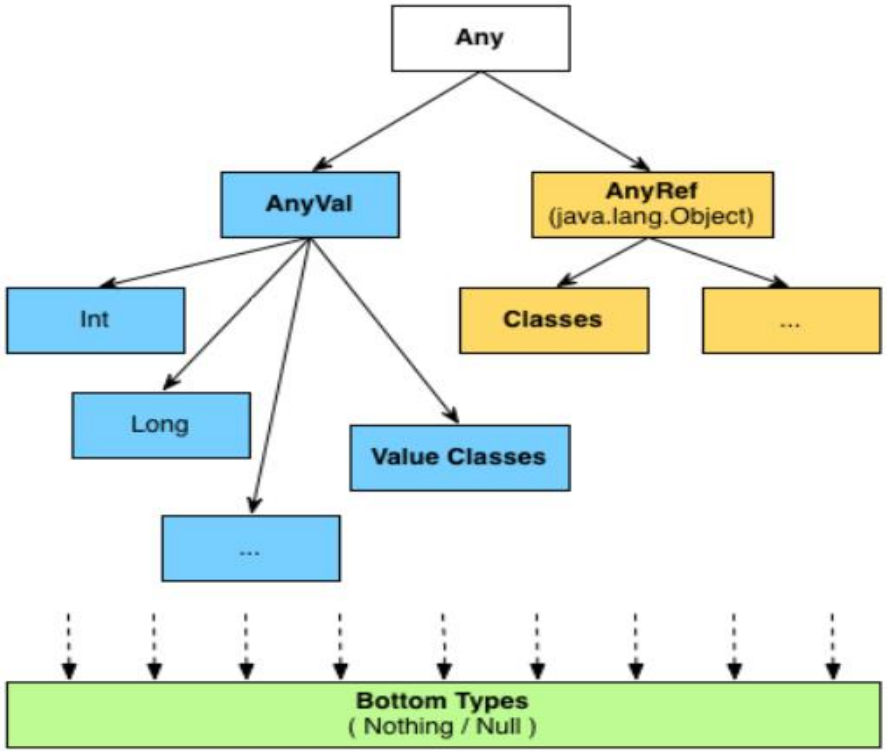
Scala 기본 문법과 함수 이해하기

myjlms99@gmail.com

타입 알아보기

타입 구조

스칼라는 기본적으로 Any를 기준으로 값과 레퍼런스 타입을 구분해서 사용한다.



타입 알아보기

REPL 툴에서 :type 값 또는 함수명 _를 이용해서 실제 타입을 조회할 수 있다.

```
scala> :type 10
Int

scala> :type 10L
Long

scala> :type 10.1
Double

scala> :type 10.1f
Float

scala> def add(x:Int, y:Int) : Int = x+y
add: (x: Int, y: Int)Int

scala> :type add _
(Int, Int) => Int
```

타입 별칭 지정하기

type 키워드를 이용해서 타입 별칭을 지정할 수 있다.

```
scala> type User = String
defined type alias User

scala> val a : User = "String"
a: User = String

scala> a + "abc"
res151: String = Stringabc
```

변수 정의

변수 정의 이해하기:var와 val

val, var는 정의하고 할당하면 바로 실행되어 값을 할당한다. 재할당이 가능한 경우는 var로 정의해서 사용한다.

```
scala> var a = 100
a: Int = 100

scala> a= 300
a: Int = 300

scala> val b = 100
b: Int = 100

scala> b = 300
<console>:19: error: reassignment to val
       b = 300
         ^
```

변수 정의 이해하기: val과 def

val, var는 정의하고 할당하면 바로 실행이 되지만 def로 정의할 때는 호출되기 전까지는 값이 할당되지 않는다.

```
scala> val z = 100
z: Int = 100

scala> def a = 100
a: Int

scala> a
res35: Int = 100
```

Def 로 정의된 경우, 이름으로 호출할 때 실행되어 값을 처리

변수 정의 이해하기 :lazy

lazy를 val 앞에 정의하면 실제 실행을 뒤로 미루지만 한 번 호출되면 값이 할당된다. Def는 항상 이름으로 호출될 때마다 실행되는 것이 다르다.

Def 로 정의된 경우, 이름으로 호출할 때 실행되어 값을 처리

```
scala> lazy val x = 100
x: Int = <lazy>

scala> def zz = 100
zz: Int

scala> x
res37: Int = 100

scala> zz
res38: Int = 100
```

타입 추론

변수에 타입주기

Var 변수에 타입을 지정하면 다른 타입으로 들어오는 경우 에러가 발생한다.

```
scala> var a : Int = 100
a: Int = 100

scala> a = "String"
<console>:19: error: type mismatch;
 found   : String("String")
 required: Int
       a = "String"
           ^
```

변수에 타입확장

변수에 타입을 지정하지만 실제 long 타입은 Int 타입보다 상위이므로 정수를 할당해도 long 타입으로 확장이 된다.

```
scala> var b : Long = 100L
b: Long = 100

scala> b.getClass
res104: Class[Long] = long

scala> b = 100
b: Long = 100

scala> val c = 100
c: Int = 100

scala> c.getClass
res105: Class[Int] = int
```

block 이해하기

블록 이해하기

블록에 여러 개의 로직을 넣어도 마지막 표현식의 결과를 처리한 결과를 넣어서 처리한다.

```
scala> val x = 10
x: Int = 10

scala> val y = 20
y: Int = 20

scala> { x+ y}
res34: Int = 30
```

결과가 없는 블록

블록 내에 변수를 지정하면 블록이 실행된 이후에 참조를 할 수 없다. 이 블록은 실행되면 내부 변수들을 전부 삭제한다.

```
scala> {
     |    var xxx = 100
     |    val yyy = 200
     |    xxx = xxx + yyy
     | }

scala> xxx
<console>:22: error: not found: value xxx
       xxx
       ^


scala> yyy
<console>:22: error: not found: value yyy
       yyy
       ^
```

If 문 이해하기

if 문 처리

if문은 결과값을 반환해서 처리하므로 결과값이 반환되므로 실제 삼항연산자 처럼 처리된다.

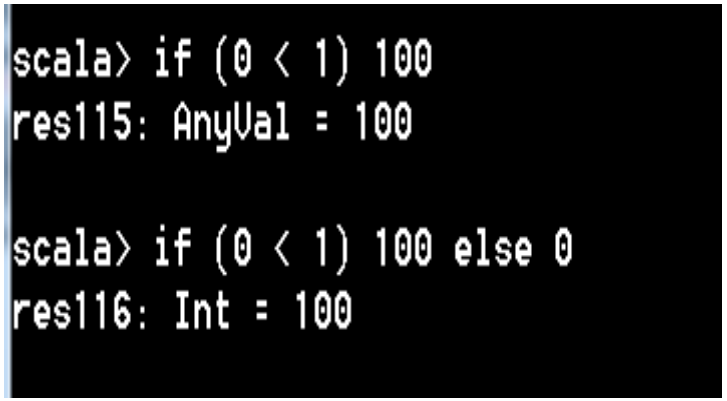
If 문은 실제 표현식으로 사용되므로 결과값을 반환한다.

삼항연산자 처리

복합 제어문은 존재하지 않아 else 문 내의 블록으로 정의된 것을 처리한다.

단일 if 문 처리

if 문을 사용할 경우 else가 없으면 값을 리턴하므로 실제 값이 AnyVal로 처리가된다. 이유는 else쪽이 값이 없으므로 최상위 값으로 처리한다.



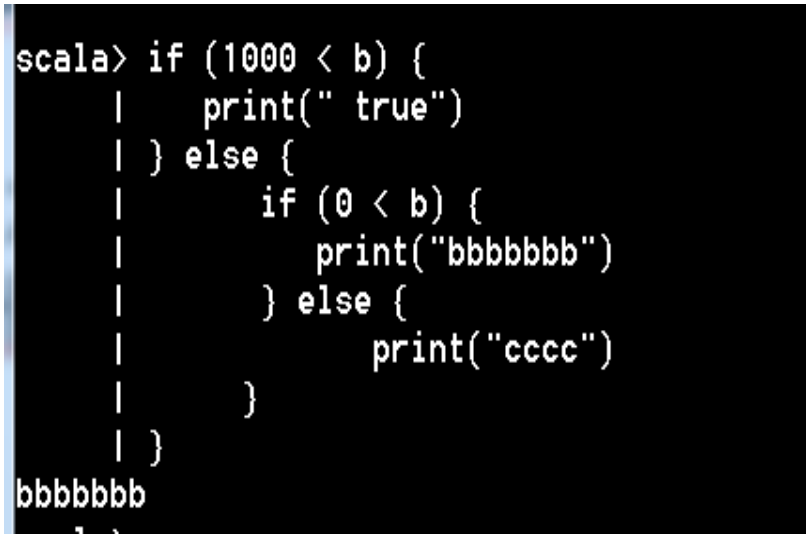
삼항연산자

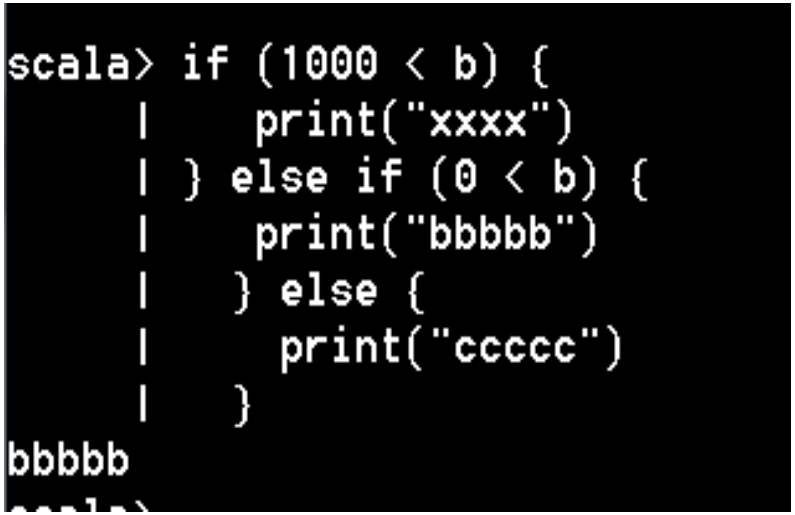
if 문을 사용할 경우 else가 없으면 값을 리턴하므로 실제 값이 AnyVal로 처리가된다. 이유는 else쪽이 값이 없으므로 최상위 값으로 처리한다.

```
scala> val a = 100
a: Int = 100

scala> if (0<a) a else 0
res117: Int = 100

scala> val b = if (0<a) a else 0
b: Int = 100

scala> b
res118: Int = 100
```

복합제어문 기준

if 와 else문을 작성하고 실제 하위의 if와 else문은 블록 내부의 if와 else 문으로 처리되는 것과 동일하다.





패턴 매칭하기

변수에 대한 패턴 매칭

일반적 변수에 대한 값이 하나에 매칭 되어 처리되도록 구성하며 _를 이용해서 default를 만들어준다.

```
scala> val a = 10
a: Int = 10

scala> a match {
     |   case 10 => println(a)
     |   case _ => println("no")
     | }
10
```

```
scala> val b =10
b: Int = 10

scala> b match {
     |   case 11 => println(b)
     |   case _ => println("no")
     | }
no
```

for 문 이해하기

For문 기준

for 문은 실제 컬렉션 타입을 받아서 원소 하나씩 처리하는 구조를 가지고 있다.

```
scala> for (x <- 1 to 3) {
     |   println(x)
     | }
1
2
3

scala> (1 to 3).getClass
res123: Class[_ <: scala.collection.immutable.Range.Inclusive] = class scala.col
lection.immutable.Range$Inclusive
```

For문 필터처리

for 문 내부에 if 문을 이용해서 실제 처리할 값을 필터링해서 처리가 가능하다.

```
scala> for (x <- 1 to 10 if (x % 2 == 0)) {
     |    println(x)
     | }
2
4
6
8
10
```

While /do while문

조건식을 판단해서 처리하는 while문과 일단 한번 실행한 후에 조건식을 처리하는 do while 문이 있다 .

```
scala> import scala.util.control.Breaks._
import scala.util.control.Breaks._

scala> while (true) {
     |   println(" aaaa")
     |   break
     | }
 aaaa
scala.util.control.BreakControl
```

```
scala> do {
     |    println(" value of a : "+a)
     |    a = a + 1
     | }
     | while(a < 20)
 value of a : 10
 value of a : 11
 value of a : 12
 value of a : 13
 value of a : 14
 value of a : 15
 value of a : 16
 value of a : 17
 value of a : 18
 value of a : 19
```

For/while문도 결과값 리턴

순환문도 실제 실행된 결과를 반환한다. 실제 반환값을 Unit으로 처리한다.

```
scala> val a = for (x <- 1 to 3) { println(x) }
1
2
3
a: Unit = ()
```

```
scala> var c = 3
c: Int = 3

scala> val w = while(c>0) {
     |     println(c)
     |     c = c -1
     | }
3
2
1
w: Unit = ()
```

함수 기본 정의

함수 특징

함수는 정의한 후에 함수명과 매개변수를 인자로 호출하면 이름으로 호출하는 것과 동일한 결과를 얻기 위해 블록식을 실행해서 결과를 처리한다.

함수 정의하기

함수는 def 함수명으로 정의하고 실제 실행될 로직은 블록에 정의한다. = 표시는 실제 호출되면 실행된 결과가 할당된다는 것을 의미한다.

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

scala> add(10,10)
res36: Int = 20
```

def 에 변수 할당하듯 실제 함수의 정의도 이름으로 호출하는 것은 동일하다.

함수 정의하기 : return 처리

함수의 처리할 때 블록처리는 기본적으로 마지막 표현식의 결과를 처리하므로 return 문장을 정의하지 않아도 된다.

Return 문이 있으나 없으나 표현식의 결과만 처리할 경우는 동일하다

```
scala> def returnProc(x:String) :String = { return x }
returnProc: (x: String)String

scala> returnProc(" return proc")
res41: String = " return proc"

scala> def returnProc(x:String) :String = { x }
returnProc: (x: String)String

scala> returnProc(" no return ")
res42: String = " no return "
```

함수 반환 처리

Unit : ()

실제 아무값도 없다는 표현을 Unit으로 표현하고 이 인스턴스 값은 ()이다.

```
scala> Unit
res141: Unit.type = object scala.Unit

scala> ()
```

함수 결과값 : Unit 처리

함수 정의할 때 실제 결과값이 없을 경우는 Unit 결과값으로 처리한다.

```
scala> def noReturn(x:String) : Unit = { println(x) }
noReturn: (x: String)Unit

scala> noReturn(" no Return ")
 no Return
```

함수 결과값: 결과를 안보내기

함수 정의할 때 = 표시가 없을 경우는 실제 리턴값이 없는 Unit으로 인식해서 처리한다. 반환값이 없는데 실제 결과를 표현하면 에러 처리

```
scala> def noReturn(x:String)  { println(x) }
noReturn: (x: String)Unit

scala> noReturn(" no Return ")
 no Return

scala> def noReturn(x:String)  { x }
<console>:14: warning: a pure expression does nothing in statement position
       def noReturn(x:String)  { x }
                                 ^
noReturn: (x: String)Unit
```

함수에 매개변수 미지정 처리

매개변수 미지정 처리

함수를 이름으로만 호출할 수 있다. 함수를 호출할 때 ()를 사용하면 에러가 발생한다.

```
scala> def add = println("add")
add: Unit

scala> add
add

scala> add()
<console>:23: error: Unit does not take parameters
       add()
           ^
```

매개변수 미지정 처리

함수 이름 뒤에 매개변수가 없다고 표시하면 함수 이름으로 호출도 가능하고 함수 다음에 ()를 붙여 호출도 가능하다.

```
scala> def add() = println("add")
add: ()Unit

scala> add
add

scala> add()
add
```

함수 매개변수에 초기값 부여

초기값 처리

함수 정의할 경우 특정 매개변수에 초기값을 정의해서 호출할 때 인자로 들어오지 않을 경우 이를 세팅해서 처리해 준다

```
scala> def add(x:Int, y:Int = 100) :Int = { x+y }
add: (x: Int, y: Int)Int

scala> add(10)
res43: Int = 110

scala> add(30,30)
res44: Int = 60
```

매개변수 이름 사용하기

매개변수 이름에 세팅

함수 호출 할 때 이름으로 호출도 가능하다. 이때는 순서에 상관없이 모든 매개변수를 이름으로 처리도 가능하다.

```
scala> def add(x:Int, y:Int = 100) :Int = { x+y }
add: (x: Int, y: Int)Int

scala> add(10)
res43: Int = 110

scala> add(30,30)
res44: Int = 60

scala> def add(x:Int, y:Int = 100) :Int = { x+y }
add: (x: Int, y: Int)Int

scala> add(y= 200, x=100)
res45: Int = 300

scala> add(100, y=300)
res46: Int = 400
```

매개변수 유의사항

이름으로 호출할 경우 함수에 정의된 모든 매개변수를 명기해서 처리해야 한다.

```
scala> def add(x:Int, y:Int = 100) :Int = { x+y }
add: (x: Int, y: Int)Int

scala> add(y=300)
<console>:16: error: not enough arguments for method add: (x: Int, y: Int)Int.
Unspecified value parameter x.
       add(y=300)
           ^
```

가변 매개변수 처리

가변파라미터 정의

가변인자는 매개변수 정의할 때 타입 지정 후에 *를 붙여 표시한다. 실제 내부적으로 배열로 구성된 것을 알 수 있다.

```
scala> def sum(x:Int*) :Unit = println(x.getClass)
sum: (x: Int*)Unit

scala> sum(1,2,3,4)
class scala.collection.mutable.WrappedArray$ofInt
```

가변 매개변수 실행

가변 매개변수를 받으면 배열로 처리되므로 이를 for문으로 받아서 실제 원소별로 분리한 후에 덧하면 결과를 구할 수 있다.

```
scala> def sum(x:Int*) :Int = {
     |    var result = 0
     |    for (v <- x)  { result += v }
     |    result
     | }
sum: (x: Int*)Int

scala> sum(1,2,3,4,5)
res93: Int = 15
```

가변파라미터 정의 및 호출

가변인자로 정의된 것을 실제 합산을 하기 위해 재귀호출로 처리할 때 함수 호출시 _* 로 다시 인자값을 언패킹 처리

```
scala> def sum(x:Int*) : Int = {
     |    if (x.isEmpty) 0 else x.head + sum(x.tail:_*)
     | }
sum: (x: Int*)Int

scala> sum(1,2,3,4)
res50: Int = 10
```

가변인자를 배열로 처리

가변인자는 배열이므로 실제 내부적으로 합산이 필요할 경우 내부의 메소드를 이용해서 계산도 가능하다.

```
scala> def sum(x:Int*) : Int = {
     |    val a = x.sum
     |    a
     | }
sum: (x: Int*)Int

scala> sum(1,2,3,4)
res54: Int = 10
```

블록문으로 인자전달

블록실행 결과 : 단일값

단일 매개변수를 처리할 경우 함수 옆에 블록문을 작성해서 결과를 바로 전달해도 단일 매개변수처럼 처리가 가능하다.

```
scala> def onePara(x:Int) = println(x)
onePara: (x: Int)Unit

scala> onePara { 10 + 10 }
20

scala> onePara(30)
30
```

블록실행 결과 : 가변인자

가변인자로 처리되는 경우도 배열의 원소를 언패킹해 주면 되므로 _*를 이용해서 언패킹해서 처리가 가능하다.

```
scala> def add(x:Int*) : Int = x.sum
add: (x: Int*)Int

scala> add { (1 to 3):_* }
res66: Int = 6

scala> val x = Array(1,2,3)
x: Array[Int] = Array(1, 2, 3)

scala> add(x:_*)
res67: Int = 6
```

순수함수와 비순수 함수 구별하기

순수함수

순수함수는 부수효과가 없이 입력된 인자만 처리해서 결과로 배출하는 함수이므로 반드시 결과값에 대한 매개변수 형을 지정해야 한다.

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

scala> add(10,10)
res60: Int = 20
```

비순수함수

비순수함수는 부수효과를 발생하는 함수로 함수의 처리결과를 반환하는 것이 아니라 다른 파일이나 다른 변수에 특정한 상태를 처리하는 함수이다.

```
scala> def noPure() :Unit = { println(" noPure") }
noPure: ()Unit

scala> noPure()
 noPure
```

재귀호출 이해하기

재귀호출 특징

재귀호출을 하는 함수를 정의할 때는 항상 함수정의할 때 결과에 대한 자료형을 명기해야 내부적으로 결과를 처리할 수 있다.

함수의 결과에 대한 자료형을 명기해야 한다.

자기 자신의 함수를 호출해서 처리하도록 명기해야 한다.

일반 재귀호출

함수를 내부에서 다시 호출해서 처리할 수 있도록 지정해서 처리가 가능하면 종료될 때까지 함수를 호출해서 처리

재귀호출 정의할 때 함수의 종료점을 먼저 로직처리

다음 순서를 처리하도록 인자를 넣고 함수를 호출한다

```
scala> def fact(n:Int) : Int = {
     |    if (n<=1) 1
     |    else n * fact(n-1)
     | }
fact: (n: Int)Int

scala> fact(3)
res68: Int = 6

scala> fact(5)
res69: Int = 120
```

꼬리 재귀호출

꼬리 재귀 호출은 일단 재귀호출할 때 다른 것과 계산이 되면 안되므로 실제 계산된 결과를 인자로 전달을 받아 처리해야 한다.

재귀호출 정의할 때 함수의 종료점을 먼저 로직처리

다음 순서를 처리하도록 인자를 넣고 함수를 호출한다

```
scala> def fact(n:Int, result:Int = 1) : Int = {
     |     if (n <= 1) result
     |     else fact(n-1, result * n)
     | }
fact: (n: Int, result: Int)Int

scala> fact(3)
res74: Int = 6

scala> fact(5)
res75: Int = 120
```

꼬리 재귀호출 어노테이션

꼬리재귀호출로 구성된 여부를 확인하기 위해 어노테이션을 사용해서 점검이 가능하다.

```
scala> import scala.annotation.tailrec
import scala.annotation.tailrec

scala> @tailrec
     | def fact(n:Int, result:Int = 1) : Int = {
     |    if (n <= 1) result
     |    else fact(n-1, result * n)
     | }
fact: (n: Int, result: Int)Int

scala> fact(3)
res76: Int = 6

scala> fact(5)
res77: Int = 120
```

꼬리 재귀호출 테스트

어노테이션을 붙이고 재귀호출 함수를 작성할 경우 꼬리재귀가 아니면 실제 구성이 잘못 되었다고 에러를 처리한다.

```
scala> import scala.annotation.tailrec
import scala.annotation.tailrec

scala> @tailrec
     | def fact(x:Int) : Int = {
     |    if (x <= 1) 1
     |    else x * fact(x-1)
     | }
<console>:23: error: could not optimize @tailrec annotated method fact: it conta
ins a recursive call not in tail position
         else x * fact(x-1)
                  ^
```

Call by Value,Name

Call by value

함수에 실제 값이 전달되므로 값은 복사되어 처리되도록 구성되어 한번 세팅된 값이 처리된다.

```
scala> def somethon():Int = {
     |    println(" calling Something")
     |    1
     | }
somethon: ()Int

scala> def callByValue(x:Int) = {
     |    println("x1 ="+ x)
     |    println("x2 ="+ x)
     | }
callByValue: (x: Int)Unit

scala> callByValue(somethon())
 calling Something
x1 =1
x2 =1
```

```
scala> def callByName(x: => Int) = {
     |    println("x1 = " + x)
     |    println("x2 = " + x)
     | }
callByName: (x: => Int)Unit

scala> callByName(somethon())
 calling Something
x1 = 1
 calling Something
x2 = 1
```

Call by name 1

위에 정의된 somethon 함수를 실행한 것을 전달하지만 실제 내부적으로는 함수의 이름으로 전달 되어 이름으로 호출될 때마다 실행되는 것을 알 수 있다.

```
scala> def callByName(x: => Int) = {
     |     println("x1 = " + x)
     |     println("x2 = " + x)
     | }
callByName: (x: => Int)Unit

scala> callByName(somethon())
 calling Something
x1 = 1
 calling Something
x2 = 1
```

Call by Name 2

매개변수를 지정할 때 특정 이름으로 전달해서 그 이름이 표현식에서 실행될 때 그 때 처리되는 방식이다.

매개변수 지정 할 때 변수명 : => 타입

출력되는 순서는 호출된 함수이고 실제 이름이 처리될 때 매개변수로 전달 된 함수가 실행

```
scala> def time() = {
     | println("Getting time in nano seconds")
     | System.nanoTime
     | }
time: ()Long

scala> def delayed(t : => Long) = {
     |   println(" In delayed method")
     |   println("Param: " +t)
     | }
delayed: (t: => Long)Unit

scala> delayed(time())
 In delayed method
Getting time in nano seconds
Param: 2153433224067351
```

Call by Name 이용 재귀처리

두개의 call by name 매개변수를 받아서 하나는 조건식 하나는 실제 구현을 하면서 재귀호출하면서 처리도 가능하다

```
scala> def myWhile(cond : => Boolean, body : => Unit) {
     |    if (cond) { body; myWhile(cond,body) }
     | }
myWhile: (cond: => Boolean, body: => Unit)Unit

scala> var x = 3
x: Int = 3
```

```
scala> myWhile(x !=0, { println(x); x=x-1})
3
2
1
```

Call by Name을 한번 호출만 처리

lazy val를 이용해서 call by name으로 들어온 매개변수를 한번만 실행하고 처리하도록 평가를 제한할 수도 있다.

```
scala> def foo2(cond:Boolean)(bar: => String) = {
     |    if (cond) bar + bar
     |    else ""
     | }
foo2: (cond: Boolean)(bar: => String)String

scala> foo2(true) { println("ffff");"Hello"}
ffff
ffff
res251: String = HelloHello
```

```
scala> def foo(cond:Boolean)(bar: => String) = {
     |    lazy val lazyBar = bar
     |    if (cond) lazyBar + lazyBar
     |    else ""
     | }
foo: (cond: Boolean)(bar: => String)String

scala> foo(true) { "Hello"}
res249: String = HelloHello

scala> foo(true) {println(" 111 "); "World"}
 111
res250: String = WorldWorld
```

람다함수 이해하기

람다함수의 특징

고차함수는 함수를 매개변수로 받거나 함수를 결과값으로 전달해서 처리하는 함수의 특징이다.

함수를 정의하고 바로 실행시킬 수 있다.

```
scala> (x:Int, y:Int) => x+y
res56: (Int, Int) => Int = $$Lambda$1660/806766230@210274a2

scala> ((x:Int, y:Int) => x+y)(10,10)
res57: Int = 20
```

함수도 일급객체

함수도 객체

함수도 다른 객체들처럼 일급 객체의 특징을 따른다. 이는 함수도 클래스에 의해 만들어진 하나의 객체라는 것이다.

- 변수에 할당이 가능하다
- 함수의 매개변수로 전달이 가능하다
- 반환값으로 전달이 가능하다

함수도 객체

함수를 정의하고 해당 클래스를 확인하면 하나의 클래스로 만들어진 것을 알 수 있다.

함수의 이름이 변수명과 다르므로 실제 생성된 함수와 _를 이용해서 함수 내부를 접근해서 조회한다.

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

scala> (add _).getClass
res96: Class[_ <: (Int, Int) => Int] = class $$Lambda$1713/1782124418

scala> (add _).getClass.getName
res97: String = $$Lambda$1714/1209225304
```

함수에 변수 할당할 때 주의

함수를 정의하고 변수에 할당할 경우 매개변수 및 결과값에 대한 추론을 할 수 있도록 지정해서 처리해야 한다. 함수명 뒤에 _를 붙여 변수에 할당해야 한ㄷ.

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

scala> val a = add
<console>:19: error: missing argument list for method add
Unapplied methods are only converted to functions when a function type is expect
ed.
You can make this conversion explicit by writing `add _` or `add(_,_)` instead o
f `add`.
       val a = add
               ^


scala> val a = add _
a: (Int, Int) => Int = $$Lambda$1708/1419871684@6b7d79d2

scala> a(5,5)
res92: Int = 10
```

함수에 변수 할당할 때 주의

함수를 정의하고 변수에 할당할 경우 매개변수 및 결과값에 대한 추론을 할 수 있도록 지정해서 처리해야 한다. 함수명 뒤에 _를 붙여 변수에 할당해야 한ㄷ.

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

scala> val a = add
<console>:19: error: missing argument list for method add
Unapplied methods are only converted to functions when a function type is expect
ed.
You can make this conversion explicit by writing `add _` or `add(_,_)` instead o
f `add`.
       val a = add
               ^


scala> val a = add _
a: (Int, Int) => Int = $$Lambda$1708/1419871684@6b7d79d2

scala> a(5,5)
res92: Int = 10
```

람다함수 변수 할당

변수를 정의할 때 함수를 받을 수 있도록 정의하고 람다함수를 전달하고 이 변수를 호출하면 람다함수가 실행된다.

함수에 이름이 없으므로 Lambda로 출력한다.

```
scala> val add : (Int, Int)=> Int = (x:Int, y:Int) => x+y
add: (Int, Int) => Int = $$Lambda$1662/1982801469@4b4ca66

scala> add(10,10)
res58: Int = 20

scala> add.getClass.getName
res59: String = $$Lambda$1662/1982801469
```

고차함수 이해하기

고차함수의 특징

고차함수는 함수를 매개변수로 받거나 함수를 결과값으로 전달해서 처리하는 함수의 특징이다.

함수를 정의할 때 함수를 매개변수로 전달

함수를 결과값으로 전달

매개변수에 함수 지정하기

함수의 시그너처를 지정하면 함수 자료형이 되므로 이를 함수 정의할 때 매개변수로 지정이 가능하다

```
scala> def add(x:Int, y:Int, func:(Int,Int)=>Int) : Int = {
     |    func(x,y)
     | }
add: (x: Int, y: Int, func: (Int, Int) => Int)Int

scala> def addFunc(x:Int, y:Int) : Int = x+y
addFunc: (x: Int, y: Int)Int

scala> add(10,10,addFunc)
res55: Int = 20
```

함수 타입 매개변수
(제너릭 처리)

제너릭 함수: 단일 인자

다양한 자료형을 처리할 수 있도록 타입 매개변수를 지정해서 처리가 가능하다. 호출시에 실제 매핑되는 타입을 지정해서 처리한다.

```
scala> def generic[A](x:A) : A = x
generic: [A](x: A)A

scala> generic[Int](100)
res78: Int = 100

scala> generic[String]("String")
res79: String = String
```

제너릭 함수: 두개 인자

다양한 자료형을 처리할 수 있도록 타입 매개변수를 지정해서 처리가 가능하다. 호출시에 실제 매핑되는 타입을 지정해서 처리한다.

```
scala> def twoPara[A,B](x:A, y:B) : A = x
twoPara: [A, B](x: A, y: B)A

scala> twoPara[Int, String](100,"xxx")
res81: Int = 100

scala> def twoPara[A,B](x:A, y:B) :B = y
twoPara: [A, B](x: A, y: B)B

scala> twoPara[Int, String](100,"XXX")
res82: String = XXX
```

제너릭 함수 : 함수 전달 처리

제너릭 함수를 만들기 위해서는 매개변수에 대한 타입을 별도로 매개화하고 실제 별도로 계산하는 함수를 전달해서 처리한다

```
scala> def add[A](x:A, y:A, f:(A,A) => A) = f(x,y)
add: [A](x: A, y: A, f: (A, A) => A)A

scala> add[Int](10,10, (x,y) => x+y)
res53: Int = 20

scala> add[String]("Hello", "World", (x,y) => x+ y)
res54: String = HelloWorld
```

두개 타입을 받고 연산 처리

두개의 타입을 받아 연산을 처리하면 에러가 발생하므로 하나의 함수를 더 받아서 처리하면 두개의 타입 매개변수도 처리가 가능

```
scala> def twoPara[A,B](x:A, y:B) :B = {y + x }
<console>:22: error: type mismatch;
 found   : A
 required: String
       def twoPara[A,B](x:A, y:B) :B = {y + x }
                                         ^


scala> def twoPara[A,B](x:A, y:B, f:(A,B) => B) =  f(x,y)
twoPara: [A, B](x: A, y: B, f: (A, B) => B)B

scala> twoPara[Int,String](10,"xxx", (x:Int,y:String) => x+y)
res83: String = 10xxx
```

내부함수 처리

내부 함수 정의 및 실행

외부 함수를 정의할 때 내부 함수를 정의하고 외부 함수 매개변수를 내부함수에 전달해서 실행하도록 처리하면 내부함수 결과값이 외부로 전달된다.

```
scala> def outer(x:Int, y:Int) = {
     |    def inner(a:Int, b:Int) = a+b
     |    inner(x,y)
     | }
outer: (x: Int, y: Int)Int

scala> outer(10,20)
res85: Int = 30
```

내부 함수 정의 후 전달

함수 내부에 함수를 정의하면 실제 내부함수가 외부로 전달도 가능하다. 함수 이름과 _를 붙여 매개변수도 전달해야 한다

```
scala> def outer(x:Int) = {
     |    def inner(y:Int) = x+y
     |    inner _
     | }
outer: (x: Int)Int => Int

scala> val inner = outer(10)
inner: Int => Int = $$Lambda$1703/909880746@36517cd6

scala> inner(20)
res84: Int = 30
```

네임스페이스와 스코프

일반 함수에서 모듈 변수 참조

함수 내부에 없는 변수가 있으면 모듈에 정의된 변수를 검색해서 처리한다.

```
scala> val a = 100
a: Int = 100

scala> def add(x:Int) : Int = a + x
add: (x: Int)Int

scala> add(200)
res100: Int = 300

scala> a
res101: Int = 100
```

내부함수에서 외부함수 변수 참조

함수 내부에 없는 변수가 있으면 외부 함수에 있는 변수를 참조해서 처리한다.

```
scala> def outer() = {
     |    val x = 100
     |    def inner(y:Int) = x+y
     |    inner _
     | }
outer: ()Int => Int

scala> val o = outer()
o: Int => Int = $$Lambda$1755/551447238@4166b58c

scala> o(300)
res102: Int = 400
```

내부함수에서 모듈 변수 참조

함수 내부에 없는 변수가 있으면 외부 함수를 검색하고 없으면 모듈을 검색해서 처리한다.

```
scala> val x = 100
x: Int = 100

scala> def outer() = {
     |    def inner(y:Int) = x+y
     |    inner _
     | }
outer: ()Int => Int

scala> val c = outer()
c: Int => Int = $$Lambda$1756/214555615@2736045f

scala> c(300)
res103: Int = 400
```

커링 함수 처리

매개변수 그룹화 하기

함수를 정의할 때 매개변수를 그룹화해서 분리해서 처리할 수 있다. 이때는 부분적으로 인자를 받아 처리하므로 함수 호출도 나눠서 처리해야 한다.

```
scala> def paraGroup(x:Int)(y:Int) = x+y
paraGroup: (x: Int)(y: Int)Int

scala> paraGroup(5)(6)
res87: Int = 11

scala> paraGroup(5,6)
<console>:20: error: too many arguments (2) for method paraGroup: (x: Int)(y: In
t)Int
       paraGroup(5,6)
                 ^
```

매개변수 그룹화 하고 변수 할당

함수를 그룹화해서 처리할 경우 함수호출하고 변수할당할 경우 그룹화된 매개변수를 _로 처리해야 다음에 더 호출해서 처리가 가능하다

```
scala> def paraGroup(x:Int)(y:Int) = x+y
paraGroup: (x: Int)(y: Int)Int

scala> val a = paraGroup(5) _
a: Int => Int = $$Lambda$1706/815909861@5fa1d419

scala> a(6)
res89: Int = 11

scala> val b = paraGroup(5)
<console>:19: error: missing argument list for method paraGroup
Unapplied methods are only converted to functions when a function type is expect
ed.
You can make this conversion explicit by writing `paraGroup _` or `paraGroup(_)(
_)` instead of `paraGroup`.
       val b = paraGroup(5)
                         ^
```

함수를 리턴해서 커링 처리

두 개의 매개변수를 처리할 경우 이를 부분으로 나눠서 처리할 때 내부 람다함수를 반환해서 처리한다.

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

scala> val a = add(100)
a: Int => Int = $$Lambda$1707/1362105037@54a2cd5a

scala> a(200)
res90: Int = 300
```

클로저란

함수의 매개변수는 기본 val 처리

외부 함수에 정의된 매개변수는 기본으로 val 변수이므로 재할당이 불가하다. 자유변수로 사용될 경우는 읽는 용도로만 사용이 가능

```
scala> def outer(x:Int) = {
     |    def inner(y:Int) :Int = {
     |       x = x+y
     |       x
     |    }
     |    inner _
     | }
<console>:26: error: reassignment to val
            x = x+y
              ^
```

```
scala> def outer(x:Int) = {
     |    def inner(y:Int) :Int = {
     |        x+y
     |    }
     |    inner _
     | }
outer: (x: Int)Int => Int

scala> val o = outer(10)
o: Int => Int = $$Lambda$1782/652067544@545bfbd2

scala> o(10)
res134: Int = 20

scala> o(20)
res135: Int = 30
```

자유변수를 가진 내부함수를 리턴

외부 함수에 있는 자유변수가 내부함수에 물려 외부로 전달되어 처리되는 환경을 클로저라 한다.

```
scala> def outer() = {
     |    var x = 0
     |    def inner(y:Int) :Int = {
     |       x = x+y
     |       x
     |    }
     |    inner _
     | }
outer: ()Int => Int

scala> val o = outer()
o: Int => Int = $$Lambda$1781/558568087@4c5396b4

scala> o(10)
res132: Int = 10

scala> o(11)
res133: Int = 21
```

Function object 사용하기

매개변수를 튜플 처리하기

여러 개의 매개변수를 가진 함수를 가지고 매개변수를 튜플로 변경하고자 하면 Function object의 tupled를 이용해서 처리

```
scala> def mfunc(i:Int, j:Int, k:Int) = i+j+k
mfunc: (i: Int, j: Int, k: Int)Int

scala> val mt:((Int,Int,Int)) => Int = Function.tupled(mfunc _)
mt: ((Int, Int, Int)) => Int = scala.Function$$$Lambda$1934/335772706@45816b0a

scala> mfunc(1,2,3)
res267: Int = 6

scala> mt((1,2,3))
res268: Int = 6
```

매개변수를 튜플을 풀어 처리하기

매개변수가 튜플로 처리된 것을 여러 매개변수로 분리 Function object untupled 메소드를 이용해서 처리

```
scala> def mfunc(i:Int, j:Int, k:Int) = i+j+k
mfunc: (i: Int, j: Int, k: Int)Int

scala> val mt:((Int,Int,Int)) => Int = Function.tupled(mfunc _)
mt: ((Int, Int, Int)) => Int = scala.Function$$$Lambda$1934/335772706@45816b0a

scala> mfunc(1,2,3)
res267: Int = 6

scala> mt((1,2,3))
res268: Int = 6

scala> val umt : (Int,Int,Int) => Int = Function.untupled(mt)
umt: (Int, Int, Int) => Int = scala.Function$$$Lambda$1935/1248547613@5e8e2ff

scala> umt(1,2,3)
res269: Int = 6
```

함수 체인 만들기

함수를 정의할 경우 sequence로 묶어서 연속적으로 처리될 수 있도록 체인화 처리를 한다. 매개변수 타입이 동일해야 한다.

```
scala> val plus : (Int) => Int = (i:Int) => i+i
plus: Int => Int = $$Lambda$1939/2116366828@72250e4a

scala> val funcSeq = Seq(plus,plus)
funcSeq: Seq[Int => Int] = List($$Lambda$1939/2116366828@72250e4a, $$Lambda$1939
/2116366828@72250e4a)

scala> val chain = Function.chain(funcSeq)
chain: Int => Int = scala.Function$$$Lambda$1940/1156838092@78bbec21

scala> chain(9)
res270: Int = 36
```

함수 체인 작동원리 1

동일한 함수를 두번 처리할 때 실제 값이 처리되는 결과를 보면 첫번째는 인자로 받은 것을 처리하고 두번째는 내부에서 생성된 거슬 계산한다.

```
scala> def plus = (l:Seq[Int]) => { println(l);val a = l.sum; List(a,l.head) }
plus: Seq[Int] => List[Int]

scala> (Function.chain(List(plus,plus)))(List(1,2,3,4))
List(1, 2, 3, 4)
List(10, 1)
res281: List[Int] = List(11, 10)
```

함수 체인 작동원리 2

실제 들어오는 데이터가 다양한 종류일 때는 Any 로 지정해서 다양한 자료형을 처리할 수 있도록 한다

```
scala> def t = (l:Seq[Any]) => l.tail
t: Seq[Any] => Seq[Any]

scala> def r = (l:Seq[Any]) => l.reverse
r: Seq[Any] => Seq[Any]

scala> (Function.chain(List(r,t)))(List(1,2,3,4,5))
res273: Seq[Any] = List(4, 3, 2, 1)
```