Scala
Block expression
이해하기

myjlms99@gmail.com

Block expression

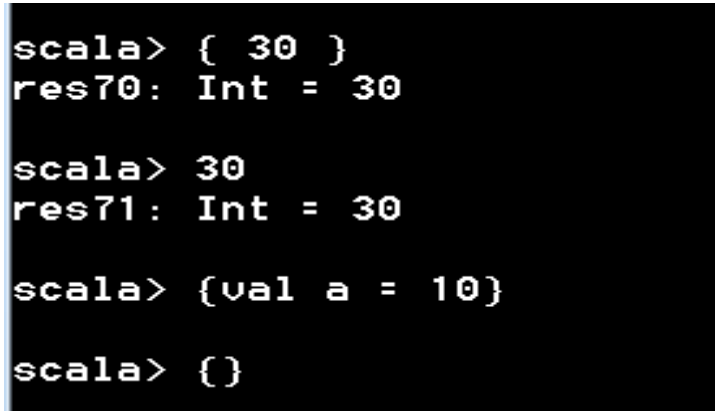
블록 표현식

블록 표현식은 문장과 표현식을 사용할 수 있으면 바로 평가가 되면 그 결과를 마지막 표현식을 보고 반환한다.

```
{
    문장과 표현식
    표현식 // 마지막 리턴값으로 인식
}
```

블록 표현식

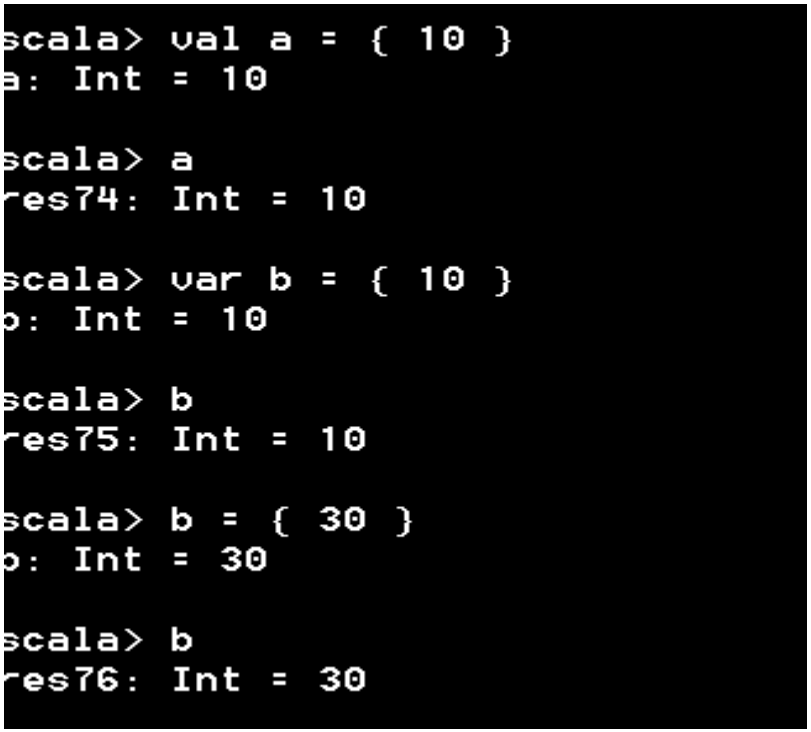
블록 표현식에 리터럴과 변수 할당 등을 하고 실행하면 마지막 평가된 결과를 가지고 처리한다



변수에 할당

val, var 변수에 할당

표현식이 평가되어 바로 변수에 할당되므로 변수에 값으로 사용할 수 있다



lazy val에 할당

lazy 키워드를 이용하면 지연평가이므로 실제 이 변수가 호출될 때 한번 실행되어 처리된다.

```
scala> lazy val c = { 30 }
c: Int = <lazy>

scala> c
res77: Int = 30
```

Def 에 할당

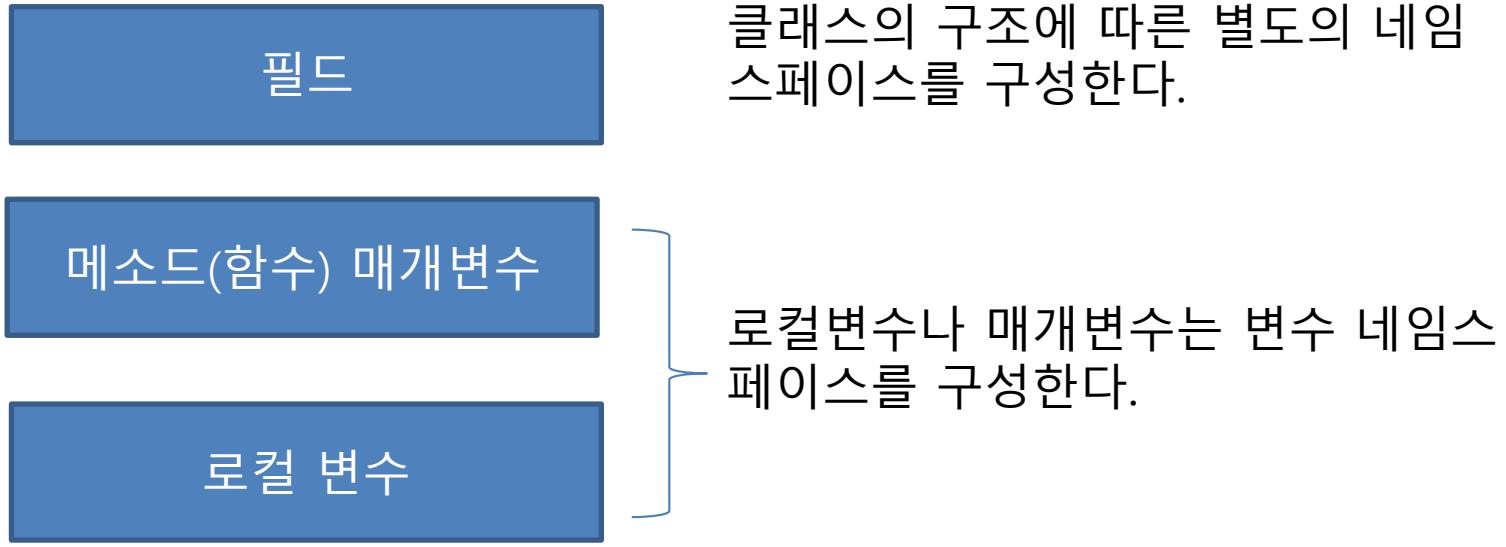
def 변수명에 할당하면 이름으로 호출되면 매번 다시 실행된다. 함수 정의이지만 실제 매개변수가 없을 경우는 변수처럼 사용되어 처리되는 것과 유사하게 사용된다.

```
scala> def d = { var a = 10; println(a); a=a+10; a }
d: Int

scala> d
10
res79: Int = 20

scala> d
10
res80: Int = 20
```

Scope

사용자 정의 변수 관리기준

스칼라 변수는 선언 된 위치에 따라 세 가지 범위로 분류됩니다. 필드, 메소드 매개 변수 및 로컬 변수입니다.



블록 정의

블록으로 정의된 것은 def(함수,메소드), class, trait 선언에 따라 내부에 정의된 변수가 필드가 되거나 로컬변수로 인식된다.

클래스나 트레이트에 블록 정의를 하면 변수는 필드로 인식된다

함수나 메소드에 블록의 정의되면 내부에 정의된 변수는 로컬변수로 인식된다.

필드 field

클래스, 객체, trait 등에 선언된 변수를 말하면 액세스 수정 자 유형에 따라 객체의 모든 메소드와 객체 외부에서 액세스 할 수 있고, var 및 val 키워드에 따라 변경 가능하거나 변경 불가능할 수 있다.

클래스에 정의된 필드는 기본 public이다

```
scala> class I {
     |    val x = 100
     | }
defined class I

scala> val i = new I
i: I = I@2ddb44da

scala> i.x
res59: Int = 100
```

메소드 매개변수 : 메소드

메소드가 호출 될 때마다 메소드 내부의 값을 전달하는 데 사용되는 변수

```
scala> class Sub {
     |     def minus(s1:Int, s2:Int) = {
     |         s1 - s2
     |     }
     | }
defined class Sub

scala> val sm = new Sub
sm: Sub = Sub@776522a0

scala> sm.minus(100,100)
res46: Int = 0
```

메소드 매개변수 : 클래스

일반 클래스를 정의할 때 사용되는 매개변수는 함수에서 바로 접근해서 사용할 수 있다.

스칼라 클래스도 매개변수로 정의가 가능하고 이를 내부 메소드에서 바로 접근 가능

```
scala> class Add(s1:Int, s2:Int) {
     |     def plus = s1 + s2
     | }
defined class Add

scala> val a = new Add(10,10)
a: Add = Add@23eba65c

scala> a.plus
res49: Int = 20
```

로컬 매개변수

지역 변수는 함수, 메소드 내부에서 선언된 변수입니다. 메소드 내에서만 접근 할 수 있다. var 및 val 키워드를 사용

함수, 메소드 내에 정의된 변수는 외부에서 참조가 되지 않는다.

```
scala> def a = { val x = 100; println(x)}
a: Unit

scala> a
100

scala> a.x
<console>:21: error: value x is not a member of Unit
       a.x
         ^
```

함수에서 매개변수를 로컬변수로 지정

매개변수 이름으로 로컬변수로 지정하면 에러가 발생한다. 동일한 이름에 대한 체크를 해서 재정의를 불가하게 한다.

```
scala> def mul(x:Int) = {
     |    val x : Int = x*3
     | }
<console>:21: error: forward reference extends over definition of value x
         val x : Int = x*3
                       ^


scala> def mul(x:Int) = {
     |    val y = x * 3
     |    y
     | }
mul: (x: Int)Int

scala> mul(3)
res53: Int = 9
```

Class도 매개변수 지정이 가능

클래스를 정의할 때 매개변수로 처리하고 블록표현식에서 매개변수를 참조하면 함수와 동일하게 사용되는 것을 알 수 있다.

인스턴스를 만들때 블럭식이 구동되는 것을 알 수 있고 매개변수는 인스턴스에서 접근할 수 없는 것을 알 수 있다.

```
scala> class A(n:Int, m:Int) {
     |    println(n.getClass, n)
     |    println(m.getClass, m)
     |  }
defined class A

scala> val a = new A(10,20)
(int,10)
(int,20)
a: A = A@42cdc28d

scala> a.n
<console>:14: error: value n is not a member of A
       a.n
         ^
```

모듈에 함수 정의 할때 scope 비교

함수 이름과 동일한 이름을 블록에 지정

모듈에 함수를 정의하고 그 내부의 블록에 동일한 이름으로 변수를 정의했지만 실제 두 scope가 달라 호출될 때 처리가 다르다

```
scala> def time(code : => Any) : Any = {
     |    val time = System.currentTimeMillis
     |    val result = code
     | println("time elapsed : " + (System.currentTimeMillis - time))
     |    result
     | }
time: (code: => Any)Any

scala> def fib(n:Int) : Int = if (n < 2) 1 else fib(n-1) + fib(n-2)
fib: (n: Int)Int

scala> time { fib(20) }
time elapsed : 1
res104: Any = 10946
```

함수와 블록 관계

함수 정의

함수명과 매개변수, 반환값을 정의한 후에 블럭표현식을 정의하면 이 블럭표현식 내에 함수의 매개변수를 활용해서 사용할 수 있다.

함수 정의의 매개변수를 블록 내부에서 참조해서 사용이 가능하다.

```
scala> def add(x:Int, y:Int) :Int = {
     |     println(x.getClass)
     |     println(y.getClass)
     |
     |     x+y
     | }
add: (x: Int, y: Int)Int

scala> add(10,10)
int
int
res83: Int = 20
```

함수 호출

함수 호출 할 때 인자가 하나일 경우는 표현식의 결과가 하나의 반환만을 표시하므로 함수명 다음에 블럭식을 사용 처리

```
scala> def add(x:Int) = x + 10
add: (x: Int)Int

scala> add { 100 }
res88: Int = 110

scala> add({100})
res89: Int = 110
```

클래스와 블럭관계

클래스 정의

클래스도 블럭표현식 없이 정의도 가능하지만 내부 멤버들을 블럭표현식에 정의하면 인스턴스 생성할 때 별도 네임스페이스를 만든다.

B, C 클래스 인스턴스를 만들면 블럭시 내의 필드가 할당된 것을 알 수 있다..

```
scala> class A
defined class A

scala> new A
res90: A = A@125a688c

scala> class B {
     |     val bb = 100
     | }
defined class B

scala> class C {
     |     val bb = 300
     | }
defined class C

scala> val (b,c) = (new B, new C)
b: B = B@132e30cd
c: C = C@5052dcd5

scala> b.bb
res91: Int = 100

scala> c.bb
res92: Int = 300
```

class 정의: 매개변수

클래스를 정의할 때 매개변수로 처리하고 블록표현식에서 매개변수를 참조하면 함수와 동일하게 사용되는 것을 알 수 있다.

인스턴스를 만들때 블럭식이 구동되는 것을 알 수 있고 매개변수는 인스턴스에서 접근할 수 없는 것을 알 수 있다.

```
scala> class A(n:Int, m:Int) {
     |    println(n.getClass, n)
     |    println(m.getClass, m)
     |  }
defined class A

scala> val a = new A(10,20)
(int,10)
(int,20)
a: A = A@42cdc28d

scala> a.n
<console>:14: error: value n is not a member of A
       a.n
         ^
```

Trait에
블럭지정하기

Trait

하나는 추상 필드를 정의하고, 하나는 구상 필드를 정의한다. 두 개를 각각 클래스에서 상속해서 인스턴스를 만들고 필드를 호출해서 처리도 가능하다.

```
scala> trait AAble {
     |    val a : Int
     | }
defined trait AAble

scala> trait Able {
     |    val a = 100
     | }
defined trait Able

scala> class A extends AAble {
     |    val a :Int = 100
     | }
defined class A

scala> class B extends Able
defined class B
```

```
scala> val a = new A
a: A = A@73a65b98

scala> a. a
res102: Int = 100

scala> val b = new B
b: B = B@3fff0199

scala> b.a
res103: Int = 100
```

Object 에
블럭지정하기

object 정의

스칼라에서는 object 키워드를 이용해서 정의하면 클래스의 하나의 인스턴스를 가지는 싱글턴 인스턴스가 만들어진다.

Object는 타입으로 지정할 수 없으므로 인스턴스 생성이 불가

```
scala> object AObj {
     |    val a = 100
     | }
defined object AObj

scala> val obj = new AObj { val b = 200 }
<console>:12: error: not found: type AObj
       val obj = new AObj { val b = 200 }
                     ^
```

인스턴스 생성할 때 블럭지정하기

클래스나 trait로 타입 지정

클래스나 trait를 변수명 다음에 지정해서 타입을 확정할 수 있다.

```
scala> trait AAA {
     |     def print = println("AAA")
     | }
defined trait AAA

scala> class BBB extends AAA
defined class BBB

scala> val a : AAA = new BBB
a: AAA = BBB@dd909bb

scala> a.print
AAA

scala> val b : BBB = new BBB
b: BBB = BBB@3eb53655

scala> b.print
AAA
```

Trait가 타입이 되려면

trait가 정의되고 상속이 되거나 trait를 이용해서 익명이 객체를 만들때 사용된다.

```
scala> trait NoType {
     |    def print = println("NoType")
     | }
defined trait NoType
```

```
scala> val a:  NoType = NoType
<console>:12: error: not found: value NoType
       val a:  NoType = NoType
                        ^


scala> val a:  NoType = new NoType { val c = 0 }
a: NoType = $anon$1@5af2f934
```

아무것도 없는 클래스에서 블럭지정

클래스를 정의하고 인스턴스 생성할 때 블록 표현식을 사용하면 익명 클래스의 인스턴스가 만들어진다.

인스턴스 생성할 때 블록 표현식을 사용하면 실제 필드가 생성되고 직접 참조도 가능하다.

```
scala> class A(n:Int, m:Int) {
     |    println(n.getClass, n)
     |    println(m.getClass, m)
     | }
defined class A

scala> val aa = new A(10,100) {val aa = 100}
(int,10)
(int,100)
aa: A{val aa: Int} = $anon$1@9534545

scala> aa.aa
res86: Int = 100

scala> aa.getClass
res87: Class[_ <: A] = class $anon$1
```

클래스 속성 정의 후 추가 블럭지정

클래스를 정의하고 인스턴스 생성할 때 블록 표현식을 사용하면 익명 클래스의 인스턴스가 만들어진다.

인스턴스 생성할 때 블록 표현식을 사용하면 실제 필드가 생성되고 직접 참조도 가능하다.

```
scala> class E {
     |    println("instance ")
     |    val ee = 100
     | }
defined class E

scala> val e = new E
instance
e: E = E@480bb176

scala> e.ee
res95: Int = 100

scala> val eb = new E { val xx = 300 }
instance
eb: E{val xx: Int} = $anon$1@4392d565

scala> eb.ee
res96: Int = 100

scala> eb.xx
res97: Int = 300
```

Trait 정의 후 인스턴스 생성 할때 추가 블럭지정

인스턴스 정의하는 추가 블록을 지정하면 new 키워드와 trait를 이용해서 인스턴스를 만들 수 있다.

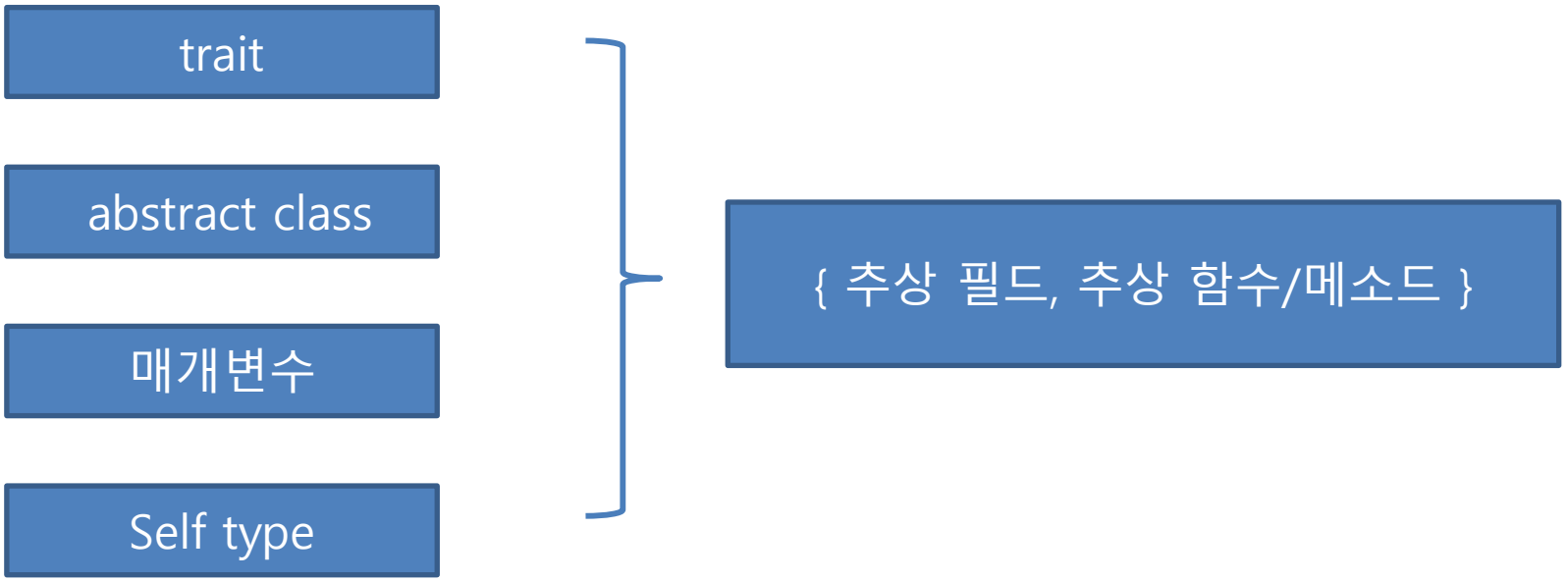
인스턴스 생성할 때 블록 표현식을 사용하면 실제 필드가 생성되고 직접 참조도 가능하다.

```
scala> trait Able
defined trait Able

scala> val a = new Able { val aa= 100 }
a: Able{val aa: Int} = $anon$1@7231ce81

scala> a.aa
res100: Int = 100

scala> a.getClass
res101: Class[_ <: Able] = class $anon$1
```

블록에 추상 타입 사용하기
- 구조적 타입

블록에 추상 타입 지정

블록에 추상타입을 지정해서 trait/abstract/매개변수/self type에 정의해서 사용이 가능하다.



함수 매개변수에 추상타입 선언

함수 내의 인자로 특정 메소드를 가진 임의의 구조적 타입을 지정한다. 실제 클래스의 인스턴스를 전달하면 메소드가 호출되어 처리되는 된다.

```
scala> class D {
     |    def call() = println(" dog ")
     | }
defined class D

scala> class C {
     |    def call() = println(" cat ")
     | }
defined class C
```

```
scala> import scala.language.reflectiveCalls
import scala.language.reflectiveCalls

scala> def call(c:{def call():Unit}) = c.call()
call: (c: AnyRef{def call(): Unit})Unit

scala> call(new D)
 dog

scala> call(new C)
 cat
```

타입 별칭에 구조적 타입 지정

구조적 타입을 타입 별칭으로 지정해서 함수에 선언하면 함수의 매개변수가 더 명확해 보인다.

```
scala> class D {
     |    def call() = println(" dog ")
     | }
defined class D

scala> class C {
     |    def call() = println(" cat ")
     | }
defined class C
```

```
scala> type Callable = {def call() ; Unit }
<console>:1: error: illegal start of declaration
       type Callable = {def call() ; Unit }
                                     ^

scala> type Callable = {def call() : Unit }
defined type alias Callable

scala> def call(c:Callable) = c.call()
call: (c: Callable)Unit

scala> call(new C)
 cat

scala> call(new D)
 dog
```

구조적 타입을 self type 지정

실제 필요한 기능들을 직접 정의해서 구조적 타입으로 지정해서도 사용이 가능하다.

구조적 타입의 메소드는 추상 타입으로 지정한다.

```
scala> trait C {
     |    self : { def aVale : Int
     |             def bVale : Int } =>
     |    def total = aVale + bVale
     | }
defined trait C

scala> class ABC extends C {
     |    def aVale = 1
     |    def bVale = 2
     | }
defined class ABC

scala> val a = new ABC
a: ABC = ABC@c71a6f8

scala> a.total
res40: Int = 3
```

구조적 타입을 명시적으로 표현

각각의 trait에 정의하기 메소드를 분리해서 정의하고 이를 상속하거나 with 키워드를 이용해서 다중상속을 처리

```
scala> trait A {
     |    def aVale : Int
     | }
defined trait A

scala> trait B {
     |    def bVale : Int
     | }
defined trait B

scala> trait C extends A with B {
     |    def total = aVale + bVale
     | }
defined trait C
```

```
trait C {
  def aValue: Int
  def bValue: Int

  def total = aValue + bValue
}
```