Functional Programming and the Scala Language

Lecture 9

Eugene Zouev
Innopolis University
Spring Semester 2018

Type Bounds
Abstract Members

Type Bounds

Variance: Explanation

From the previous lecture

`Base`

`Derived`

`Queue[Base]`

`Queue[Derived]`

Invariance

Typical (but **not ubitiqous**) for C++.

Contravariance

`Queue[Base]`

`Queue[Derived]`

A bit artificial case; doesn't have any sense for queues. However, sometimes it does make sense.

Covariance

`Queue[Base]`

`Queue[Derived]`

Typical for most cases; intuitively obvious.

Variance: Notation

From the previous lecture

```
class Queue[T]
{
  def head: T = ...
  def tail: Queue[T] = ...
  def enqueue(x: T): Queue[T] = ...
  ...
}

// Companion object
object Queue
{
  def apply[T](xs: T*): Queue[T] =
        new Queue[T](xs.toList, Nil)
}
```

So, for the class `Queue`, can we consider `Queue[String]` as a particular case for, say, `Queue[AnyRef]`?

If yes, we should make the `Queue` **covariant** in respect of its type parameter `T`.

Rules:

- By default, classes are **invariant**
- To make a class **covariant**, **add + sign** to its generic parameter
- To make a class **contravariant**, **add – sign** to its generic parameter

```
class Queue[T]
```

```
class Queue[+T]
```

```
class Queue[-T]
```

Covariance

From the previous lecture

Example: Queue class

```
class Queue[+T]
{
  def head: T = ...
  def tail: Queue[T] = ...
  def enqueue(x: T): Queue[T] = ...
  ...
}

// Companion object
object Queue
{
  def apply[T](xs: T*): Queue[T] =
        new Queue[T](xs.toList, Nil)
}
```

Actually, this is <u>not completely correct</u> example; Why? ➔

Star * sign after type denotes several actual arguments in the call

```
val q1 = Queue(1, 2, 3)        // Queue[Int]
val q2 = Queue("One", "Two")   // Queue[String]
```

```
q1 = q2   // correct
```

Covariance & Queue

```
class Queue[+T] {
  def head: T = ...
  def tail: Queue[T] = ...
  def enqueue(x: T): Queue[T] = ...
  ...
}

class StrangeQueue extends Queue[Int]
{
  override def enqueue(x: Int) = {
    println(math.sqrt(x))
    super.enqueue(x)
  }
}
```

Completely correct: `StrangeQueue` is a subclass of `Queue[Int]` and, hence, a subclass of `Queue[Any]`

Completely correct: values of type `String` can be added to `Queue[Any]`

Counterexample:

```
val x: Queue[Any] = new StrangeQueue

x.enqueue("abc")
```

Taking both lines together, we get completely incorrect effect: apply `sqrt` to `String`s!

Solution: Lower Bound

Step 1: make enqueue ***generic* (polymorphic)**

Step 2: **add <u>lower bound</u> for its type parameter**

```
class Queue[+T] {
  def head: T = ...
  def tail: Queue[T] = ...
  def enqueue[U >: T](x: U) =
    new Queue[U](head, x::tail)
  ...
}
```

The construct like

`U >: T` lower bound

means that the type of enqueue's parameter U must be **<u>a base class for T</u>** (or the type T itself)

Lower Bound: Example

```
class Queue[+T] {
  def head: T = ...
  def tail: Queue[T] = ...
  def enqueue[U >: T](x: U) =
    new Queue[U](head, x::tail)
  ...
}
```

```
class Fruit
class Apple extends Fruit
class Orange extends Fruit

val q: Queue[Apple]

q.enqueue(new Orange)
```

`q` is of type `Queue[Apple]` that is a subclass (particular case) of `Queue[Fruit]`

We add an item of type `Orange` (which is also a subclass of `Fruit`) to the collection of `Apple`s.

The result is `Queue[Fruit]` – see the implementation of `enqueue`

Upper Bound

```
...
def sort[T <: Ordered[T]](xs: List[T]) = ...
...
```

The construct like

`T <: U` upper bound

means that the type of `sort`'s parameter `T` must be **<u>a subclass of</u>** <u>`U`</u>

Examples will follow…

Abstract Members

Abstract Members: Introduction

Java, C#:
abstract classes, abstract methods

C++:
abstract classes, “pure virtual” methods.

Scala: **more general approach**:
- abstract methods
- abstract var-variables
- abstract val-variables
- abstract **types**

> M.Odersky:
> A member of a class or trait is abstract if the member does not have a complete definition in the class.

Abstract Members: Example

Abstract trait

```
trait Abstract
{
  type T
  def transform(x: T): T
  val initial: T
  var current: T
  val x: String
}
```

Abstract type: to be made concrete in subclass(es)

Abstract method: uses abstract type

Abstract variables: use abstract type

Abstract val: uses concrete type

```
class Concrete extends Abstract
{
  type T = String
  def transform(x: String) = x + x
  val initial = "hi"
  var current = initial
  val x = "hello"
}
```

Abstract Types

Why abstract types? - an example

Step 1: Abstract classes

```
class Food
abstract class Animal {
  def eat(food: Food)
}
```

The idea of the example is to model eating habits of animals

Step 2: Concrete classes: real animal & real food

```
class Grass extends Food
class Cow extends Animal {
  override def eat(food: Grass)
}
```

Why not to allow this?

Error: this method doesn't override `eat` from `Animal`

Abstract Types

```
class Food
abstract class Animal {
  def eat(food: Food)
}
```

Counterexample

```
class Grass extends Food
class Cow extends Animal {
  override def eat(food: Grass) = { }
}
```

Suppose this is correct...

```
class Fish extends Food

val myCow: Animal = new Cow
myCow.eat(new Fish)
```

...it should be correct as well!

You could feed fish to cows!

Abstract Types

Food and cows: **The solution**

```
class Food
abstract class Animal {
  def eat(food: Food)
}
```

Now each concrete animal can consume its appropriate food only

```
class Food
abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood)
}
```

Here, we don't know which kind of food is suitable for a concrete animal; the only thing we know for sure is that each concrete food is actually Food (i.e., is a subclass of Food)

Abstract Types

Food and cows: **The solution**

```
class Food
abstract class Animal {
  type SuitableFood <: Food
  def eat(food: SuitableFood)
}
```

```
class Grass extends Food
class Cow extends Animal {
  type SuitableFood = Grass
  override def eat(food: Grass) = { }
}
```

The suitable food for cows is obviously grass

Overrides eat method correctly

The same counterexample:

```
class Fish extends Food
val myCow: Animal = new Cow
myCow.eat(new Fish)
```

error: type mismatch;
found : `Fish`
required: `myCow.SuitableFood`

Cows do not eat fish!

Abstract vals

```
trait Abstract
{
  ...
  val x: String
}
```

Abstract val; **no initialization.** In subclasses x might get different values (depending on the need of each subclass)

```
class Concrete1 extends Abstract
{
  ...
  val x = "Hello"
}
```

```
class Concrete2 extends Abstract
{
  ...
  val x = "Bye-bye"
}
```

Important remark: declaration of x looks very similar to abstract method!

```
def x: String
```

Abstract vals

Important remark: declaration of x looks very similar to abstract method without parameters.

```
trait Abstract
{
  ...
  val x: String
  def y: String
}
```

```
class Concrete extends Abstract
{
  ...
  val x = "Hello“
  def y: String = { ... }
}
```

The similarity is not occasional:

```
val obj = new Concrete
...
xx = obj.x
yy = obj.y
...
```

In client code, the access to val looks exactly like access to method.

(The difference is that each access to val returns exactly the same value, whereas the access to method might return different values.)

Abstract vals

The conclusion from the previous considerations:

- An abstract val restricts its implementation: it can be redeclared ("concretized") in a subclass as a declaration of corresponding val but not as a var neither as a def.
- An abstract method can be redeclared either as a var/val or as a method.

```
abstract class Fruit
{
  val v: String // abstract val
  def m: String // abstract method
}
```

```
abstract class Apple extends Fruit
{
  val v: String // (still abstract) val
  val m: String // redeclaration of def as a val
}
```

```
abstract class BadApple extends Fruit
{
  def v: String // Error: cannot redeclare val as def
  def m: String
```

Next time or in labs:

- Abstract `vars` & getters/setters
- Anonymous classes
- Lazy `vals`
- Structural subtyping
- Enumerations