Scala 클래스
일반 구조 알아보기

myjlms99@gmail.com

클래스와 인스턴스 구조

클래스 특징

클래스 내부에 정의된 속성과 메소드는 인스턴스를 위한 속성과 메소드이다.

클래스는 단일 상속만 지원하면 추가적인 것은 trait를 통해 상속해서 처리한다.

클래스는 기본 생성자는 매개변수에 var/val로 정의하고 이를 정의하지 않는 경우는 일반 매개변수로만 사용된다.

case class를 지원해서 클래스와 object가 바로 생성해서 자동으로 처리하는 방법을 제공한다.

클래스 기본 생성

아무 것도 하지 않는 클래스를 정의하고 new로 생성하면 클래스 내부의 함수 호출도 같이 실행되는 이유는 클래스 기본 생성자가 실행될 때 클래스 내부의 호출이 가능한 것을 전부 실행한다.

class 키워드 클래스명

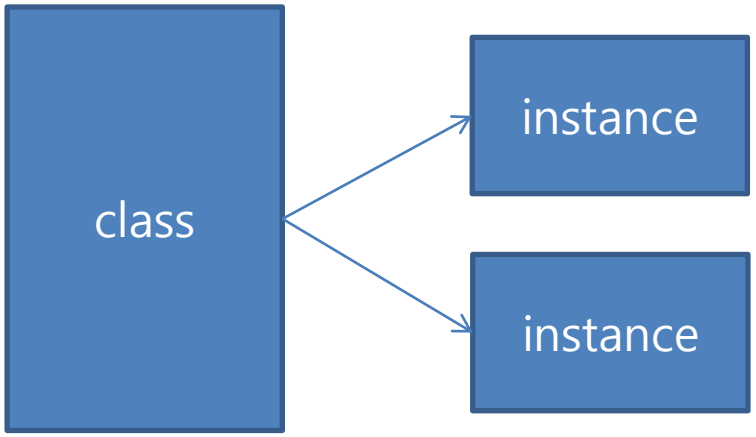
내부 구조는 { } 내부에 정의

인스턴스 생성은 new 를 이용

```
scala> class A {
     |   println(" A instance 생성 ")
     | }
defined class A

scala> new A
 A instance 생성
res50: A = A@4ab874e0
```

인스턴스 메모리 구조

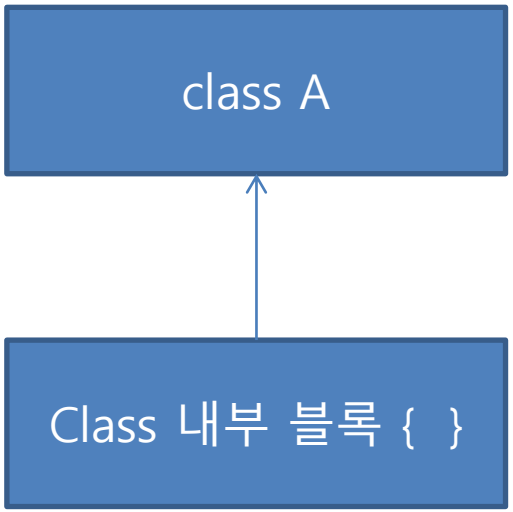
클래스를 가지고 인스턴스를 생성하면 실제 인스턴스는 var와 val로 정의된 속성만 가지고 있다. 메소드는 클래스를 참조한다.




```
scala> class A
defined class A

scala> val a = new A
a: A = A@4636e646

scala> a.getClass
res59: Class[_ <: A] = class A

scala> val b = new A
b: A = A@631eba76

scala> b.getClass
res60: Class[_ <: A] = class A

scala> a.getClass == b.getClass
res61: Boolean = true
```

클래스 구조

클래스는 class A와 블록으로 구성되면 new 로 인스턴스 생성할 때 블록 내부가 실행되어 인스턴스 내부를 만든다.



new A로 인스턴스 생성

```
scala> class A {
     |    var a :Int = 0
     |    val b : Int = 0
     |    println(" 초기화 실행")
     | }
defined class A

scala> val a = new A
 초기화 실행
a: A = A@24baa07b

scala> a.
a   b

scala> a.
```

익명 인스턴스 생성과 메소드 실행

인스턴스를 생성하고 변수에 할당하지 않고 이 인스턴스 내의 메소드를 직접 호출해서 처리한다.

```
scala> class An {
     |    def area(x:Int, y:Int) : Int = {
     |       x*y
     |   }
     | }
defined class An

scala> new An().area(10,10)
res1: Int = 100
```

속성 이해하기

멤버인 속성 정의

클래스 내부에 인스턴스 생성시 속성으로 만들 변수를 정의할 때 var와 val을 정의 가능하다. 이때 변경이 가능한 경우만 var를 사용

인스턴스를 만들면 내부에 속성이 2개 만들어져 있는 것을 알 수 있다

변경 불가능한 속성에 재할당을 하면 에러가 발생한다.

```
scala> class A {
     |   var x = 0
     |   val y = 0
     | }
defined class A

scala> val a = new A
a: A = A@5b93413

scala> a.
x   y

scala> a.x = 30
a.x: Int = 30

scala> a.x
res64: Int = 30

scala> a.y = 10
<console>:13: error: reassignment to val
       a.y = 10
           ^
```

접근 제어 (private)

클래스 내부의 멤버인 속성에 대해 외부 접근을 제어하기 위해 private으로 지정하고 메소드를 지정해서 조회하도록 구성

클래스 내부의 메소드에서만 private 속성 접근할 수 있다

```
scala> class A {
     |   private var x : Int = 0
     |   def getX = x
     | }
defined class A

scala> val a = new A
a: A = A@4963ff1c

scala> a.x
<console>:14: error: variable x in class A cannot be accessed in A
       a.x
         ^


scala> a.getX
res63: Int = 0
```

게터 메소드 작동생성

속성을 val로 정의하면 내부적으로 동일한 이름으로 게터 메소드가 만들어져 직접 접근하면 내부적으로 메소드 처리가 된다.

클래스 정의할 때 하나의 속성을 선언

내부적으로 동일한 이름으로 getter 메소드가 만들어져 처리한다

```
scala> class A {
     |     val a : Int = 0
     | }
defined class A

scala> val a = new A
a: A = A@ad16fe

scala> a.a
res65: Int = 0
```

게터와 세터 메소드 작동생성

속성을 var로 정의할 경우 내부에 동일한 이름으로 게터가 만들어지고 이름_= 로 세터가 만들어진다.

a_= 로 세터 메소드가 만들어져 있으므로 이를 호출해서 실제 속성 값을 변경

```
scala> class A {
     |    var a : Int = 0
     | }
defined class A

scala> val a = new A
a: A = A@490332a1

scala> a.a_=(30)

scala> a.a
res53: Int = 30
```

생성자 이해하기

생성자

생성자는 클래스 정의할 때 매개변수로 정의하면 주생성자가 만들어지고 보조생성자를 추가하면 보조생성자로도 인스턴스를 만든다.

```
Class 클래스명[ 주생성자 ] {
보조생성자 //def this로 정의
본문
}
```

속성없는 주생성자 정의

class 키워드와 클래스명만 처리하면 매개변수가 아무것도 없는 주 생성자가 만들어지고 new로 인스턴스를 실행하면 빈 인스턴스가 만들어진다.

인스턴스를 만들고 isInstanceOf 메소드를 이용해서 인스턴스 여부를 체크하면 true

```
scala> class A
defined class A

scala> val a = new A
a: A = A@2e29f28e

scala> a.isInstanceOf[A]
res0: Boolean = true
```

클래스 내의 매개변수

클래스 정의할 때 var와 val로 선언하지 않으면 내부 속성으로 사용되지 않는다

```
scala> class A(n:Int)
defined class A

scala> val a = new A(10)
a: A = A@7cf66580

scala> a.
!=   ->            ensuring   formatted   isInstanceOf   notifyAll      wait

##   ==            eq         getClass    ne             synchronized   →

+    asInstanceOf  equals     hashCode    notify         toString
```

클래스의 클래스 매개변수

클래스 내의 기본생성자는 클래스 이름 옆에 var와 val로 정의하면 new로 생성할 때 기본 생성자를 호출해서 처리된다.

```
scala> class A(var a:Int, val b:Int)
defined class A

scala> val a = new A
<console>:13: error: not enough arguments for constructor A: (a: Int, b: Int)A.
Unspecified value parameters a, b.
       val a = new A
               ^


scala> val a = new A(10,30)
a: A = A@a7bf458

scala> a.
a   b

scala> a.a_=(30)

scala> a.a
res56: Int = 30
```

클래스의 보조 생성자 정의

보조 생성자는 def this로 추가하면 속성이 여러 개일 경우 이를 인자로 받고 내부에서 상위의 생성자를 호출후에 this.멤버 정보를 갱신하면 처리가 된다.

Class 명+ 기본생성자

Def this로 보조생성자 내부에 this로 기본생성자 호출

```
scala> class A(var name: String) {
     |     var age : Int = 0
     |     def this(name : String, age:Int) {
     |         this(name)
     |         this.age = age
     |     }
     | }
defined class A

scala> val a = new A("Dahl")
a: A = A@5a24afb

scala> val b = new A("Moon", 55)
b: A = A@516a69de

scala> a.
age   name

scala> b.
age   name
```

생성자에 매개변수만 이용

클래스 기본 생성자와 보조 생성자에 매개변수로 처리만 할 경우 실제 인스턴스 속성이 생기지 않는다.

```
scala> class S(name:String) {
     |   def this(name:String, age:Int) {
     |       this(name)
     |       println(name + " " + age)
     |    }
     | }
defined class S

scala> val s = new S("This")
s: S = S@79c30ecc

scala> s.name
<console>:14: error: value name is not a member of S
       s.name
         ^


scala> val s1 = new S("That", 33)
That 33
s1: S = S@38922953

scala> s1.name
<console>:14: error: value name is not a member of S
       s1.name
          ^
```

this 키워드

this 키워드는 생성자 및 실제 인스턴스 변수를 접근할 때 사용된다.

보조생성자 내에 this()는 기본생성자를 호출했고 이 인스턴스에 속성에 값을 재할당하는 것을 볼 수 있다.

```
scala> class This {
     | var id : Int = 0
     | var name : String = ""
     |
     | def this(id:Int, name: String) {
     |    this()
     |    this.id = id
     |    this.name = name
     | }
     |
     | def show() {
     |    println(id + " " + name)
     | }
     | }
defined class This

scala> val t = new This
t: This = This@442c57d8

scala> t.show
0

scala> val t2 = new This(100, "Dahl")
t2: This = This@74d3211c

scala> t2.show
100 Dahl
```

메소드 이해하기

메소드 정의 1

인스턴스에서 행위 즉 기능에 맞는 메소드를 만든다.

def 메소드명(매개변수) :
반환값 = { 로직 }

```
scala> class A {
     |    var a : Int = 0
     |    var b : Int = 0
     |    def area : Int = a *b
     |    def set(x:Int, y:Int) = {
     |       a = x
     |       b = y
     |  }
     | }
defined class A

scala> val a = new A
a: A = A@6b5ee4b3

scala> a.set(10,10)

scala> a.area
res67: Int = 100
```

메소드 접근제어 1

메소드도 외부에 공개되지 않도록 private 을 지정하면 접근할 수 없다.

```
scala> class AA {
     |    private var x : Int = 0
     |    private def square = x* x
     |    def getX = x
     |    def setX(a : Int) = x = a
     | }
defined class AA

scala> val aa = new AA
aa: AA = AA@62783069

scala> aa.setX(10)

scala> aa.getX
res85: Int = 10

scala> aa.s
setX   synchronized
```

인스턴스에서 접근 가능한 메소드에 square가 존재하지 않는다

메소드 접근제어 2

메소드에 접근제어를 붙이면 내부 메소드에서 호출만 가능하다.

square메소드에 접근제어를 처리하면 내부 메소드에서만 접근

```
scala> class AA {
     |   private var x : Int = 0
     |   private def square = x*x
     |   def getX = x
     |   def setX(a:Int) = x=a
     |   def calc = square
     | }
defined class AA

scala> val aa = new AA
aa: AA = AA@405731f8

scala> aa.setX(10)

scala> aa.get
getClass   getX

scala> aa.getX
res88: Int = 10

scala> aa.calc
res89: Int = 100
```

calc 메소드를 호출하면 내부적으로 square 메소드가 실행

오버로딩 알아보기

메소드 오버로딩 1

매개변수의 개수를 다르게 정의해서 메소드 오버로딩 처리하기

set 메소드의 매개변수가 개수를 달리해서 메소드 재정의

```
scala> class A {
     |   var a : Int = 0
     |   var b : Int = 0
     |   def area : Int = a*b
     |   def set() = {a=0; b=0}
     |   def set(x:Int) = a =x
     |   def set(x:Int, y:Int) = { a=x; b=y}
     | }
defined class A

scala> val a = new A
a: A = A@289d89a7

scala> a.set(10,10)

scala> a.area
res70: Int = 100

scala> a.set(30)

scala> a.area
res72: Int = 300
```

메소드 오버로딩 2

동일한 매개변수를 가질 경우도 실제 데이터 타입에 따라 메소드가 오버로딩이 가능하다.

시그너처는 실제 매개변수의 자료형을 기준으로 처리

```
scala> class A {
     |    def area(x:Int, y:Int) = x*y
     |    def area(x:Double, y:Double) = x*y
     | }
defined class A

scala> val a = new A
a: A = A@6e231794

scala> a.area(10,10)
res74: Int = 100

scala> a.area(10.1,10.1)
res75: Double = 102.00999999999999
```

Class 내의 정보 확인하기

인스턴스의 클래스 정보확인

인스턴스를 만들고 getClass 메소드를 이용해서 클래스 정보를 확인한다.

```
scala> class A {
     |     var x = 0
     |     def getX = x
     | }
defined class A

scala> val a = new A
a: A = A@1c63aa5d

scala> a.getClass
res5: Class[_ <: A] = class A
```

Class 내부 속성 및 메소드

getClass. 을 작성하고 키보드의 <<tab>> 을 누르면 내부의 속성과 메소드가 나온다.

```
scala> a.getClass.
asSubclass                      getDeclaredFields           getSigners
cast                            getDeclaredMethod           getSimpleName
desiredAssertionStatus          getDeclaredMethods          getSuperclass
getAnnotatedInterfaces          getDeclaringClass           getTypeName
getAnnotatedSuperclass          getEnclosingClass           getTypeParameters
getAnnotation                   getEnclosingConstructor     isAnnotation
getAnnotations                  getEnclosingMethod          isAnnotationPresent
getAnnotationsByType            getEnumConstants            isAnonymousClass
getCanonicalName                getField                    isArray
getClassLoader                  getFields                   isAssignableFrom
getClasses                      getGenericInterfaces        isEnum
getComponentType                getGenericSuperclass        isInstance
getConstructor                  getInterfaces               isInterface
getConstructors                 getMethod                   isLocalClass
getDeclaredAnnotation           getMethods                  isMemberClass
getDeclaredAnnotations          getModifiers                isPrimitive
getDeclaredAnnotationsByType    getName                     isSynthetic
getDeclaredClasses              getPackage                  newInstance
getDeclaredConstructor          getProtectionDomain         toGenericString
getDeclaredConstructors         getResource                 toString
getDeclaredField                getResourceAsStream
```

Class 내부의 기본 정보 확인

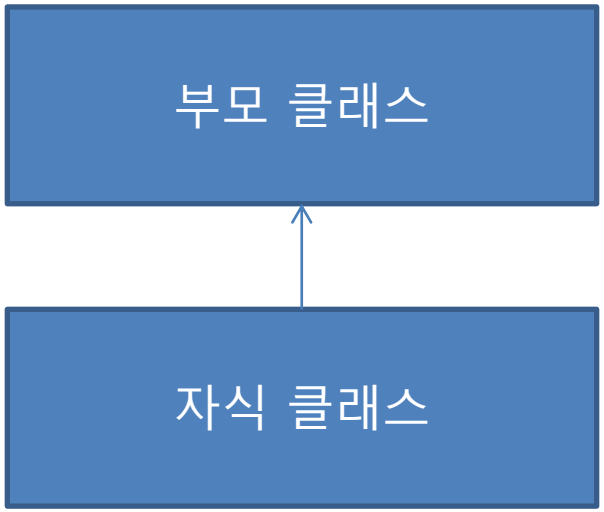
생성된 인스턴스 내에서 클래스에 정의된 속성과 메소드를 확인한다.

```
scala> a.getClass.getName
res15: String = A

scala> a.getClass.getDeclaredConstructors
res16: Array[java.lang.reflect.Constructor[_]] = Array(public A())

scala> a.getClass.getDeclaredFields
res17: Array[java.lang.reflect.Field] = Array(private int A.x)

scala> a.getClass.getDeclaredMethods
res18: Array[java.lang.reflect.Method] = Array(public int A.x(), public void A.x
_$eq(int), public int A.getX())
```

Class 상속하기

상속

class 를 직접 상속해서 자기 class의 인스턴스가 사용할 수 있는 체계를 만드는 것



부모 클래스 상속 : 속성

부모 클래스에 속성이 있고 자식 클래스에는 아무것도 없다. 상속하면 부모 클래스의 것을 사용할 수 있다.

부모 클래스를 extends로 상속

부모 클래스의 속성을 검색해서 결과를 가져온다.

```
scala> class AA {
     |    val a : String = "A attribute "
     | }
defined class AA

scala> class BB extends AA
defined class BB

scala> val bb = new BB
bb: BB = BB@11096418

scala> bb.a
res113: String = "A attribute "
```

부모 클래스 상속: 메소드

부모 클래스에 속성과 메소드가 존재하고 자식 클래스에는 아무것도 없어도 부모클래스의 멤버를 사용

```
scala> class AA {
     |    val a : String = " AA 속성 "
     |    def getA = a
     | }
defined class AA

scala> class BB extends AA {
     |   }
defined class BB

scala> val bb = new BB
bb: BB = BB@18cec008

scala> bb.getA
res114: String = " AA 속성 "
```

변수에 부모클래스로 타입정의

부모 클래스와 자식 클래스가 상속 관계이므로 부모클래스로 지정된 변수에도 자식클래스로 인스턴스 정의가 가능

자식클래스로 변수 정의할 때 부모 클래스로 객체 생성하면 에러가 발생

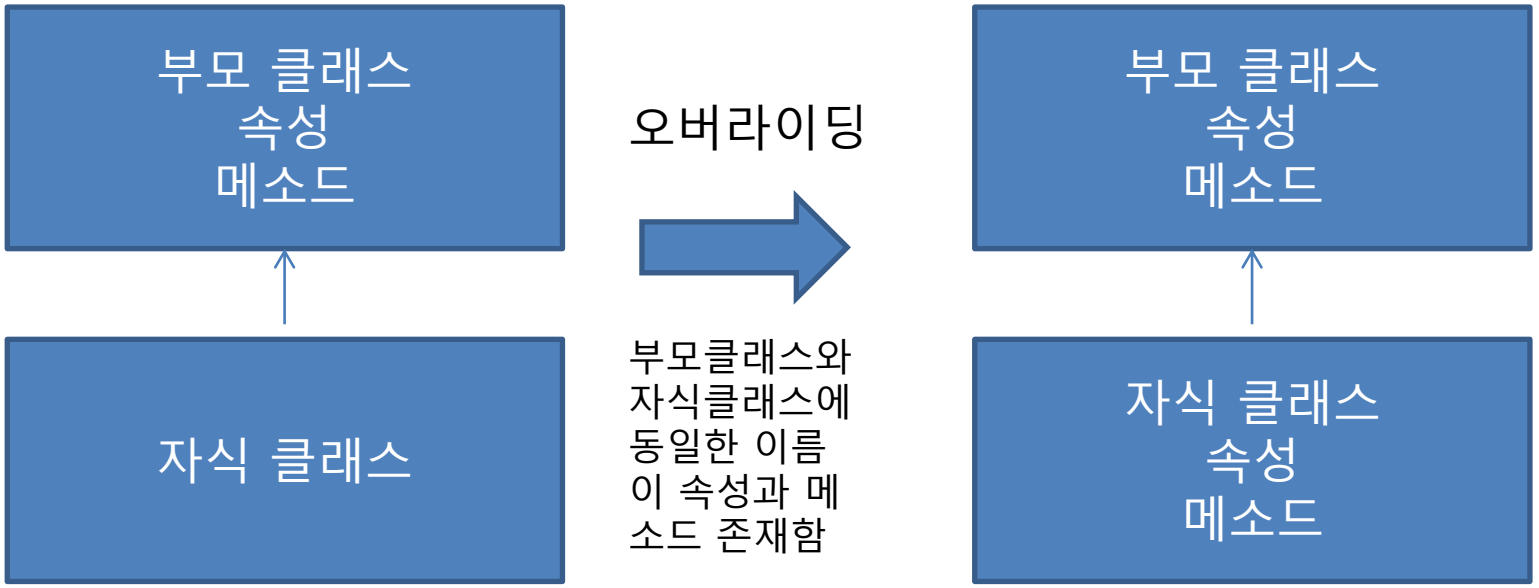
```
scala> class AA {
     |    val a : String = " AA 속성 "
     |    def getA = a
     | }
defined class AA

scala> class BB extends AA {
     |   }
defined class BB

scala> val bb = new BB
bb: BB = BB@18cec008

scala> bb.getA
res114: String = " AA 속성 "

scala> val cc : AA = new BB
cc: AA = BB@25427ff6

scala> cc.getClass
res115: Class[_ <: AA] = class BB

scala> bb.getClass
res116: Class[_ <: BB] = class BB
```

```
scala> val dd : BB = new AA
<console>:14: error: type mismatch;
 found   : AA
 required: BB
       val dd : BB = new AA
                     ^
```

오버라이딩 알아보기

오버라이딩이란

상속이 발생할 경우 부모 클래스의 속성과 메소드도 자식 클래스에서 사용이 가능하지만 자식클래스만의 특징을 반영할 때 동일한 속성과 메소드를 재정의해서 사용



변수 오버라이딩

상속을 할 경우 부모 클래스에 있는 속성(val로 정의)을 자기 클래스에 필요한 부분을 변경할 때 발생한다.

부모 클래스의 val 속성을 자식 클래스에서 재정의해서 사용

```
scala> class U {
     |    val speed : Int = 100
     | }
defined class U

scala> class B extends U {
     |    override val speed : Int = 30
     | }
defined class B

scala> val b = new B
b: B = B@5a4be77f

scala> b.speed
res20: Int = 30

scala> b.getClass
res21: Class[_ <: B] = class B
```

변수 오버라이딩 에러 1

부모 클래스에 지정된 var 속성은 자식클래스에서 재정의할 때 override 하라고 나오지만 실제 override가 안됨

```
scala> class V {
     |    var name = "V class"
     |    def getName = name
     | }
defined class V

scala> class B extends V {
     |    var name = "B class"
     | }
<console>:13: error: overriding variable name in class V of type String;
 variable name needs `override' modifier
           var name = "B class"
               ^
```

변수 오버라이딩 에러 2

부모 클래스에 지정된 val 속성을 자식클래스에서 재정의할 때 var로 override를 하면 에러가 발생한다

```
scala> class U {
     |    val speed = 100
     | }
defined class U

scala> class B extends U {
     |    override var speed = 60
     | }
<console>:13: error: overriding value speed in class U of type Int;
 variable speed needs to be a stable, immutable value
           override var speed = 60
                        ^
```

메소드 오버라이딩

상속을 할 경우 부모 클래스에 있는 메소드를 자기 클래스에 필요한 부분을 변경할 때 발생한다.

부모클래스와 자식클래스에 동일한 메소드 정의되어 호출할 때 자식 클래스의 메소드 부터 실행

```
scala> class U {
     |    def run {
     |       println(" U running")
     |    }
     | }
defined class U

scala> class B extends U {
     |    override def run {
     |       println(" B running")
     |    }
     | }
defined class B

scala> val b = new B
b: B = B@7032e80e

scala> b.run
 B running
```

Final 키워드 알아보기

final 키워드 사용

부모 클래스를 상속하면 자식클래스는 부모 클래스의 모든 것을 재정의가 가능하다 이를 차단해서 원래의 것만을 사용이 필요할 때 이용한다.

클래스 내의 특정 속성, 메소드를 더 이상 확장하지 않고 고정해서 사용이 필요할 경우

클래스가 더 이상 상속이 되지 않도록 막을 경우

final val 정의

부모 클래스에 final val로 지정하면 그 변수는 더 이상 자식클래스에서 오버라이딩이 불가하다.

부모 클래스에서 고정되는 변수는 final로 지정해서 상속해도 다시 오버라이딩을 할 수 없도록 지정이 필요

```
scala> class A {
     |   final val a = 1
     | }
defined class A

scala> class B extends A {
     |   override val a = 2
     | }
<console>:13: error: type mismatch;
 found   : Int(2)
 required: Int(1)
         override val a = 2
                          ^
```

final def 정의

부모 클래스에 final def로 지정하면 그 메소드는 더 이상 자식클래스에서 오버라이딩이 불가하다.

```
scala> class A {
     | final def x = 1
     | def y = 2
     | }
defined class A

scala> class B extends A {
     |   override def x = 3
     |   override def y = 4
     | }
<console>:13: error: overriding method x in class A of type => Int;
 method x cannot override final member
         override def x = 3
                      ^
```

final class 정의

final 클래스는 더 이상 상속을 할 수 없는 클래스를 만들었으므로 이를 상속해서 처리할 수 없는 예러가 발생한다.

```
scala> final class A {
     |    def x = 1
     |    def y = 2
     | }
defined class A

scala> class B extends A {
     |    override def x =3
     |    override def y = 4
     | }
<console>:12: error: illegal inheritance from final class A
       class B extends A {
                       ^
```

Object 이해하기

오브젝트 특징

오브젝트는 클래스와 인스턴스가 하나인 싱글턴을 만든다.

클래스의 정적 속성과 메소드가 없으므로 오브젝트를 이용해서 인스턴스를 만들지 않고 실행이 가능하다.

별도의 메인함수를 지정해서 실제 실행하는 입구로 만들 수 있다.

싱글턴이므로 기본생성자나 매개변수를 지정할 수 없다.

싱글턴 객체 생성

Object를 이용해서 생성하면 하나의 인스턴스를 가진 싱글턴 인스턴스가 만들어진다.

내부에 정의된 부분을 인스턴스로 직접 사용하는 것과 동일

```
scala> object AB {
     |     var x : Int = 0
     |     def getX = x
     | }
defined object AB

scala> AB.x
res78: Int = 0

scala> AB.x = 100
AB.x: Int = 100

scala> AB.getX
res79: Int = 100
```

컴패니언 객체

클래스를 접근해서 정적 속성과 메소드를 처리하기 위해서는 class와 object를 가지 정의하면 실제 클래스 정적 처리처럼 실행

Class와 object를 동시 정의

Class 명으로 접근해서 처리

인스턴스에서는 정적 처리 불가

```
class A

object A {
   var x : Int = 0
   def setX(a : Int) = x =a
}

// Exiting paste mode, now interpreting.

defined class A
defined object A

scala> A.x
res80: Int = 0

scala> A.setX(10)

scala> A.x
res82: Int = 10

scala> val a = new A
a: A = A@1dd67363

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

컴패니언 처리 기준

class와 object를 두개를 지정해서 컴패니언 구조를 만들어지면 실제 이를 가지고 new와 apply로 객체를 생성할 수 있다.

Class와 object를 동시 정의

두 가지 방식으로 객체 생성

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

class ABCD(val name: String)
object ABCD {
   def apply(name:String) = new ABCD(name)
}

// Exiting paste mode, now interpreting.

defined class ABCD
defined object ABCD

scala> new ABCD("Moon")
res96: ABCD = ABCD@3d0496cd

scala> ABCD("Dahl")
res97: ABCD = ABCD@6e9fffdc
```

추상클래스 상속하기

추상클래스를 object에서 상속하고 추상 속성을 재정의해서 apply로 처리

```
abstract class ABC {
   val x : Int
}
object ABC extends ABC {
   val x : Int = 100
   def apply() = x
}

// Exiting paste mode, now interpreting.

defined class ABC
defined object ABC

scala> ABC()
res98: Int = 100
```

private, protected
접근제어자 이해

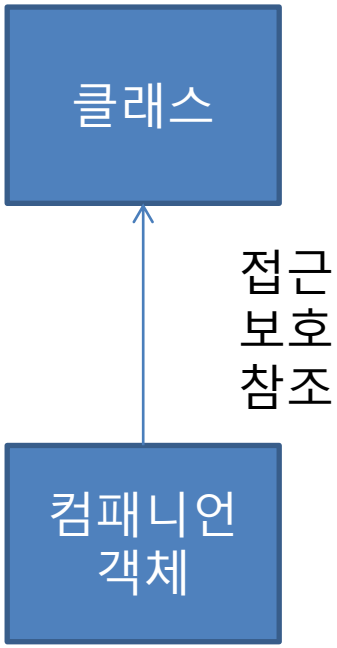
접근제어자

protected, private을 사용할 경우 실제 클래스간의 참조 범위는 아래와 같다. 특히 private일 경우 하위 클래스는 참조가 불가하지만 컴패니언은 사용이 가능한 것을 명확히 이해해야 한다.

Modifier	Outside package	Package	Class	Subclass	Companion
No access modifier	Yes	Yes	Yes	Yes	Yes
Protected	No	No	Yes	Yes	Yes
Private	No	No	Yes	No	Yes

주 생성자에 접근제어 처리

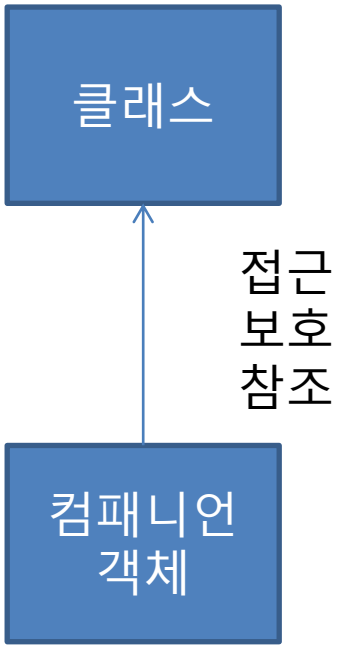
주 생성자에 private으로 처리하면 이를 object를 정의해서 그 내부의 apply 메소드를 이용해서 처리할 수 있다.




```
scala> :paste
// Entering paste mode (ctrl-D to finish)

class A private(var name:String)

object A {
   def apply(name : String) = new A(name)
}

// Exiting paste mode, now interpreting.

defined class A
defined object A

scala> val a = new A("Dahl")
<console>:14: error: constructor A in class A cannot be accessed in object $iw
       val a = new A("Dahl")
                   ^


scala> val b = A("Dahl")
b: A = A@55e5be0a

scala> b.name
res76: String = Dahl
```

컴패니언 클래스 private 속성

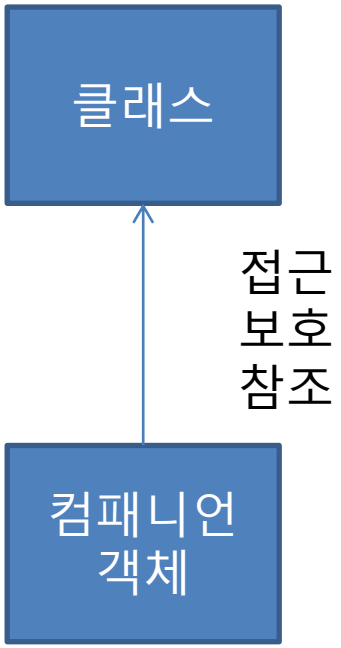
컴패니언 클래스 내부의 private 속성에 대해 처리할 때 인스턴스를 만들고 변수를 참조하면 처리가 됨




```
scala> :paste
// Entering paste mode (ctrl-D to finish)

class A {
   private val x = "Dahl"
}
object A {
   def apply() = new A().x
}

// Exiting paste mode, now interpreting.

defined class A
defined object A

scala> A()
res90: String = Dahl
```

컴패니언 클래스 private 메소드

컴패니언 클래스 내부의 private 속성과 메소드를 정의하고 이 메소드를 호출해서 처리




```
scala> :paste
// Entering paste mode (ctrl-D to finish)

class ABC private (val name: String) {
   private def getName = name
}
object ABC {
   def apply() = {
      new ABC("Moon").getName
   }
}

// Exiting paste mode, now interpreting.

defined class ABC
defined object ABC

scala> ABC()
res91: String = Moon
```

case class 이해하기

case class 구조

class와 object를 동시에 자동으로 만들어 준다.

case class 클래스명(매개변수) extends 클래스명(매개변수
{ 필드; 메소드 }

apply	객체	클래스의 인스턴스를 생성하기 위한 팩토리 메소드
unapply	객체	인스턴스를 그 인스턴스의 필드의 튜플로 추출해서 패턴매칭에 사용
equals	클래스	두개의 인스턴스를 비교해서 일치여부를 확인
hashcode	클래스	인스턴스의 필드들의 해시코드를 반환
toString	클래스	클래스명과 필드들을 String으로 전환
copy	클래스	변경사항을 받아서 새로운 인스턴스 사본을 만들어 준다

case class 정의: 타입 매개변수

case class는 반드시 매개변수를 지정해서 정의해야 한다.

```
scala> case class A
<console>:1: error: case classes must have a parameter list; try 'case class A()
' or 'case object A'
       case class A
                   ^
```

매개변수를 지정해서 처리

```
scala> case class A(name : String)
defined class A

scala> A("Dahl")
res103: A = A(Dahl)
```

case class 정의: 클래스 매개변수

case class는 반드시 매개변수를 지정할 때 val과 var를 사용하면 인스턴스의 속성으로 세팅된다.

```
scala> case class A(val name : String)
defined class A

scala> val a = A("Moon")
a: A = A(Moon)

scala> a.name
res104: String = Moon
```

case class 정의: 패턴 매칭

case class로 정의하면 unapply가 지정되므로 실제 패턴매칭에서 object apply를 호출하면 기존 만든 인스턴스를 가지고 실제 값들을 처리한다.

```
scala> case class Matching(name: String,isBool: Boolean)
defined class Matching

scala> val m = Matching("Dahl", true)
m: Matching = Matching(Dahl,true)

scala> m match {
     |    case Matching(x,true) => println("true " + x)
     |    case Matching(x,flase) => println("false " + x)
     | }
true Dahl
```

추상 Class 상속하기

추상클래스는 인스턴스 생성 불가

추상 클래스는 구현된 클래스가 아니므로 새로운 인스턴스를 생성할 수 없다.

Scala 언어의 Int는 추상클래스이다. 상속을 해서 구현 클래스만 만들 수 있다.

```
scala> val a = new Int
<console>:13: error: class Int is abstract; cannot be instantiated
       val a = new Int
               ^


scala> val a = 1
a: Int = 1

scala> a.getClass
res29: Class[Int] = int
```

추상클래스 메소드 처리

추상클래스를 상속하면 명확히 내부에 있는 메소드를 전부 구체 메소드로 구현을 해야 한다. 구현하지 않으면 에러 발생한다.

추상클래스를 상속해서 추상메소드를 반드시 구현해야 한다

```
scala> abstract class Abc {
     |    def x
     | }
defined class Abc

scala> class A extends Abc {
     |    def y = 2
     | }
<console>:12: error: class A needs to be abstract, since method x in class Abc o
f type => Unit is not defined
       class A extends Abc {
             ^
```

부모 추상클래스 상속: 메소드

부모 추상클래스에 메소드가 존재하고 자식 클래스에는 이를 구현해야 해서 인스턴스를 생성해서 실행시킨다.

추상클래스와 구현클래스 정의

```
scala> abstract class Abstract {
     |    def getX :Int
     | }
defined class Abstract

scala> class Concrte(var x:Int) extends Abstract {
     |    def getX = x
     | }
defined class Concrte
```

구현 클래스로 인스턴스 생성 및 메소드 호출

```
scala> val c = new Concrte(10)
c: Concrte = Concrte@6da44b78

scala> c.getX
res117: Int = 10
```

Trait 이해하기

트레이트 특징

속성과 메소드를 추상과 구현으로 정의가 가능하다.

트레이트 및 클래스를 상속해서 정의할 수 있다.

구현메소드만 제공해서 실제 클래스가 상속 받을 때 Mixin을 처리할 수 있다.

특정 메소드를 정의하고 다양한 클래스에서 상속해서 구현할 수 있으므로 다양한 클래스의 메소드 호출을 할 수 있는 duck typing 지원한다.

trait 정의할 때 유의점

trait를 정의할 때는 abstract와 final 키워드를 사용할 수 없다.

```
scala> final trait A {
     |    def x = 1
     | }
<console>:11: error: illegal combination of modifiers: abstract and final for: t
rait A
       final trait A {
                   ^
```

trait 정의 후 접근

trait를 정의하고 실제 내부 속성을 접근하면 발견할 수 없다.

```
scala> trait Atrait {
     |    val a : String = "Moon"
     | }
defined trait Atrait

scala> Atrait.a
<console>:13: error: not found: value Atrait
       Atrait.a
       ^
```

trait 정의 시 매개변수 지정

trait를 정의할 때 매개변수를 지정하면 필요 없다는 에러가 발생한다.

```
scala> trait Btrait(name:String) {
<console>:1: error: traits or objects may not have parameters
       trait Btrait(name:String) {
                    ^
```

trait 를 클래스 정의시 상속

trait를 정의할 때 내부 구현 속성을 정의하면 이 trait를 class에서 상속하면 인스턴스에서 직접 호출해서 사용할 수 있다.

```
scala> trait Atrait {
     |    val a : String = "Moon"
     | }
defined trait Atrait

scala> class At extends Atrait
defined class At

scala> val a = new At
a: At = At@636b3049

scala> a.a
res109: String = Moon

scala> At.a
<console>:13: error: not found: value At
       At.a
       ^
```

트레이트의 정보는 클래스에서 접근할 수 없다

trait를 인스턴스 생성시 사용

클래스 정의할 때 상속을 하지 않고 인스턴스 생성할 때 직접 사용하기 위해 with로 연결해서도 사용이 가능하다. 익명클래스가 만들어지는 것을 볼 수 있다.

```
scala> trait Btrait {
     |    val a : String = "Dahl"
     | }
defined trait Btrait

scala> class B
defined class B

scala> val b = new B with Btrait
b: B with Btrait = $anon$1@5220d693

scala> b.a
res111: String = Dahl

scala> val c = new B
c: B = B@6cb65024

scala> b.getClass == c.getClass
res112: Boolean = false
```

클래스를 비교해보면 다른 클래스인 것을 확인할 수 있다.

trait 용도

추상 클래스 대용

Trait 내부에 함수 타입만 정의하면 추상 메소드로 인식하므로 클래스에서 이를 구현해야 함

```
scala> trait ABC {
     |    def getname :String
     | }
defined trait ABC

scala> class Person(val name:String) extends ABC {
     |    def getname = name
     | }
defined class Person

scala> val p = new Person("Dahl")
p: Person = Person@7001e12f

scala> p.getname
res8: String = Dahl
```

제너릭 메소드: 추상

Trait 내부 메소드의 매개변수를 제너릭으로 처리할 경우 실제 구현에서 match를 이용해서 타입 체크

```
scala> trait Sim {
     |  def isSim(x:Any) : Boolean
     | }
defined trait Sim

scala> class Point(xc: Int, yc:Int) extends Sim {
     |   var x = xc
     |   var y = yc
     |   def isSim(obj:Any) = {
     |     obj match {
     |       case x:Point => true
     |       case _ => false
     | }
     | }
     | }
defined class Point

scala> val p = new Point(10,10)
p: Point = Point@537f2782

scala> p.isSim(p)
res14: Boolean = true
```

특정 클래스로 한정처리

Trait을 클래스의 메소드를 오버라이딩하고 이 클래스를 생성할 때 with 구문으로 재사용하면 실제 익명클래스가 만들어져 처리가 된다. 이 클래스 타입이 아닌 경우에는 예외가 발생한다.

```
scala> trait GetName extends Person {
     |    override def getname = "GetName" + super.getname
     | }
defined trait GetName

scala> val v = new Person("Dahl") with GetName
v: Person with GetName = $anon$1@224361de

scala> v.getname
res9: String = GetNameDahl

scala> class People
defined class People

scala> var x = new People with GetName
<console>:13: error: illegal inheritance; superclass People
 is not a subclass of the superclass Person
 of the mixin trait GetName
       var x = new People with GetName
                               ^
```

익명 클래스 처리

추상클래스를 지정해서 새로운 익명클래스를 정의할 때 실제 내부적으로는 warning이 발생한다.

```
scala> abstract class Foo {
     | def saybye = "Bye"
     | }
defined class Foo

scala> val hello = new Foo {
     | def sayhello = "hello"
     | }
hello: Foo{def sayhello: String} = $anon$1@20b23903

scala> hello.sayhello
<console>:13: warning: reflective access of structural type member method sayhe
lo should be enabled
by making the implicit value scala.language.reflectiveCalls visible.
This can be achieved by adding the import clause 'import scala.language.reflect
veCalls'
or by setting the compiler option -language:reflectiveCalls.
See the Scaladoc for value scala.language.reflectiveCalls for a discussion
why the feature should be explicitly enabled.
       hello.sayhello
             ^

res11: String = hello
```

익명 클래스 처리: 경고 없애기

Import scala.language.reflectiveCalls를 먼저 선언하고 익명클래스를 만들면 실제 경고가 사라진다.

```
scala> improt scala.language.reflectiveCalls
<console>:1: error: ';' expected but '.' found.
       improt scala.language.reflectiveCalls
                    ^


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

scala> abstract class Foo {
     |   def saybye = "Bye"
     | }
defined class Foo

scala> val x = new Foo { def sayhello = "Hello" }
x: Foo{def sayhello: String} = $anon$1@413b45bc

scala> x.sayhello
res13: String = Hello
```

sealed 이해

동일한 파일 내로 상속 제한하기

class나 trait를 특정 파일 내에서만 상속을 하기 위해서 제한을 할 경우 sealed로 봉인해서 사용해야 한다. Option 클래스는 sealed된 클래스이므로 상속을 받아 처리하면 상속에 되한 오류가 발생한다.

```
scala> sealed class A
defined class A

scala> class B extends A
defined class B

scala> sealed trait AT
defined trait AT

scala> class C extends AT
defined class C

scala> class d extends Opt
OptManifest   Option

scala> class d extends Option
<console>:11: error: illegal inheritance from sealed class Option
       class d extends Option
                       ^
```

implicit 이해하기

암시적 처리 규칙

정의된 것을 내부적으로 규칙화 되므로 implicit 키워드 정의된 것을 처리한다.

표시규칙 : implicit 로 표시한 정의만 검토 대상이 된다

스코프 규칙 : 삽입할 implict 변환은 스코프 내에 단일 식별자로만 존재하거나, 변환의 결과나 원래 타입과 연관 필요

한번에 하나만 규칙 : 오직 하나의 암시적 선언만 사용

명시성 우선 규칙: 코드가 그 상태 그대로 타입 검사를 통과 한다면 암시를 통한 변환을 시도치 않음

명시적 처리 클래스 정의

클래스를 명시적을 주고 인스턴스를 생성해서 메소드를 호출해서 처리

```
scala> class Awe1(val s:String) extends AnyVal {
     |    def awesome_ = s+ " Awesme "
     | }
defined class Awe1
```

```
scala> val a = new Awe1("SSSSSS")
a: Awe1 = Awe1@925ad720

scala> a.awesome_
res28: String = "SSSSSS Awesme "
```

암묵적 처리 클래스 정의

실제 암묵적 처리는 내부에서 자동으로 이 클래스 내의 메소드를 호출해서 처리하도록 자동으로 연결해 준다.

```
scala> implicit class Awe(val s:String) extends AnyVal {
     |  def awesome_! = s + "! Awesone !"
     | }
defined class Awe
```

```
scala> "$$$$".awesome_!
res27: String = $$$$! Awesone !
```