Scala
self type & inheritance

myjlms99@gmail.com

명기적 상속과 구성

Inheritance

상속은 수퍼클래스 내의 멤버를 상속을 받아서 재사용하거나 재정의해서 사용할 수 있다.

```
scala> class Vehicle {
     |    def fuelType = "diesel"
     | }
defined class Vehicle

scala> class Car extends Vehicle {
     |    override def fuelType = super.fuelType
     | }
defined class Car

scala> val c = new Car
c: Car = Car@29601d11

scala> c.fuelType
res30: String = diesel
```

Composition:

상속 관계로 인해 수퍼 클래스의 인터페이스를 변경하기 어렵다. 컴포지션을 사용하면 코드 재사용을 위한 변경 쉬운 코드를 생성하는 방식을 제공

```
scala> class Vehicle {
     |    def fuleType = "diesel"
     | }
defined class Vehicle

scala> class Car {
     |    val vehicle = new Vehicle
     |    def fuleType = vehicle.fuleType
     | }
defined class Car

scala> val c = new Car
c: Car = Car@39ab14bb

scala> c.fuleType
res29: String = diesel
```

Inheritance

상속은 수퍼클래스 내의 멤버를 상속을 받아서 재사용하거나 재정의해서 사용할 수 있다.

```
scala> class Vehicle {
     |    def fuelType = "diesel"
     | }
defined class Vehicle

scala> class Car extends Vehicle {
     |    override def fuelType = super.fuelType
     | }
defined class Car

scala> val c = new Car
c: Car = Car@29601d11

scala> c.fuelType
res30: String = diesel
```

자기 참조(self reference)

Self type으로 별칭 사용하기

self=>로만 정의한 경우는 실제 this 별칭으로만 사용된다.

self => 삽입은 this 키워드 대신 다른 이름인 self를 사용하기

Self type으로 별칭 사용하기

self=>로만 정의한 경우는 실제 this 별칭으로만 사용된다.

```
scala> trait Self {
     |    self =>
     |    val name = "outer"
     |    def getName = self.name
     |    def thisName = this.name
     | }
defined trait Self

scala> class A extends Self
defined class A

scala> val a = new A
a: A = A@22691b34

scala> a.getName
res33: String = outer

scala> a.thisName
res34: String = outer
```

```
scala> class Self {
     |    self =>
     |    val name = "outer"
     |    def getName = self.name
     |    def thisName = this.name
     | }
defined class Self

scala> class S extends Self
defined class S

scala> val s = new S
s: S = S@3dff5340

scala> s.getName
res35: String = outer

scala> s.thisName
res36: String = outer
```

Self-type annotations

클래스 내부에 클래스를 추가해서 지정할 경우 상위 클래스와 동일한 이름이 멤버가 있는 경우 이를 구별하기 표시한다.

자체 유형으로 사용하기

```
class C1 { self =>
  def talk(message: String) = println("C1.talk: " + message)
  class C2 {
    class C3 {
      def talk(message: String) = self.talk("C3.talk: " + message)
    }
    val c3 = new C3
  }
  val c2 = new C2
}
val c1 = new C1
c1.talk("Hello")
c1.c2.c3.talk("World")
```

Self-type으로 상속 처리하기

Self type으로 trait나 class 지정

self type 뒤에 실제 상속할 trait나 class를 넣어서 상속과 동일한 효과를 보기 위한 방식으로 직접 확장보다 간접 확정을 하는 선언을 위해 사용되는 구조이다.

In other words, what is the point of doing this:

```
trait A
trait B { this: A => }
```

when you could instead just do this:

```
trait A
trait B extends A
```

자가 유형은 특성을 직접 확장하지는 않더라도 특성을 다른 특성과 혼합해야 한다고 선언하는 방법입니다.

실제 정의된 것을 명기 없이 가져오는 것이므로 혼란은 올 수 있지만 상속표시 없이 처리가 된다.

Where to use self-types?

Self type을 지정할 경우 실제 클래스에 사용되는 순서

```
trait MustHave1
trait MustHave2
trait Independent

// constrain the traits A, B and C
trait A { self: MustHave1 => }
trait B { self: MustHave2 => }
trait C { self: MustHave1 with MustHave2 => }
```

```
// correct usage
class Test1 extends MustHave1 with A
class Test2 extends MustHave2 with B
class Test3 extends MustHave1 with MustHave2 with C
class Test4 extends MustHave1 with Independent with A
class Test5 extends Test1 with A
class Test6 extends Test3 with A with B
```

명시적 extends 처리 : class

클래스를 만들고 이를 extends한 trait를 만들면 실제 클래스가 명기적으로 trait에서 사용가능하고 이 클래스의 인스턴스를 생성할 때 항상 trait를 with 다음에 위치한다.

Class 정의

Trait 에 class 상속

인스턴스 생성시 trait 사용

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

scala> trait Get extends A {
     |     def getName = name
     | }
defined trait Get

scala> val a = new A("dahl") with Get
a: A with Get = $anon$1@7eb01546

scala> a.getName
res39: String = dahl
```

Self type 정의

하나의 trait을 정의하고 self type으로 지정해서 그 내용을 가져온다. 이를 가지고 추가적으로 확장도 가능하다.

```
scala> trait A {
     |    def foo = "foo"
     | }
defined trait A

scala> trait B {
     |    this : A =>
     |    def foobar = foo + "bar" }
defined trait B
```

```
scala> trait C {
     |    this : A =>
     |    def fooNope = foo + "Nope"
     | }
defined trait C
```

```
scala> trait D {
     |    this : B =>
     |    def barNope = foobar + "Nope"
     | }
defined trait D
```

Self type 정의 및 실행

2개의 trait를 정의하고 하나의 클래스 내에 self type으로 지정하면 정의가 되고 실제 인스턴스 만들때도 with 문으로 구성한다.

```
scala> trait FooAble {
     |    def foo()  = "foo "
     | }
defined trait FooAble

scala> trait BazAble {
     |    def baz() = " baz too"
     | }
defined trait BazAble

scala> class BarUsing {
     |    this : FooAble with BazAble =>
     |    def bar() = s"bar calls foo:${foo()} and baz : ${baz()} "
     | }
defined class BarUsing

scala> val bar = new BarUsing with FooAble with BazAble
bar: BarUsing with FooAble with BazAble = $anon$1@249fd630

scala> bar.bar()
res38: String = "bar calls foo:foo  and baz :  baz too "
```

타 trait나 인스턴스 참조

Self Type Annotation 정의

하나의 trait을 이용해서 self type annotation을 작성하고 class를 정의한다.

```
scala> trait Vehicle {
     |    def fuelType = "diesel"
     | }
defined trait Vehicle
```

```
scala> class Car {
     |    self : Vehicle =>
     |    def fuel = self.fuelType
     | }
defined class Car
```

동일한 이름으로 메소드 정의할 때 컴파일 오류 발생

Self Type Annotation 실행

self type annotation은 간접적 확장이므로 새로운 인스턴ㅅ를 만들때 with 문을 사용해서 배정해야 한다.

```
scala> val c = new Car
<console>:12: error: class Car cannot be instantiated because it does not confor
m to its self-type Car with Vehicle
       val c = new Car
               ^


scala> val c = new Car with Vehicle
c: Car with Vehicle = $anon$1@1766b674

scala> c.fuel
res31: String = diesel
```

Self type을 이용한 object 처리

Trait를 지정할 때 self type 사용

하나의 추상 trait을 self type으로 지정한 trait이 있다.

```
scala> trait User {
     |     def name : String
     | }
defined trait User

scala> trait B {
     |     user : User =>
     |     def foo() {
     |         println(name)
     |     }
     | }
defined trait B
```

Object를 이용한 self type 활용

Object가 사용할 Mix trait를 상속했고 User trait 옆에 실제 구현된 블록을 정의한다. 실제 foo 메소드를 호출하면 구현된 name이 처리되는 것을 확인할 수 있다.

```
scala> object Main extends B with User { override def name="Dahl" }
defined object Main
```

```
scala> Main.foo
Dahl
```

단일 Self type

Self type 처리

상속하지 않고 상속하는 클래스를 미리 지정해서 그 내용을 사용한다고 처리하면 실제 클래스에 동일한 이름을 검색해서 처리해준다.

```
scala> trait User {
     |     def username: String
     | }
defined trait User

scala> trait Tweeter {
     |     this: User =>
     |     def tweet(text:String) = println(s"$username : $text")
     | }
defined trait Tweeter

scala> class VTweeter(val username_ : String) extends Tweeter with User {
     |     def username = s"real $username_"
     | }
defined class VTweeter

scala> val r = new VTweeter("DAHL")
r: VTweeter = VTweeter@53727b8a

scala> r.username
res415: String = real DAHL

scala> r.tweet("DAHLMOON")
real DAHL : DAHLMOON
```

상속해서 처리

상속해서 처리하면 필요한 것을 전부 구현해야 하지만 self type으로 처리하면 자동으로 처리되는 것을 볼 수 잇다.

```
scala> trait User {
     |    def username : String
     | }
defined trait User

scala> trait Tweeter extends User {
     |    def username  : String = "Dahl"
     |    def tweeter(text:String) = println(s"$username : $text")
     | }
defined trait Tweeter

scala> class UTweeter(val username_ : String) extends Tweeter {
     |    override def username = s"real $username_"
     | }
defined class UTweeter

scala> val r = new UTweeter("MOON")
r: UTweeter = UTweeter@37043e85

scala> r.tweeter("xxxx")
real MOON : xxxx
```

복수 개 Self type 처리

여러 개를 self 타입 처리

self type에 여러 개를 정의하려면 with문으로 연결해서 사용한다.

```
trait A {
  def aValue = 1
}
trait B {
  def bValue = 1
}
trait C {
  self: A with B =>
  def total = aValue + bValue
}

class ABC extends A with B with C
```

구조적 self type

구조적 타입의 이점

구조적 타이핑은 특성을 구현하도록 클래스를 수정할 수 없거나 결합을 줄이고 재사용을 늘리려는 경우에 매우 유용합니다.

구조적 타이핑 대신 trait의 이점은 클래스의 역할을 설명하는 방법입니다.

구조적 타입 지정

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
```

구조적 타입을 타입 별칭 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
}
```

Object에 self type 사용하기

Object/Class에 self type 1

Object 내에 self type을 지정하면 실제 self 가 인식되지 않는다.

```
scala> trait T {
     |    def t = 42
     | }
defined trait T

scala> object X {
     |    self: T =>
     |    def f = self.t
     | }
<console>:14: error: value t is not a member of object X
          def f = self.t
                       ^


scala> class C {
     |    self: T =>
     |    def f = self.t
     | }
defined class C
```

Object/Class에 self type 2

클래스 내의 with 문을 이용해서 처리하면 새로운 인스턴스가 만들어지고 내부 메소드를 호출해서 처리되는 것을 볼 수 있다.

```
scala> (new C with T).f
res25: Int = 42

scala> new C
<console>:13: error: class C cannot be instantiated because it does not conform
to its self-type C with T
       new C
       ^
```