01.1주차1-2Groovy단순자료형

그루비의 단순 자료형

Table of Contents

객체, 모두 객체

자바의 자료형 - 원시형과 참조형

자바는 원시형 과 참조형 을 구분한다. 원시형만 직접 값을 가질수 있고, 값은 숫자, 문자 아니면 참/거짓이다.
원시형을 제외한 모든것은 참조형이다. 참조형은 객체에 대한 참조만을 할 수 있다. 참조형은 변수에 값을 할당하더라도 이 변수가 이전에 참조하던 객체에는 영향을 미치지 않는다. 우리는 단지 그 변수가 새로운 객체를 참조하도록 하거나, 아무 객체도 가리키지 않도록 만들 수 있다.

원시형 변수에 대해서는 메서드 호출을 할 수도, 객체가 요구되는 곳에 사용할수도 없다. 이런 제한으로 인해 자바는 포장 클래스(wrapper type)를 제공한다. int 형 변수의 포장 클래스는 java.lang.Integer 다.
반대로 참조형에는 3*2 나 a*b 같은 곱셈 연산자 등을 쓸 수없다.

그루비의 해결책 - 모두 객체

그루비에서는 모든것이 객체라서 코드가 짧고 일기 쉬워지게 만들 수 있다.
다음의 예제에서 그루비를 사용시 더 보기 좋은 코드를 만들수 있다.

```
1  // 자바
2  Integer a = 1;
3  Integer b = 2;
4  int sum = a.intValue() + b.intValue();
5
6  // 그루비
7  def a = 1;
8  def b = 2;
9  def sum = a + b;
```

그루비를 완전한 객체지향 언어로 만들기 위해, 그루비 설계자들은 원시형을 사용하지 않기로 결정했다. 가상 기계 수준에서도 원시형에는 메서드 호출을 할 수없기 때문이다. 그루비에서 자바의 원시형에 해당하는 값을 사용하려면 자바 플랫폼이 제공하는 포장 클래스를 사용해야 한다.

표.자바의 원시형과 포장 클래스들 (작성중)

원시형	포장 클래스	설명
byte	java.lang.Byte	8비트 정수
short	java.lang.Short	16비트 정수
int	java.lang.Integer	32비트 정수
long	java.lang.Long	64비트 정수
float	java.lang.Float	단정도(32-bit) 부동 소수점 값
double	java.lang.Double	배정도(64-bit) 부동 소수점 값
char	java.lang.Character	16비트 유니코드
boolean	java.lang.Boolean	Boolean 값(참 또는 거짓)

그루비에서는 5 나 ture 처럼 원시형으로 보이는 표기가 있다 해도 이는 각 자료형에 해당하는 포장 클래스의 참조형이다.
간결함과 익숙함을 위해 원시형 변수인 것처럼 선언 할 수 있게 만들었다.
그러나 실제로는 포장 클래스를 쓰고 있음을 잊지 말자.

자바와 그루비의 숫자 표기법을 비교했을때 둘 사이에는 약간 다른 점이 있는데, 그루비는 java.math.BigDecimal 과 java.math.BigInteger를 지원하는 문법이 있기 때문이다.

다음은 그루비에서 쓸 수 있는 숫자 표기법이다.

표. 그루비의 숫자 표현식 (작성중)

자료형	사용 예
java.lang.Integer	15, 0x1234ffff
java.lang.Long	100L, 200l
java.lang.Float	1.23f, 4.56F
java.lang.Double	1.23d, 4.56D
java.math.BigInteger	1.23g, 456G
java.math.BigDecimal	1.23, 4.56, 1.4E4, 2.8e4, 1.23g, 1.23G

자바와 연동 - 자동포장

자바나 다른 언어에서 원시형을 포장 클래스로 감싸는 동작을 포장(boxing)이라하고, 반대로 클래스에서 원시형 값을 꺼내는 것을 (unboxing)라고 한다. 이러한 작업은 그루비가 처리하는데 이때 그루비에서는 자동으로 처리가 되고 이를 자동포장 이라고 부른다.

그루비가 자바와 잘 연동하도록 설계되어있다고 하는데 자바 메서드의 인수나 리턴 값이 원시형인 경우는 어떻게 그루비가 호출 할수 있을지
String의 메서드 int indexOf(int ch)를 호출해 보자.
그루비로는 이렇게 호출 할 수가 있다.

```
1 assert 'ABCDE'.indexOf(67) == 2
```

자바 메서드는 인자로 int를 요구하지만, 그루비 쪽에서는 67 (문자 C의 유니코드값)을 담고 있는 Integer를 전달하는 상황이다.
이때 그루비가 알아서 객체에서 값을 꺼내어 전달한다. 리턴 값은 int형이지만 그루비 쪽으로 넘어오면 Integer로 포장된다. 그리고
스크립트에 보이는 것처럼 2를 담는 Integer 객체와 비교하게 된다.

쓸데없이 포장을 풀지 않는다.

1+1에서 두 숫자가 모두 Integer 라면 덧셈을 하려고 포장을 풀어야하면 할까?
아니다. 그루비는 무슨 표시든 (숫자, 문자열 등) 그루비 코드에 등장하는 것은 모두 객체다. 자바와 경계를 이루는 곳에서만 포장되거나, 꺼내진다.

자료형의 생략

그루비는 명시적으로 사용했던 자료형을 지정하지 않을경우 이런 변수들을 전부 java.lang.Object 라고 가정해버린다.

자료형 명시하기

그루비에서도 자바처럼 자료형을 명시할수 있다.

표 그루비 예문들과 실행시 자료형(작성중)

문장	자료형	설명
def a = 1	java.lang.Integer	암시적 자료형
def b = 1.0f	java.lang.Float	암시적 자료형
int c = 1	java.lang.Integer	자바 원시형 이름을 이용한 명시적 자료형
float d = 1	java.lang.Float	자바 원시형 이름을 이용한 명시적 자료형
Integer e = 1	java.lang.Integer	참조형으로 이름을 지정하는 명시적 자료형
String f = '1'	java.lang.String	참조형으로 이름을 지정하는 명시적 자료형

그루비는 int 와 Integer 사이에 아무런 차이가 없다. 그루비는 두가지 경우 모두 참조형(Integer)을 쓴다.
간결함을 좋아하고 소스를 읽을 사람이 그루비를 잘 이해하고 있다면 int를 쓰자. 명확함을 좋아하거나 그루비를 잘 모른다면 Integer을 쓰면 된다.
중요한 것은 변수의 자료형을 명시하지 않아도 시스템이 안전하다는 점이다.
다만 그루비는 잘 정의된 변환 규칙없이 어떤 자료형의 객체를 다른 자료형처럼 사용하지는 못하게 한다. 예를 들어 java.lang.String 에 "1" 이 있다고해서 적당한 계산 결과가 나올 것을 기대하면서 java.lang.Number처럼 사용할 수는 없다.
그루비는 자바가 허용하는 정도까지만 허용한다.

정적 자료형 vs 동적 자료형

정적 자료형과 동적 자료형을 선택적으로 사용할 수 있는 능력은 그루비가 지닌 핵심장점 중 하나다.
정적 자료형을 사용하면 최적화를 위한 정보를 더 많이 얻을 수 있고, 컴파일 할 때 더 제대로 된 검사를 할 수 있다.
메서드의 인자에 대해 더많은 정보를 얻고, 메서드를 중복으로 정의할 수도 있다. 정적인 자료형은 리플렉션으로 정보를 가져오는데 전제 조건이 되기도 하다.
동적 자료형도 변수를 전달하기만 할때와 오리형(duck type)을 사용할 때 매우 유용하다.
어떤 메서드를 호출해서 객체를 받아서 그대로 다른 메서드의 인자로 전달하는 경우를 생각해보자.

```
1  // 가상의 소스
2  def node = document.findMyNode()
3  log.info node
4  db.store node
```

이 경우 node의 형이 무엇인지, 어느 패키지 소속인지에는 관심이 없다. 단지 " 저건 그냥 node야 " 라고만 하면된다.

동적 자료형의 두 번째 용도는 어떤 객체의 메서드를 호출하면서 그 객체의 자료형을 정하지 않는 경우다.(이런 자료형을 오리형(duck type)이라고 부르는데 추후에 자세히 설명한다.(그루비의 동적 객체지향 파트 맡으신분 ^^)

연산자 재정의하기(오버라이딩)

재정의 가능한 연산자들

표. 메서드에 기반을 둔 연산자들 (작성중)

연산자	이름	메서드	동작 가능한 자료형
a+b	덧셈	a.plus(b)	숫자, 문자열, 컬렉션
a-b	뺄셈	a.minus(b)	숫자, 문자열, 컬렉션
a*b	곱셈	a.multiply(b)	숫자, 문자열, 컬렉션
a/b	나눗셈	a.div(b)	숫자
a % b	모듈로	a.mod(b)	정수
a++	후행증가	a.next()	숫자, 문자열, 범위
++a	선행증가	a.next()	숫자, 문자열, 범위
a--	후행감소	a.previous()	숫자, 문자열, 범위
--a	선행감속	a.previous()	숫자, 문자열, 범위
a**b	제곱	a.power(b)	숫자
a 하이푼 b	산술 or	a.or(b)	정수
a & b	산술 AND	a.and(b)	정수
a^b	산술 xor	a.xor(b)	정수
~a	비트 보수	a.negate()	정수, 문자열(문자열의 경우는 연산결과는 정규 표현식임)

a[b]	배열 첨자	a.getAt(b)	객체, 리스트, 맵, 문자열, 배열
a[b] = c	배열 첨자 할당	a.putAt(b,c)	객체, 리스트, 맵, StringBuffer, 배열
a<	좌측 시프트	a.leftShift(b)	정수(StringBuffer, Writer, File, Socket, List에서는 append로 동작)
a>>b	우측 시프트	a.rightShift(b)	정수
a>>>b	부호 없는 우측 시프트	a.rightShiftUnsigned(b)	정수
switch(a){ case b: }	분류	b.isCase(a)	객체, 범위, 리스트, 컬렉션, 패턴, 클로저-(컬렉션 c에서 c.grep(b)을 호출하는 경우에는 b.isCase(item)인 모든 c의 item을 리턴한다.)
a==b	비교	a.equals(b)	객체 - hashCode()를 사용한다.
a!=b	비교	!a.equals(b)	객체
a<=>b	비행접시	a.compareTo(b)	java.lang.Comparable
a>b	부등호	a.compareTo(b) > 0	java.lang.Comparable
a>=b	부등호	a.compareTo(b) >= 0	java.lang.Comparable
a	부등호	a.compareTo(b) < 0	java.lang.Comparable
a<=b	부등호	a.compareTo(b) <= 0	java.lang.Comparable
a as type	강제 형변환	a.asType(typeClass)	모든 자료형

연산자 재정의하기

다음 예제에서 머니 클래스의 비교 연산자(==)와 덧셈 연산자(+)를 재정의 했다. 살펴보자

```
//연산자 재정의하기 예제 ( 작성중)
class Money {
    private int amount
    private String currency
    Money (amountValue, currencyValue) {
        amount = amountValue
        currency = currencyValue
    }
    // 1. == 연산자 재정의
    boolean equals ( Object other ) {
        if (null == other )                 return false
        if (! (other instanceof Money ) ) return false
        if (currency != other.currency)     return false
        if (amount != other.amount)          return false
        return true
    }
    int hashCode() {
        amount.hashCode() | currency.hashCode()
    }
    // 2. +연산자 규현
    Money plus ( Money other ) {
        if( null == other ) return null
        if(other.currency != currency) {
            throw new IllegalArgumentException("cannot add $other.currency to $currency")
        }
        return new Money( amount + other.amount, currency )
    }
}

def buck = new Money(1, 'USD')
assert buck
// 3. 재정의된 == 연산자 사용
assert buck == new Money(1, 'USD')
// 4. 재정의된 + 연산자 사용
assert buck+buck == new Money(2. 'USD')
```

예제에서 1주석 내용처럼 간단하게 equals 메서드를 재정의할 수 있다. 값이 같은 머니 객체가 동일한 해시코드를 가지도록 hashCode 메서드도 재정의했다. 해시코드는 java.lang.Object가 요구하는 메서드다.
2주석 내용은 덧셈 연산자를 재정의한것이지만 정확히 말하자면 상위 클래스인 java.lang.Object는 덧셈 연산자를 정의하지 않기 때문에 재정의한 것은 아니다. 이경우에는 '연산자 정의'가 가장 맞는 말일것이다.

재정의(overriding) 와 중복정의(overloading)의 차이를 설명하기 위해 머니 객체의 덧셈 연산자를 중복 정의 해보자.

```
assert buck + 1 == new Money(2. 'USD')
```

위와 같은 코드도 지원하려면 다음과 같은 메서드가 필요하다.

```
Money plus (Integer more) {
    return new Money(amount + more, currency )
}
```

강제형변환 활용하기

자료형이 같은 변수 일때 연산자를 만드는 일은 간단하다. 그러나 자료형이 여러개 등장하면 복잡해진다.
1 + 1.0
이 코드는 Integer 과 BigDecimal 을 더한다. 리턴 값은 무슨 형인가? 이때 둘 중 하나를 더 일반적인 자료형으로 확장해야한다.
이런것을 "강제 형변환"이라 한다.

지원할 자료형

어떤 자료형과 값을 인자로 받을지를 먼저 정하고, 인자가 잘못되었을 때는 IllegalArgumentException을 발생시킨다.

한정적인 자료형을 확장하기

인자의 자료형이 구현 중인 자료형보다 제한적이라면, 더 일반적인 쪽으로 끌어올려야 하고 리턴 값도 마찬가지여야 한다.
만약 BigDecimal클래스를 설계하는 중에 Integer를 인자로 받는 덧셈 연산자를 구현하기로 했다면,
Integer는 BigDecimal보다 한정된 자료형이다. 모든 Integer의 값은 BigDecimal로 표현 가능하지만 반대로는 불가능하다. 따라서 Integer를 BigDecimal로 확장해서 계산한 후에, 새로운 BigDecimal을 만들어 리턴하는것을 고려해봐야 한다. 결과 값이 Integer로 정확하게 표현할 수 있다.

(이건 예제가 필요할듯. 신규로 만드는것 고려중)

더 일반적인 인자로 다시 전달

만약 인자쪽의 자료형이 더 일반적이면 인자의 메서드를 호출하면서 현재 객체, 즉 this를 인자로 준다. 그러면 인자였던 객체가 this를 확장해준다.
이런 상황을 더블 디스패치 라고 부른다.
예를 들어 Integer.plus(BigDecimal operand)일때 BigDecimal의 plus(Integer)메서드에 위임해서 operand.plus(this)를 호출하면 결과값은 BigDecimal 일것이다.(교환법칙이 성립하는 연산자에만 적용할 수 있다.)
(교환법칙이란 연산 순서를 바꿔도 그 결과가 바뀌지 않는다는 것이다. 덧셈은 성립하지만 뺄셈은 성립하지 않는다)

그루비의 관례

강제형변환을 할때 "가장 일반적인 자료형"을 돌려 준다는 것이 그루비의 일반적인 전략이다.

문자열로 작업하기

그루비의 문자열은 일반적인 문자열과 GString, 두가지가 있다. 일반적인 문자열은 java.lang.String 객체이고 GString은 groovy.lang.GString 객체이다.

다양한 문자열 표시

표. 그루비 문자열 표기법의 정리(작성중)

감싸는 문자들	사용 예	GString 가능 여부	역슬래시 치환문자
작은따옴표	'hello Dierk'	불가	가능
큰따옴표	"hello $name"	사용가능	사용 가능
삼중 작은따옴표(''')	'''---- total: $0.02 ----'''	불가	사용 가능
삼중 큰따옴표(""")	"""first line second line third line"""	가능	사용 가능
슬래시	/x(d*)y/	가능	상황에 따라

슬래스 - 이 자료형은 치환을 하지 않는 것이 핵심이다. 따라서 가능한 한 치환 문자를 지원하지 않지만, 유니코드 때문에 사용하는 ₩u 의 경우와 문자열 끝을 말하는 $가 아닌, 문자 $를 의미하는 ₩$만은 예외로 한다.
각 표기법에는 다른 표기와 구분되는 특징이 하나씩 있다.

- 작은따옴표는 내용에 뭐기 들어가든 절대로 GString이 되지 않는다. 자바의 문자열 표기법과 거의 똑같다.
- 큰따옴표는 작은따옴표와 같지만, 치환문자(escape)가 아닌 달러표시($)가 들어가면 GString이 된다. GString에 대한 더 자세한 내용은 다음에 다룬다.
- 삼중따옴표는 여러 줄에 걸치는 문자열을 만들어 준다. 모든 줄은 플랫폼에 상관없이 항상 ₩n으로 치환되고, 다른 공백문자들은 텍스트 파일에서 읽은 것처럼 그대로 보존된다.
- 슬래시 문자열 표기법은 역슬래시를 치환 문자로 다루지 않기 위한 것이다. 이것은 특히 정규표현식을 사용할때 유용하다.

표. 그루비에서 인식하는 치환 문자들 (작정중)

치환 문자	의미
₩b	백스페이스
₩t	탭
₩r	캐리지 리턴

₩n	새로운 줄, 개행
₩f	새로운 페이지, 폼 피드
₩	역슬래시
₩$	달러 기호
₩uabcd	유니코드 문자. 여기서 a,b,c,d는 16진수
₩abc	유니코드 문자. 여기서 a,b,c,는 8진수 이며, b,c는 생략 가능
₩'	작은따옴표
₩"	큰 따옴표

₩abc - 8진수를 이용한 치환은 잘못 쓰기 쉬우므로 가급적 사용하지 말아야한다. 호환성 때문에 지원하는것이다.

GString 사용하기

GString에 달러 기호와 변수 이름을 적으면 그 값을 가져올 수 있다.

- 1. 가장 간단한 사용법이다.
- 2. 도트 연산자를 이용해서 프로퍼티를 가져오는 법도 있다.
- 3. 달러 기호와 중괄호를 사용하는 완전한 문법이다. 중괄호 안에는 어떤 그루비 표현식이든 원하는대로 쓸 수 있다.(클로저)
- 4. SQL 쿼리를 만들어내는데 GString을 사용했다.
- 5. GString 내부에서 달러 기호를 표시하고 싶을때 역슬래시로 치환해주면 된다.

```
//GString 사용법 ( 작성중)

// 1. 달러기호를 이용하는 단축 문법
me = 'Tarzan'
you = 'Jane'
line = "me $me - you $you"
assert line == 'me Tarzan - you Jane'
// 2.확장된 단축 문법
date = new Date(0)
out = "Year $date.year Month $date.month Day $date.date"
assert out == 'Year 70 Month 0 Day 1'
// 3. 중괄호를 사용하는 완전한 문법
out = "Date is ${date.toGMTString()} !"
assert out == 'Date is 1 Jan 1970 00:00:00 GMT !'
// 4. 여러 줄짜리 GString
sql = """
SELECT FROM MyTable
WHERE Year = $date.year
"""
assert sql == """
SELECT FROM MyTable
WHERE Year = 70
"""
// 5. 달러기호를 문자열에 쓰려는 경우
out = "my 0.02₩$"
assert out == 'my 0.02$'
```

GString은 프로그래머가 사용하는 모든 연산에 대해 java.lang.String 처럼 동작하지만, 문자열 내부의 변하지 않는 부분과 변하는 부분(치환되는 값들을 말한다)을 다루기 위해 서로 다르게 구현되어 있다. 이는 다음 코드를 보면 알 수 있다.

```
me = 'Tarzan'
you = 'Jane'
line = "me $me - you $you"
assert line == 'me Tarzan - you Jane'
assert line instanceof GString
assert line.strings[0] == 'me '
assert line.strings[1] == ' - you '
assert line.values[0] == 'Tarzan'
assert line.values[1] == 'Jane'
```

자바에서 그루비로

```
//문자열로 하는 일들 ( 작성중)
greeting = 'Hello Groovy!'

assert greeting.startsWith('Hello')

assert greeting.getAt(0) == 'H'
assert greeting[0] == 'H'

assert greeting[6..11] == 'Groovy'

assert 'Hi' + greeting - 'Hello' == 'Hi Groovy!'

assert greeting.count('o') == 3
```

```
assert 'x'.padLeft(3) == '  x'
assert 'x'.padRight(3,'_') == 'x__'
assert 'x'.center(3) == ' x '
assert 'x' * 3 == 'xxx'
```

그루비는 문자열의 내용을 그자리에서 바꾸는 기능을 지원하지 않는다. 이유는 java.lang.String을 사용하고, 문자열 '내용은'은 '변하지 않는다'는 자바의 규칙을 따르기 때문이다.
그래서 그루비에서는 StringBuffer를 써서 해당 기능을 지원한다. 상단의 예제에서 StringBuffer를 적용하면 아래의 코드가 된다.

```
//StringBuffer 를 적용한 예제 ( 작성중)
greeting = 'Hello'
// 1. 시프트 연산과 동시에 할당하기
greeting <<= ' Groovy'
assert greeting instanceof java.lang.StringBuffer
// 2. StringBuffer에 시프트 연산
greeting << '!'
assert greeting.toString() == 'Hello Groovy!'
greeting[1..4] = 'i' // 내부의 ello가 i가 되었다.
assert greeting.toString() == 'Hi Groovy!'
```

stringA << stringB표현식은 STringBuffer를 리턴한다. 하지만 여기에 사용된 stringA에 자동으로 할당되지는 않는다.

- 1. String에 << 을 사용하는 경우에는 명시적으로 할당 해주어 야한다.
- 2. StringBuffer에 << 을 사용하려면 연산자 좌측의 객체를 변경해야한다. String에 대해서는 기존 데이터를 변경할 수 없기 때문에 새로운 객체를 리턴하는 것이다.

정규 표현식 사용하기

그루비는 자바의 정규 표현식에 연산자를 세개 더 추가했다.

- 정규 표현식 검색 연산자 =~
- 정규 표현식 일치 연산자 ==~
- 정규 표현식 패턴 연산자 ~String

표. 간단한 정규 표현식 패턴들(작성중)

패턴	의미
some text	"some text"라는 문자열
some₩s+text	문자열 "some"뒤에 공백 문자가 하나 혹은 그 이상 있고, 그 뒤에 "text"가 있다.
^₩d+(₩.₩d+)?(.*)	장이나 절 제목에 해당한다. ^는 줄의 시작을, ₩d는 숫자를, ₩d+는 하나 혹은 그 이상의 숫자를 나타낸다. 괄호는 그룹을 만들때 사용한다. 물음표는 첫 번째 그룹이 없을 수도 있다는 것을 뜻한다. 두 번째 그룹에는 제목이 들어간다. 도트는 모든 문자를 나타내고 별표는 이런 문자가 몇 번이든 반복된다는 것을 나타낸다.
₩d₩d/₩d₩d/₩d₩d₩d₩d	형식을 갖춘 날짜. 숫자 두 개 뒤에 슬래시가 있고, 그 뒤에 숫자 두개, 슬래시 그리고 숫자 네 개가 나온다.

패턴의 문자열 표기법

```
//GString에서 패턴 정의하기 ( 작정중)
assert "abc" == /abc/
assert "₩d" == /d/

def reference = "hello"
assert reference == /$reference/

assert "₩$" == /$/
```

표. 정규 표현식 기호 - 일부 (작성중)

기호	의미
.	모든 문자
^	줄의 처음(싱글라인 모드에서는 문서의 처음)
$	줄의 끝(싱글라인 모드에서는 문서의 끝)
₩d	숫자
₩D	숫자를 제외한 문자
₩s	공백 문자
₩S	공백 문자를 제외한 문자
₩w	단어
₩W	단어가 아닌 문자

₩b	단어 경계
()	그룹
(x하이푼y)	x 또는 y, 예를 들면(그루비 하이푼 자바 하이푼 루비)
₩1	1번 그룹의 백매치, 예를 들면 두 번 나오는 글자를 찾으려면(.)₩1 이라고 한다.
x*	0번 이상 반복되는 x
x+	1번 이상 반복되는 x
x?	0번 또는 1번 출현하는 x
x{m,n}	m번 이상 n번 이하 반복되는 x
x{m}	정확히 m번 반복되는 x
[a-f]	a,b,c,d,e,f 문자들
[^a]	a가 아닌 문자들
(?is:x)	x를 검사할 때 모드를 바꿀 것, i는 대소문자 구분을(ignoreCase), s는 싱글라인 모드(single-line)를 의미한다.

패턴 사용하기

그루비는 정규 표현식을 문자열에 대해 이런 일을 할 수 있다.

- 문자열 전체가 패턴과 일치하는지 검사
- 문자열에 패턴과 일치하는 부분이 있는지 검사
- 패턴과 일치하는 부분이 몇 번이나 나오는지 세어보기
- 패턴과 일치하는 부분을 모두 다른 문자열로 바꾸기
- 패턴과 일치하는 부분들을 기준을 문자열을 조각내기

```
//정규 표현식 예제 ( 작성중)
twister = 'she sells sea shells at the sea shore of seychelles'

//twister에는 s로 시작하고 a로 끝나는 단어가 세 개 있다.
assert twister =~ /s.a/ // 1. if문에서 쓰기 좋은 검색 연산자

// 2. 검색 연산자는 Matcher 객체를 돌려준다.
finder = (twister =~ /s.a/)
assert finder instanceof java.util.regex.Matcher

//twister의 단어들 사이에는 공백이 한 칸씩만 있다.
assert twister ==~ /w+( w+)*/ // 일치 연산

// 일치 연산은 Boolean 객체를 돌려준다.
WORD = /w+/
matches = (twister ==~ /($WORD $WORD)*/)
assert matches instanceof java.lang.Boolean

//일치 연산자는 부분이 아닌 문자열 전체와 비교한다.
assert (twister ==~ /s.e/) == false

wordsByX = twister.replaceAll(WORD, 'x')
assert wordsByX == 'x x x x x x x x x x'

//단어들을 얻는 split 메서드
words = twister.split(/ /)
assert words.size() == 10
assert words[0] == 'she'
```

실전 패턴

```
//패턴의 일치영역들과 작업하기 ( 작성중)
myFairStringy = 'The rain in Spain stays mainly in the plain!'

// 'ain:' 으로 끝나는 단어를 :bW*ainb
BOUNDS = /b/
rhyme = /$BOUNDSw*ain$BOUNDS/
found = ''
// 1. string.eachMatch(패턴 문자열)
myFairStringy.eachMatch(rhyme) {
    match ->  found += match[0] + ' '
}
assert found == 'r S p '

//2. matcher.each(클로저)
found = ''
(myFairStringy =~ rhyme).each{
    match -> found += match + ' '
}
assert found == 'rain Spain plain '

// 3. string.replaceAll(패턴 문자열, 클로저)
cloze = myFairStringy.replaceAll(rhyme){
    it-'ain'+'    '
```

```
}
assert cloze == 'The r    in Sp    stays mainly in the pl    !'
```

문자열에 eachMatch라는 메서드가 있는데, 이 메서드는 패턴에 일치하는 각 부분에서 수행할 클로저를 인자로 받는다.
정규표현식 검사 결과에 대해, 두가지 방법으로 반복문을 만들 수 있는데 결과는 똑같다.
String.eachMatch(Pattern)이나 Matcher.each()을 쓰는것이다. 그루비의 정규표현식이 동작하는 방식을 완전히 이해하려면 JDK의 java.util.regex.Matcher 클래스를 보아야만 한다.

GDK는 Matcher 클래스가 제공하는 이 내부 정보를 배열처럼 쓸 수 있게 확장했다.
공백 문자 이외 문자를 검색하는 다음 예제를 보자

```
matcher = 'a b c' =~ /S/
assert matcher[0] == 'a'
assert matcher[1..2] == ['b','c']
assert matcher.count == 3
```

패턴과 검색 속도

패턴 연산자는 문자열을 패턴 객체(java.util.regex.Pattern)로 만든다. 문자열에 이 패턴 객체를 적용하면 matcher 객체를 얻는다.
여기에는 패턴이 내부적으로 "유한 상태 기계(finite state machine)"라 불리는 구조로 만들어졌고, 덕분에 높은 성능을 낼 수 있다는 이론적인 배경이 있따. 복잡한 패턴일수록 생성하는데 오랜 시간이 걸리지만, 패턴 검색 과정은 굉장히 빠르게 수행된다.

```
//패턴 재사용으로 실행 속도 최적화 하기( 작성중)
twister = 'she sells sea shells at the sea shore of seychelles'
// 약간 복잡한 정규 표현식:
// 처음과 끝 글자가 같은 단어들
regex = /b(w)w*1b/

start = System.currentTimeMillis()
// 암시적 패턴 생성을 이용한 검색 연산
100000.times{
    twister =~ regex
}
first = System.currentTimeMillis() - start

start = System.currentTimeMillis()
// 패턴의 명시적 생성( a =~ b 가 아니라 a = ~b 이다. 조심하자)
pattern = ~regex
100000.times{
    pattern.matcher(twister) // 패턴을 문자열에 적용
}
second = System.currentTimeMillis() - start

assert first > second * 1.20
```

숫자 사용하기

그루비의 정수형의 기본 자료형은 java.match.BigDecimal이다. 이는 부동소수점 연산과 관련해 흔히 저지르는 실수를 방지한다.

사칙연산과 강제형변환

덧셈,곱셈,뺄셈은 자바와 거의 동일하게 동작한다. 그러나 부동소수점, BigInteger, BigDecimal은 약간 다르게 동작한다.
다음은 덧셈, 곱셈, 뺄셈에 대해 적용되는 규칙들이다.

- 피연산자 중 Float나 Double이 있으면 결과는 Double이다.(자바에서는 피연산자 모두 Float일 경우에만 결과도 Float다)
- 그렇지 않고, 피연산자 중에 BigDecimal가 있으면, 결과는 BigDecimal이다.
- 그렇지 않고, 피연산자 중에 BigInteger가 있으면 결과는 BigInteger다.
- 그렇지 않고, 피연산자 중에 Long이 있으면 결과는 Long이다.
- 그렇지 않으면 결과는 Integer다.

표. 숫자 강제형변환 (작성중)

+-*	B	S	I	C	L	BI	BD	F	D
Byte	I	I	I	I	L	BI	BD	D	D
Short	I	I	I	I	L	BI	BD	D	D
Integer	I	I	I	I	L	BI	BD	D	D
Character	I	I	I	I	L	BI	BD	D	D
Long	L	L	L	L	L	BI	BD	D	D
BigInteger	BI	BI	BI	BI	BI	BI	BD	D	D

BigDecimal	BD	BD	BD	BD	BD	BI	BD	D	D
Float	D	D	D	D	D	D	D	D	D
Double	D	D	D	D	D	D	D	D	D

강제 형변환에서 살펴볼 사항들이 있다.

- 자바와 비슷하지만 루비와는 다른 특징으로, 연산 결과를 현재 자료형으로 표현할 수 없는 경우에도 강제형변환을 하지 않는다.(제곱 연산은 예외)
- 나눗셈을 할 때 피연산자 중Float나 Double이 있다면, 결과는 Double 이다. 그렇지 않다면 결과는 BigDecimal로 피연산자 중 더 높은 쪽의 정밀도로 반올림한다. 그리고 결과 값의 뒤쪽은 '0'이 없도록 처리된다.
- 정수의 나눗셈에서 정수 결과 값이 필요할 때는 캐스팅을 하거나 혹은 intdiv()을 사용한다.
- 시프트 연산자들은 Integer와 Long 자료형에 대해서만 정의되어 있다. 이 연산자들은 강제형변환을 하지 않는다.
- 제곱 연산자들은 결과 값의 범위나 정밀도를 처리할 수 있는 최적의 자료형을 찾아서 강제형변환을 한다. 찾는 순서는 Integer, Long, Double 이다.
- 비교 연산자들은 비교하기 전에 더 일반적인 자료형으로 강제형변환을 수행한다.

표. 숫자 표현 예제들 (작성중)
자바에서 결과는 정수 0이 될 것이다.

표현식	결과 자료형	비고
1f*2f	Double	자바에서는 Float가 된다.
(Byte)1+(Byte)2	Integer	자바처럼 Integer다. 정수 연산은 언제나 최소한 32비트로 이루어진다.
1*2L	Long	
1/2	BigDecimal(0.5)	
(int) 1/2	Integer(0)	BigDecimal을 Integer로 강제형변환 한다.
1.intdiv(2)	Integer(0)	자바의 1/2과 동일하다.
Integer.MAX_VALUE + 1	Integer	제곱 연산이 아닌 경우에는 결과 값을 확장하지 않는다
2**31**	Integer	제곱연산은 필요할 때 결과 값의 자료형을 확장한다.
233	Long	제곱연산은 필요할 때 결과 값의 자료형을 확장한다.
2**3.5	Double	제곱연산은 필요할 때 결과 값의 자료형을 확장한다.
2G+1G	BigInteger	
2.5G + 1G	BigDecimal	
1.5G == 1.5F	Boolean(true)	비교하기 전에 Float가 BigDecimal 형으로 확장된다.
1.1G == 1.1F	Boolean(false)	Float는 (사실은 Double도) 숫자 1.1을 정확하게 표현할수 없다.

숫자를 위한 GDK 메서드들

GDK에는 숫자들을 위해 plus, minus.power 등의 연산자가 구현되어 있으며, 이 메서드들을 재정의 할 수도 있다. 추가로 abs, toInteger,round 등의 메서드도 이름이 의미하는대로 동작한다.
GDK가 정의해 놓은, 더 흥미로운 메서드로는 times, upto, downto, step 이 있다. 모두 클로저를 인자로 받는다. 반복을 위해서는 times를 증가하는 숫자를 따라 작업할 때는 upto를, 감소할 때는 downto를 사용한다.

```
1  //숫자에 대한 GDK 메서드 ( 작성중)
2  def store = ''
3  10.times{
4      store += 'x'
5  }
6  assert store == 'xxxxxxxxxx'
7  // 반복분 내의 증가 변수
8  store = ''
9  1.upto(5) {
10     number -> store += number
11 }
12 assert store == '12345'
13
14 //반복분 내의 감소 변스
15 store = ''
16 2.downto(-2) {
17     number -> store += number + ' '
18 }
19 assert store == '2 1 0 -1 -2 '
20
21 // 지정된 간격으로 증감
22 store = ''
23 0.step(0.5, 0.1) {
24     number -> store += number + ' '
25 }
26 assert store == '0 0.1 0.2 0.3 0.4 '
```

자바 프로그래머라면 숫자에 메서드를 호출하는 광경이 익숙하지 않을 것이다. 그러나 그루비에서는 숫자도 객체이므로 이렇게 써도 된다는 점을 기억해두자.