Returning the "Current" Type in Scala

http://tpolecat.github.io/2015/04/29/f-bounds.html

이글을 읽다가 내용이 좋은것 같아

1. 나도 정리해 보고
2. 한번 공유 해보려 함

Needs

상위 클래스를 상속받은 하위 클래스가 자신의 타입을 반환하게 강제 싶다.

뭔말이래..?

상속을 통한 타입 반환

OOP의 특징중의 하나인 상속을 활용한 간단한 코드를 보자.

```
trait Pet {
  def name: String
  def renamed(newName: String): Pet
}

case class Fish(name: String, age: Int) extends Pet {
  def renamed(newName: String): Fish = copy(name = newName)
}
```

잘된다.

```
scala> val a = Fish("Jimmy", 2)
a: Fish = Fish(Jimmy,2)

// renamed 함수가 Fish 타입을 반환하며 문제가 없다.
scala> val b = a.renamed("Bob")
b: Fish = Fish(Bob,2)
```

시시하다잉...

그러나...

우리가 정의한 **`Trait Pet`**은 상속받은 구현체에 어떤 강제도 할수 없다.

```
case class Kitty(name: String, color: Color) extends Pet {
  def renamed(newName: String): Fish = new Fish(newName, 42) // oops
}
```

Oops, 고양이 이름만 바꾸었는데 물고기가 되다니...

The problem

- 상위 Trait에서 renamed는 Pet으로 되어 있어서 Pet의 하위 타입인 Fish 를 반환하는건 문제가 되지 않는다.
- 하지만 우리가 원하는 결과는 아니다. Compiler의 도움을 받아서 하위 구현체에 타입에 대한 어떤 제약을 하고 싶다.
- 이름만 바꾸었으니 기존 종족은 유지되도록 구현을 강제해보자.

1. 반환 타입이 강제되도록 한번해보자.

general 함수를 만들어보자.

```
def esquire[A <: Pet](a: A): A = a.renamed(a.name + ", Esq.")
```

```
<console>:28: error: type mismatch;
 found   : Pet
 required: A
       def esquire[A <: Pet](a: A): A = a.renamed(a.name + ", Esq.")
                                                           ^
```

임의의 **A <: Pet**에 대해 반환타입이 구체적이지 않아서 위 코드는 컴파일 되지 않는다. 현재 상황에서는 반환타입을 Pet으로 하는것이 최선이다. ㅠㅠ

2. F-Bounded Types

대부분 처음 들어보는 용어 일것이라 생각된다.

F-bounded typs은 하위 타입의 타입을 상위 타입의 type parameter로 전달한다.

이를 통해서 superclass로 구현 대상의 타입 정보를 전달할수 있다.

```
Pet[A <: Pet[A]]
```

자기 참조의 특성때문에 코드를 처음 보면 난해하게 느껴질수도 있다.
첨엔 다 이해 하려 하지 말고 **그냥 받아들이자.**
시간이 해결해줄것이다.

F-Bounded 타입 패턴을 이용한 구현

```
trait Pet[A <: Pet[A]] {
  def name: String
  def renamed(newName: String): A // note this return type
}
```

A는 Pet을 상속 받는 녀석이고 renamed는 그 A를 반환한다.
이제 Pet Trait은 누가 구현하는지 알고 그녀석을 반환할수 있다.
물론 그건 구현하는 녀석이 자신을 상위 Trait으로 타입을 전달해 줬기 때문이다.

잘된다.

```
case class Fish(name: String, age: Int) extends Pet[Fish] { // note the type argument
  def renamed(newName: String) = copy(name = newName)
}

scala> val a = Fish("Jimmy", 2)
a: Fish = Fish(Jimmy,2)

scala> val b = a.renamed("Bob")
b: Fish = Fish(Bob,2)


// 위에서 실패했던 generic한 rename 메소드를 만들수 있게 되었다. Pet[A]의 renamed가 A 타입을 반환하니까.
scala> def esquire[A <: Pet[A]](a: A): A = a.renamed(a.name + ", Esq.")
esquire: [A <: Pet[A]](a: A)A

scala> esquire(a)
res8: Fish = Fish(Jimmy, Esq.,2)
```

그러나...

아직 타입 파라메터에 바른 값을 넣는것을 강제할수는 없다.

```
case class Kitty(name: String, color: Color) extends Pet[Fish] { // oops
  def renamed(newName: String): Fish = new Fish(newName, 42)
}
```

고양이가 또 물고기가 되었다 ㅡㅡ;;

self type annotation을 활용해보자.

```scala
trait Pet[A <: Pet[A]] { this: A => // self-type
  def name: String
  def renamed(newName: String): A
}
```

self type annotation을 통하여 하위 구현체는 타입 파라메터와 같은 타입이 되도록 강제하고 있다.

```scala
case class Kitty(name: String, color: Color) extends Pet[Fish] {
  def renamed(newName: String): Fish = new Fish(newName, 42)
}
```

```
<console>:19: error: illegal inheritance;
 self-type Kitty does not conform to Pet[Fish]'s selftype Fish
       case class Kitty(name: String, color: Color) extends Pet[Fish] {
                                                            ^
```

이제 타입 파라메터 Fish와 구현체 Kitty가 다른 타입이기 때문에 컴파일이 되지 않는다.

상속의 상속....

Pet을 상속한 class를 다른 class가 상속하면 더이상 타입을 찾지 못한다.

```
class Mammal(val name: String) extends Pet[Mammal] {
  def renamed(newName: String) = new Mammal(newName)
}

class Monkey(name: String) extends Mammal(name) // hmm, Monkey is a Pet[Mammal]

res0: Mammal = Mammal@5587afd3
```

앗 또 안된다!

OTL

Typeclass를 써볼까?

Pet에서 rename 함수를 분리 해서 그부분을 typeclass로 구현해보자.

```
trait Pet {
  def name: String
}

// rename 함수가 Pet에서 분리 되었다
trait Rename[A] {
  def rename(a: A, newName: String): A
}
```

Typeclass를 이용한 구현

```
case class Fish(name: String, age: Int) extends Pet

object Fish {
  // Fish의 rename 함수는 Rename Trait을 이용하여 구현한다.
  implicit val FishRename = new Rename[Fish] {
    def renamed(a: Fish, newName: String) = a.copy(name = newName)
  }
}
```

Typeclass와 implicit class의 조합

implicit class 를 이용해서 이전에 Pet에서 rename 함수를 호출하는것 처럼 구현할수 있다.

```
implicit class RenameOps[A](a: A)(implicit ev: Rename[A]) {
  def renamed(newName: String) = ev.renamed(a, newName)
}

scala> val a = Fish("Jimmy", 2)
a: Fish = Fish(Jimmy,2)

scala> val b = a.renamed("Bob")
b: Fish = Fish(Bob,2)

// F bounded 패턴을 이용해서
scala> def esquire[A <: Pet : Rename](a: A): A = a.renamed(a.name + ", Esq.")
esquire: [A <: Pet](a: A)(implicit evidence$1: Rename[A])A

scala> esquire(a)
res10: Fish = Fish(Jimmy, Esq.,2)
```

구현하는 매커니즘과 패턴은 다르지만 사용하는 입장에서는 같은 syntax를 사용할수 있다.

Typeclass만?

상속 금지! ad-hoc polymorphism으로 만들어 보자

```
trait Pet[A] {
  def name(a: A): String
  def renamed(a: A, newName: String): A
}

implicit class PetOps[A](a: A)(implicit ev: Pet[A]) {
  def name = ev.name(a)
  def renamed(newName: String): A = ev.renamed(a, newName)
}
```

이제 물고기는 상속을 받지 않는다!

```
case class Fish(name: String, age: Int)

object Fish {
  implicit val FishPet = new Pet[Fish] {
    def name(a: Fish) = a.name
    def renamed(a: Fish, newName: String) = a.copy(name = newName)
  }
}

// renamed 함수는 `PetOps`를 통해서 실행한다.
scala> Fish("Bob", 42).renamed("Steve")
res0: Fish = Fish(Steve,42)
```