Scala generic 표현

abstract type, bounded type, type variance

myjlms99@gmail.com

추상 타입
abstract type

Abstract type

추상타입은 다양한 클래스로 해석할 수 있는 명세이다. 추상 타입은 보편적으로 전달받을 수 있는 즉 허용가능한 타입의 범위를 지정하기 위해 타입 매개변수로 사용된다.

트레이트 내부에 type 키워드를 이용해서 추상타입 선언

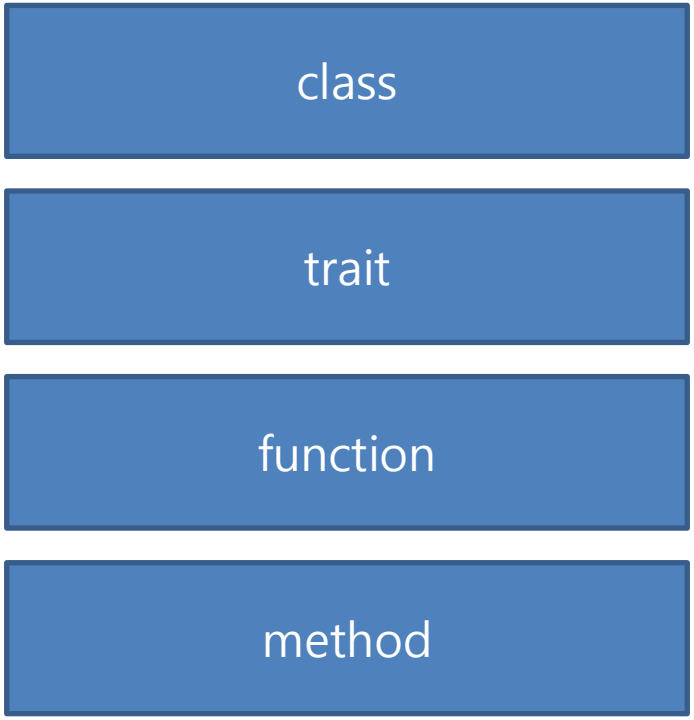
추상타입을 지정한 것을 상속하고 이에 명확한 타입을 지정하고 클래스까지 만듦

```
scala> class A
defined class A

scala> trait B {
     |   type T
     |   def create : T
     | }
defined trait B

scala> trait C extends B {
     |     type T = A
     |     def create = new A
     | }
defined trait C

scala> class D extends C
defined class D

scala> (new D).create
res18: A = A@4e29aac7
```

타입 매개변수
type parameter

타입매개변수 가능한 곳

4개의 대표적인 곳에 타입인자를 사용할 수 있다.



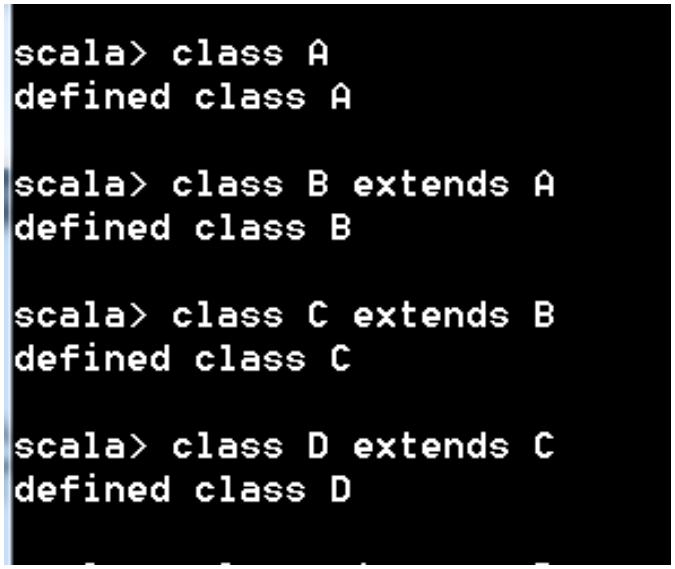
타입매개변수 표기법

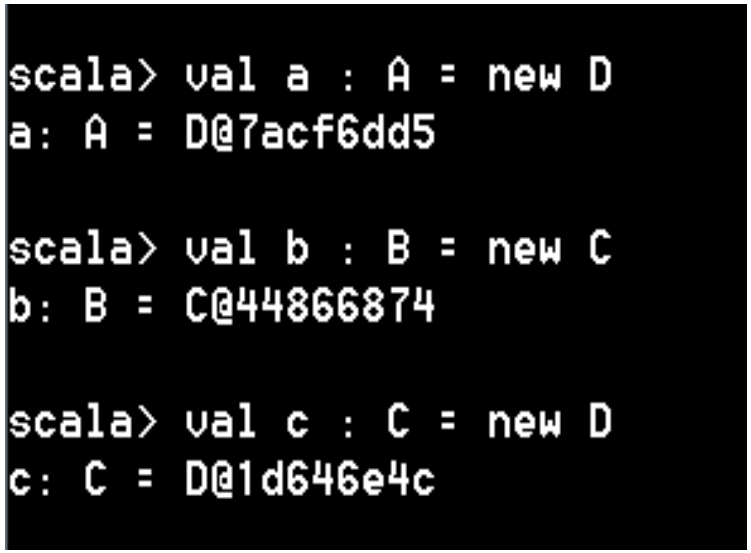
4개의 대표적인 곳에 대괄호를 사용해서 내부에 타입인자를 사용할 수 있다.

class/trait/function/method[타입인자]

타입 매개변수 미사용 처리

일반적으로 상위 타입에 하위 타입을 지정해서 처리가 가능하다.





타입매개변수 사용

타입 매개변수를 정의하면 특정한 클래스가 아닌 매개변수로 사용된다

클래스 명과 타입 매개변수 명이 같지만 실제 타입매개변수는 단순히 인자를 받는 변수로 처리된다.

```
scala> class A
defined class A

scala> class B[A]
defined class B

scala> new B[Int]
res47: B[Int] = B@2a21aeb5
```

타입 매개변수 종류

타입 가변성을 지정하면 그 타입의 상속관계를 기준으로 처리하지만 명확한 타입을 추가적으로 지정하고자 하면 경계 타입을 지정해서 처리할 수 있다.

타입 변성 : type variance
=> 경계 타입보다 덜 제한적 처리, 주로 타입 치환

경계가 있는 타입 : Bounded type
=> 제한적 처리, 타입이 어느 범주 처리가 중요

Type Parameter 지정하기

제너릭 클래스

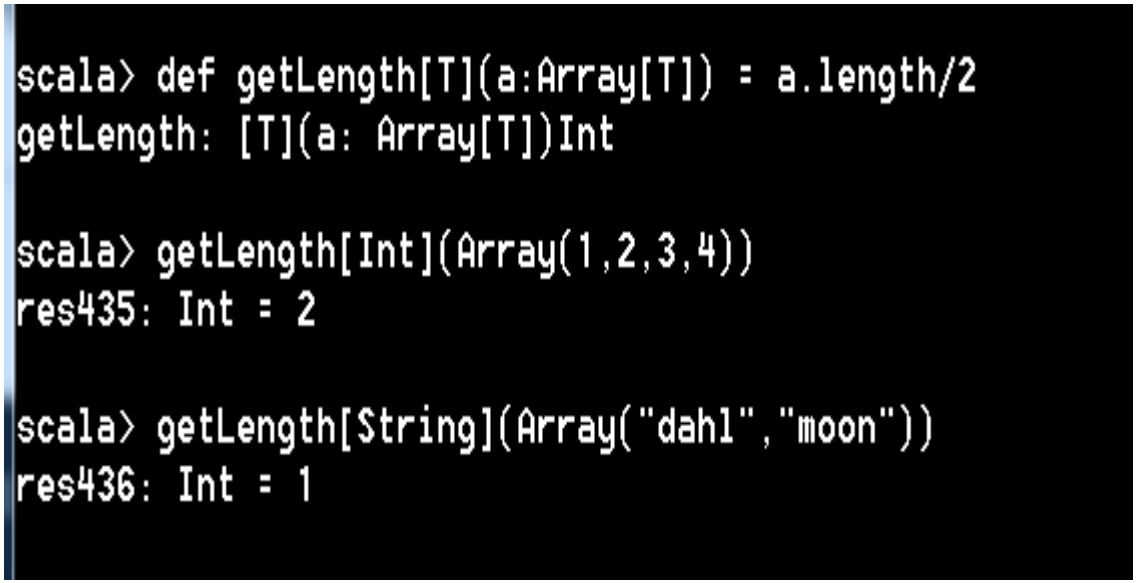
하나 이상의 타입인자가 있는 클래스는 '제네릭 클래스'이고 타입인자를 실제 타입으로 대체하면 일반 클래스이다.

생성자에 추론을 하거나 명시적으로 타입을 지정

```
scala> class A[T,S](val f:T, val s:S)
defined class A

scala> val p1 = new A(42, "Dahl")
p1: A[Int,String] = A@3dfd110d

scala> val p2 = new A[Int, String](42,"Moon")
p2: A[Int,String] = A@1b384e38
```

제너릭 함수

타입인자를 이용해서 함수의 매개변수를 다양한 결과를 받을 수 있도록 정의해서 사용한다



함수에 타입 매개변수 주의 사항

함수에 타입인자로 처리할 때 실제 로직에서 함수의 연산자 등을 사용할 때 실제 인식을 하지 못하므로 함수를 전달해서 처리하는 방식을 사용한다.

```
scala> def add[T](x:T, y:T) : T = x + y
<console>:19: error: type mismatch;
 found   : T
 required: String
       def add[T](x:T, y:T) : T = x + y
                                  ^
```

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

scala> add[Int](10,10,(x:Int,y:Int) => x+y)
res49: Int = 20
```

Type variance
상속관계로 제한하기

공변성, 반공변성, 무공변성

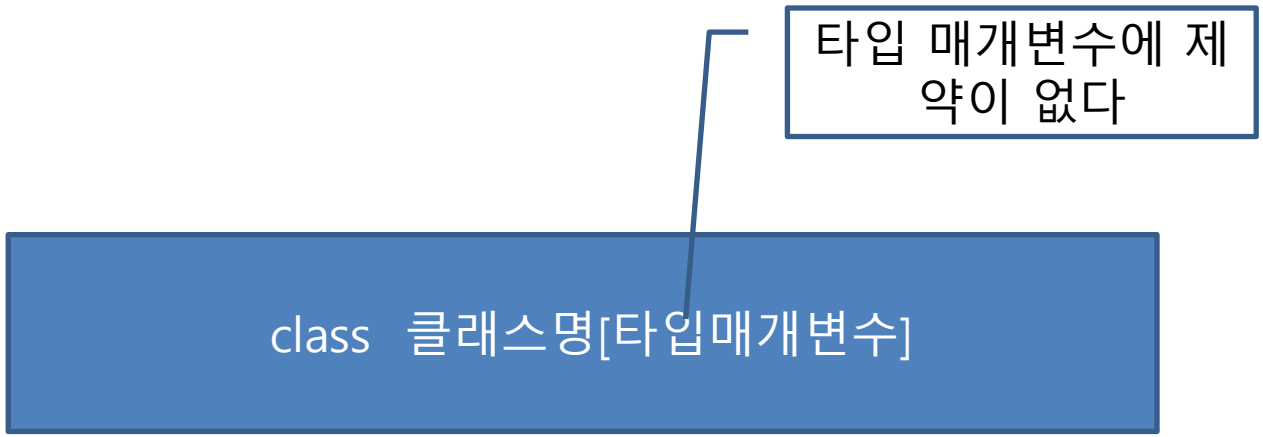
공변성은 자식 타입으로 치환이 가능하고
반공변성은 부모 타입으로 치환이 가능하다.
무공변성은 지정된 타입으로만 처리된다.

상속에 따른 변성
타입시스템을 이해 하기 위해선 **상속에 다른 변성(variance)**을 이해 해야 한다.

	의미	스칼라 표기
공변성(covariant)	C[T']는 C[T]의 하위 클래스이다	[+T]
반공변성(contravariant)	C[T]는 C[T']의 하위 클래스이다	[-T]
무공변성(invariant)	C[T]와 C[T']는 아무 관계가 없다	[T]

무공변성

타입 매개변수을 선언할 경우 실제 인스턴스 생성할 때 타입이 한정되면 상속관계와 상관없이 한정되어 처리된다.



변하지 않는 타입 매개변수 사용

주어진 타입 매개변수에 따라만 결정되어 처리된다.

```
scala> class A[T](val a:T)
defined class A

scala> val a = new A[Int](10)
a: A[Int] = A@7be98a17

scala> a.a
res21: Int = 10

scala> val b = new A[Double](10)
b: A[Double] = A@37d11b78

scala> b.a
res22: Double = 10.0
```

변하지 않는 타입에 대한 변수할당

타입 매개변수로 지정할 경우 실제 상위 클래스 내에 하위클래스로 처리할 경우 명확히 클래스들을 이해하지 못한다.

```
scala> class A
defined class A

scala> class B extends A
defined class B

scala> class Item[T](a:T) {
     |    def get : T = a
     | }
defined class Item

scala> val c : Item[A] = new Item[B](new B)
<console>:14: error: type mismatch;
 found   : Item[B]
 required: Item[A]
Note: B <: A, but class Item is invariant in type T.
You may wish to define T as +T instead. (SLS 4.5)
       val c : Item[A] = new Item[B](new B)
                         ^
```

타입 변성 처리 기준

일반적으로 타입 매개변수를 지정할 때는 클래스와 반환값, 메소드나 함수 등을 구분해서 타입 가변성을 표시하며, "제네릭 유형"개념이며 매개 변수화 된 유형을 메소드에 전달할 수 있는 규칙을 정의합니다

메소드나 함수는 반공변성

클래스 정의 및 반환값은 공변성

타입변성 처리기준

공변성은 현재 클래스의 하위 클래스를 기준으로 처리하고 반공변성은 상위 클래스를 기준으로 처리하는 것을 알 수 있다.

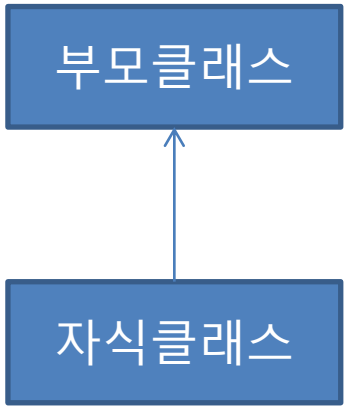
```
scala> class BBB[+Int]
defined class BBB

scala> new BBB
res30: BBB[Nothing] = BBB@6cd475c4

scala> class BBB[-Int]
defined class BBB

scala> new BBB
res31: BBB[Any] = BBB@44f3dd07
```

```
scala> class CCC[+AnyVal]
defined class CCC

scala> new CCC
res32: CCC[Nothing] = CCC@2b475ff5

scala> class CCC[-AnyVal]
defined class CCC

scala> new CCC
res33: CCC[Any] = CCC@4926ce55
```

변수 할당은 공변성 처리

클래스를 정의할 대 타입 베리언스로 처리해야 상위 클래스로 지정된 변수에 할당이 가능하다.

공변성은 부모 클래스로 지정된 것을 자식 클래스로 대입해도 처리된다.




```
scala> class ItemV[+T](a:T) {
     |    def get : T = a
     | }
defined class ItemV

scala> val c : ItemV[A] = new ItemV[B](new B)
c: ItemV[A] = ItemV@c20ac4a
```

타입변성 : 공변성 사용

타입 매개변수에 타입변성을 공변성으로 지정하면 본인부터 하위 타입이 가능하므로 변수에 할당할 때도 처리가 가능하다.

```
scala> class A
defined class A

scala> class B extends A
defined class B

scala> class Item[+T](a:T) {
     |     def get : T = a
     | }
defined class Item


scala> val c : Item[A] = new Item[B](new B)
c: Item[A] = Item@fdafd25
```

서브타입이 기본타입의 변수에 할당이 되었다

변수 할당을 두 개 동시 사용

바운드 타입과 타입변성을 다 사용해서 처리도 가능하다.

```
scala> class ItemU[+T <: A](a:T) {
     |    def get : T = a
     | }
defined class ItemU

scala> val c : ItemU[A] = new ItemU[B](new B)
c: ItemU[A] = ItemU@68ea4307
```

타입 매개변수를 특정해서 제한하기(Bounded Type)

타입매개변수 제한하기

타입 매개변수를 지정할 때 다형성에 대한 특정 경계를 제한할 수 있다.

특정 상속 관계의 타입 매개변수(변성 지정)	<: >: <% :	특정 범위 제한 타입
	특정 범위 제한을 위한 연산	

Bounded type 사용 이유 1

매개변수화한 타입이나 메서드를 정의할 때, 타입 매개변수에 대해 구체적인 바운드를 지정할 수 있다.

```
scala> class Person {
     |    val name = "Person"
     | }
defined class Person

scala> class Student extends Person {
     |    override val name = "Student"
     | }
defined class Student

scala> class Other extends Person {
     |    override val name = "Other "
     | }
defined class Other

scala> def getName[T](p:Seq[T]) = p.map(_.name)
<console>:11: error: value name is not a member of type parameter T
       def getName[T](p:Seq[T]) = p.map(_.name)
                                        ^
```

Bounded type 사용 이유 2

타입 매개변수가 특정 타입으로 인식이 필요하므로 타입 매개변수에 올 수 있는 클래스를 한정하는 것을 말한다.

```
scala> def getName[T <: Person](p:Seq[T]) = p.map(_.name)
getName: [T <: Person](p: Seq[T])Seq[String]

scala> val a = Seq(new Person, new Student, new Other)
a: Seq[Person] = List(Person@31ad911d, Student@1b97fed6, Other@2d8d1f64)

scala> getName(a)
res34: Seq[String] = List(Person, Student, "Other ")
```

상위 Bound 이해하기

상위경계와 하위경계

상위경계과 하위경계는 정해진 경계를 중심으로 상위 타입과 하위 타입을 처리한다.

상위경계 [T <: A] 로 표시 해당 A 타입과 그 하위 타입들을 포함해서 처리

Upper Bound (T <: Pet) : T는 적어도 Pet 클래스 나 Pet의 하위 클래스를 상속 한 모든 클래스에 적용 할 수 있음을 의미합니다.

상위경계: upper bound 1

클래스를 정의하고 상속을 한다. 실제 처리하는 클래스 내의 메소드 내에 상위경계를 지정한다

```
scala> class A
defined class A

scala> class B extends A
defined class B

scala> class C extends B
defined class C

scala> class UP{
     |    def display[T<: B](t:T) {
     |      println(t)
     |    }
     | }
defined class UP
```

```
scala> val a = new A
a: A = A@77d17a16

scala> val b = new B
b: B = B@5d082237

scala> val c = new C
c: C = C@4e9c5bb0
```

상위경계: upper bound 2

상위 경계를 최상위 A 클래스로 지정하지 않아서 최상위 클래스를 처리하려면 예외가 발생한다.

```
scala> val up = new UP
up: UP = UP@34e10322

scala> up.display(a)
<console>:25: error: inferred type arguments [A] do not conform to me
y's type parameter bounds [T <: B]
       up.display(a)
          ^
<console>:25: error: type mismatch;
 found   : A
 required: T
       up.display(a)
                  ^

scala> up.display(b)
$line991.$read$$iw$$iw$$iw$$iw$$iw$$iw$B@5d082237

scala> up.display(c)
$line992.$read$$iw$$iw$$iw$$iw$$iw$$iw$C@4e9c5bb0
```

변수 할당에 bounded type 1

타입 매개변수에 타입 바운드로 지정할 경우는 실제 하위타입이 상위타입에 맞춰 변하지 않는다.

Bounded type 으로 지정하지만 변수 할당에는 공변성이 필요하다.

```
scala> class A
defined class A

scala> class B extends A
defined class B

scala> class Item[T <: A](a:T) {
     |    def get : T = a
     | }
defined class Item

scala> val c : Item[A] = new Item[B](new B)
<console>:14: error: type mismatch;
 found   : Item[B]
 required: Item[A]
Note: B <: A, but class Item is invariant in type T.
You may wish to define T as +T instead. (SLS 4.5)
       val c : Item[A] = new Item[B](new B)
                         ^
```

변수 할당에 bounded type 2

위 페이지이 오류를 해결하기 위해서는 타입 공변성 처리가 필요하다.

공변성을 지정해서 상속관계를 지정하고 특정 클래스로 바운드를 지정해서 처리하면 변수에 지정해서 처리

```
scala> class A
defined class A

scala> class B extends A
defined class B

scala> class Item[+T <: A](a:T) {
     |    def get : T = a
     | }
defined class Item

scala> val c : Item[A] = new Item[B](new B)
c: Item[A] = Item@41365d44

scala> c.get
res58: A = B@45302abb
```

함수 타입 매개변수를 특정화하기

제너릭 타입을 처리하기 보다 특정 타입을 지정해서 지정 범위 내에서 처리하기 위해 지정할 수 있다.

```
scala> def add[T <:Int](x:T, y:T) = x+y
add: [T <: Int](x: T, y: T)Int

scala> add(10.1,20)
<console>:24: error: inferred type arguments [Double] do not conform to method a
dd's type parameter bounds [T <: Int]
       add(10.1,20)
       ^
<console>:24: error: type mismatch;
 found   : Double(10.1)
 required: T
       add(10.1,20)
           ^
<console>:24: error: type mismatch;
 found   : Int(20)
 required: T
       add(10.1,20)
                ^

scala> add(10,10)
res367: Int = 20
```

메소드에 bounded type

메소드 내의 특정 연산자를 사용할 경우 그 매개변수를 특정 경계로 한정하면 실제 그 연산자를 사용할 수 있다.

```
scala> class ADD[T <: Int](val x:T) {
     |    def add(y: T) = x + y
     | }
defined class ADD

scala> val a = new ADD[Int](10)
a: ADD[Int] = ADD@373cd815

scala> a.add(100)
res57: Int = 110
```

함수에서 상위경계: upper bound

<: 를 이용해서 상위 클래스 경계를할 때 공변성을 사용하면 컴파일 오류가 발생한다.

상위경계는 invariant에서만 가능

```
scala> def add[+T <:Int](x:T) : T = x
<console>:1: error: ']' expected but identifier found.
       def add[+T <:Int](x:T) : T = x
               ^


scala> def add[T <:Int](x:T) : T = x
add: [T <: Int](x: T)T

scala> add(101)
<console>:21: error: inferred type arguments [Long] do not conform to met
's type parameter bounds [T <: Int]
       add(101)
       ^
<console>:21: error: type mismatch;
 found   : Long(10L)
 required: T
       add(101)
           ^


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

함수에서 상위경계에 상위 클래스 지정

<: 를 이용해서 상위 클래스 경계를 부여하면 하위 클래스에 대한 것을 처리가 가능하다.

상위 클래스를 지정하면 하위 클래스에 대한 처리도 가능하

```
scala> def add[T <: AnyVal](x:T):T = x
add: [T <: AnyVal](x: T)T

scala> add(100)
res372: Int = 100

scala> add(1001)
res373: Long = 100

scala> add(100f)
res374: Float = 100.0

scala> add(100d)
res375: Double = 100.0
```

하위 Bound 이해하기

하위경계

하위경계는 정해진 경계를 중심으로 상위 타입과 하위 타입을 처리한다.

하위경계 [T >: A] 로 표시 해당 A 타입과 그 상위 타입들을 포함해서 처리

Lower Bound (T >: Pet) : T는 Pet 클래스의 부모 클래스 중 적어도 하나를 상속받은 모든 클래스에 적용됨을 의미합니다.

하위경계 이해하기

>: 로 지정하면 하위경계가 형성되면 실제 하위경계가 구성된 부모 클래스로 구현된 모든 것이 해당하므로 숫자말고도 문자열도 처리된다.

```
scala> class A
defined class A

scala> class B extends A
defined class B

scala> class C extends B
defined class C

scala> class D extends C
defined class D

scala> class UP[T >: B] {
     |    def get(x:T) = println(x)
     | }
defined class UP
```

```
scala> up.get(new B)
$line96.$read$$iw$$iw$B@1bd357c6

scala> up.get(new C)
$line97.$read$$iw$$iw$C@2e9d759d

scala> up.get(new D)
$line98.$read$$iw$$iw$D@3a58332a

scala> up.get(new A)
<console>:14: error: type mismatch;
 found   : A
 required: B
       up.get(new A)
                  ^
```

하위경계 이해하기

>: 로 지정하면 하위경계가 형성되면 실제 하위경계가 구성된 부모 클래스로 구현된 모든 것이 해당하므로 숫자말고도 문자열도 처리된다.

```
scala> def add[T >: Int](c: T) = c
add: [T >: Int](c: T)T

scala> add(100)
res15: Int = 100

scala> add(1001)
res16: AnyVal = 100

scala> add("String")
res17: Any = String
```

하위경계: lower bound

>: 를 이용해서 하위 클래스 경계를 부여하면 상위 클래스에 대한 것을 처리가 가능하다. 일단 AnyVal로 지정하고 Long 자료형을 넣어서 처리할 수 있다.

```
scala> def add[T >: Int](x:T):T = x
add: [T >: Int](x: T)T

scala> add[AnyVal](1001)
res378: AnyVal = 100
```

하위경계: lower bound

>: 를 이용해서 하위 클래스 경계를 부여하면 상위 클래스에 대한 것을 처리가 가능하다. 일단 String으로 처리해도 해당 타입도 처리가 된다.

```
scala> func[AnyRef](List("1"))
res385: List[AnyRef] = List(1)

scala> def func[T >: String](x:List[T]) = x
func: [T >: String](x: List[T])List[T]

scala> func[AnyRef](List("1"))
res386: List[AnyRef] = List(1)

scala> func[String](List("1"))
res387: List[String] = List(1)
```

뷰 Bound 이해하기

뷰 경계

뷰 바운드 (view bound)는 어떤 타입 A를 마치 어떤 타입 T (실제 타입)처럼 사용할 수 있도록하는 Scala의 메커니즘입니다.

뷰 경계 [T <% A] 로 표시 어떤 타입을 다른 타입으로 "볼 수 있는지"를 지정한다

클래스에 타입매개변수 지정

뷰 경계는 타입 매개변수를 지정된 타입으로 인정하고 처리한다.

```
scala> class A[T <% Int] {
     |     def addIt(x : T) = 123 + x
     | }
defined class A
```

```
scala> val a = new A[Int]
a: A[Int] = A@76930861

scala> a.addIt(100)
res437: Int = 223

scala> val b = new A[String]
<console>:23: error: No implicit view available from String => Int.
       val b = new A[String]
                   ^
```

함수에 타입매개변수 지정

<% 를 이용해서 현재 타입 매개변수에 암묵적으로 클래스를 지정해서 처리

```
scala> def add[T <% Int](x:T,y:T) = x+y
add: [T](x: T, y: T)(implicit evidence$1: T => Int)Int

scala> add(10,10)
res4: Int = 20

scala> add(10l,10)
<console>:13: error: No implicit view available from Long => Int.
       add(10l,10)
           ^
```

컨텍스트 Bound 이해하기

컨텍스트 경계

컨텍트스 경계는 T : M으로 쓰여진다. M [T]에 대한 암묵적인 값의 존재를 필요로한다.

컨텍스트 경계 [T : A] 로 표시 해당 A 타입과 그 상위 타입들을 포함해서 처리

컨텍스트 바운드는 타입 파라미터와 타입 클래스 사이의 제약 조건이다
컨텍스트 바운드는 뷰 바인딩의 암시적 변환 대신 암시적 값을 설명합니다
context bound는 암시적인 값의 존재를 주장하는 방법이다

컨텍스트 바운드

하나의 클래스를 만들고 함수에 컨텍스트 바운드를 지정하고 이 내부에 메소드를 호출해서 처리하기

```
scala> class Echo[A](){
     |    def echo(a:A) = a.toString
     | }
defined class Echo

scala> implicit val intEcho = new Echo[Int]()
intEcho: Echo[Int] = Echo@5ae63432

scala> def g[A:Echo](a:A) = implicitly[Echo[A]].echo(a)
g: [A](a: A)(implicit evidence$1: Echo[A])String

scala> g(3)
res37: String = 3
```