size()
Array	java.lang.reflect.Array.getLength	size()
String	length()	size()
StringBuffer	length()	size()
Collection	size()	size()
Map	size()	size()
File	length()	size()
Matcher	groupCount()	size()

위와 같이 그루비는 메타클래스라는 장치를 통해서 모든 메서드 호출을 걸러주기 때문에 가능하다.

그루비 개발 주기

그루비 문법은 줄 단위로 정의되지만 그루비 코드가 실행될 때는 그렇지않다. 다른 스크립트 언어들과 달리 그루비 코드는 한 줄씩 실행되지 않는다.
대신 그루비 코드는 전체가 한꺼번에 파싱되고, 파서가 만든 정보에서 클래스가 생성된다. 이렇게 생성된 클래스가 그루비와 자바 사이를 연결해주는 장치가 된다.
자바 실행 환경에서 클래스들은 클래스 로더가 관리하는데 그루비가 생성한 클래스도 자바 클래스와 똑같이 생겼으므로, 글래스로더가 같은 방식으로 관리 할 수 있다. 그루비 클래스로더는 *.groovy 파일 에서도 클래스를 로딩할 수 있다는 차이가 있다.(사실은 그냥 로딩하는것이 아니라 읽어들인 후에, 파싱하고, 클래스를 생성해서, 캐시에 저장한다.)

그루비 클래스 만들기

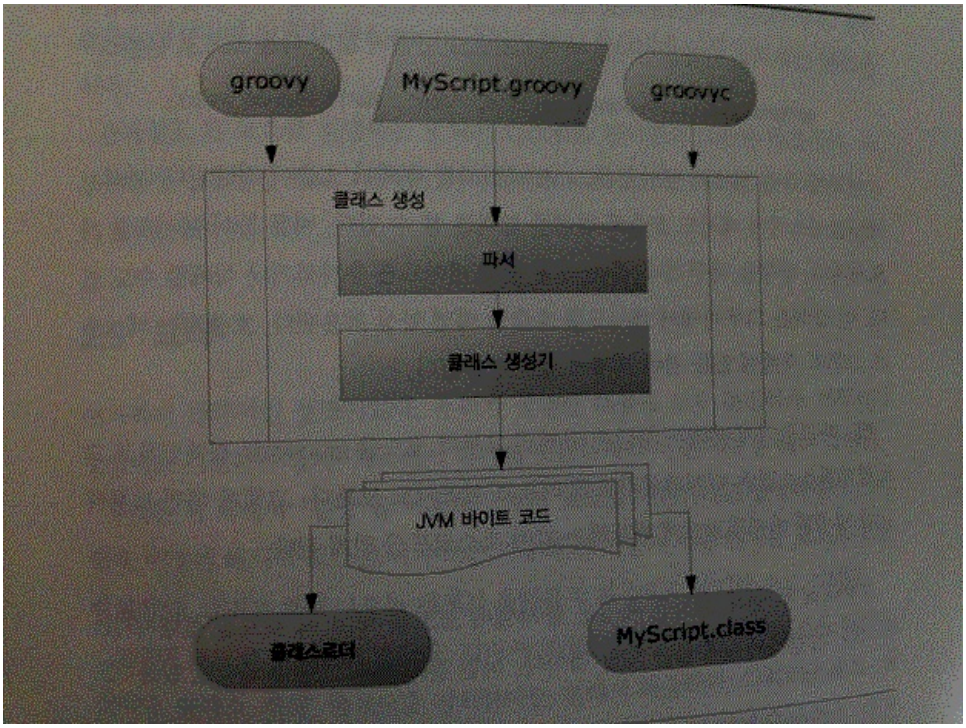
MyScript.groovy라는 파일이 있다고 가정하고 이 파일을 groovy.MyScript.groovy의 명령으로 실행하면 다음과 같이 클래스가 생성된다.

1. 그루비 파서로 MyScript.groovy가 전달된다.
2. 파서는 소스코드에서 추상 구문트리 를 생성한다.
3. 그루비 클래스 생성기가 AST를 받아서 자바 바이트코드를 생성한다. 경우에 따라 클래스를 한 개 이상 만들기도 한다. 만들어진 클래스는 그

루비 클래스로더에 전달된다.

4. 이제 java. Myscript라는 명령을 실행한 것처럼 자바 실행 환경에서 실행된다.

이 모든 과정이 무대 뒤에서 일어나기 때문에 마치 그루비가 인터프리터 인것 처럼 보인다. 하지만 그루비는 인터프리터가 아니다. 클래스는 실행되기 전에 완전한 모습으로 만들어지고, 실행되는 동안에는 내용이 수정되지 않는다.



동적인 그루비

동적인 언어들이 강력한 이유는 실행 중 메서드 추가와 같은 클래스 변경 능력 덕분이다. 하지만 방금 배웠다시피 그루비는 한번 클래스를 만들고 로딩하고 나면 그 바이트코드는 수정하지 않는다. 그런데 클래스에 손대지 않고 어떻게 메서드를 추가할 수 있을까?

그루비의 클래스 생성기는 자바 컴파일러가 만드는것과는 다른(포맷이 아니라 내용면에서 다른)바이트코드를 만든다. 만약 그루비 코드에 'foo'라는 명령문이 있다면, 그루비는 이것을 곧바로 바이트코드로 만들지 않고 객체의 메타클래스를 통해서 호출이 전달된다.
따라서 메타클래스에서 메서드 호출과 관련된 트릭을 쓸 수 있다. 이방식은 그루비에서 메서드를 호출 할때면 항상 적용된다. 호출되는 메서드가 그루비 객체에 있든 자바 객체에 있든 상관없다.

두번째 동적인 특징은 문자열에 코드를 넣고 이코드를 실행시키는 것이다. 이방식은 추후에 그루비통합 에서 배울것이다.

링크 목록

- http://ko.wikipedia.org/wiki/%EA%B7%B8%EB%A3%A8%EB%B9%84 - http://ko.wikipedia.org/wiki/%EA%B7%B8%EB%A3%A8%EB%B9%84
