Functional Programming and the Scala Language

Lecture 8

Eugene Zouev
Innopolis University
Spring Semester 2018

Option, Some and None
Implicit conversions
Extractors

Option, Some and None

Type Option

```
val capitals =
    Map("France"->"Paris", "Japan"->"Tokyo"

val capFrance = capitals get "France"

val capNP = capitals get "North Pole"
```

What's the **type** of the result?

- **Both results are of type** `Option[String]`

`Option[`*type*`]` is the type that represents objects that **can have no values.**

`Option[String]` is either a string or… **nothing**

Type Option

To be more precise:

```
val x: Option[String]
...
```

- x is either Some(x) where x is a real value
- or None, if no value

How to get "real" values from Option:

```
def show(x: Option[String]) =
   x match {
      case Some(x) => x
      case None=> "?"
   }
```

```
show(capitals get "France")     // returns Some("France")
show(capitals get "North Pole") // returns "?"
```

Type Option

Java & C#:
A value of type `String` can contain either a string or... `null`

C++:
A string object can contain either a string or... `0` (now `nullptr`)

A question:
How to represent and absence of a value of any type?
For example, for `Integer`?

C#

```
int? x;
```

C++

```
std::optional<int> x;
```

Scala

```
val x: Option[Integer]
```

Implicit conversions

Scala Collections: Map

From the previous lecture

What is ->?

```
(1 -> "Go to island.")
```
→
```
(1).->("Go to island.")
```

```
("Japan"->"Tokyo")
```
→
```
("Japan“).->("Tokyo")
```

`->` is user-defined binary operator, and this is infix form of its use

This is conventional use form: method call of `->` operator

`->` returns a `Tuple` (pair) of two elements

Notice that `+=` is also an operator; we can use either `x += y` or `(x).+=(y)`

Why `->` is applicable to any type?

```
(Key -> Value)
```

```
Key.->(Value)
```

`Key` can be of any type in maps; does it mean any type has its own `->` operator??

Map: A Problem with ->

From the previous lecture

Preamble: Rationals and Integers

```
class Rational(n: Int, d: Int) {
  ...
  def + (that: Rational): Rational = ...
  def + (i: Int): Rational = ...
}
```

oneHalf.+(oneHalf)

```
val oneHalf = new Rational(1,2)

val one = oneHalf + oneHalf
val oneMore = oneHalf + 1
```

oneHalf.+(1)

```
val oneMore2 = 1 + oneHalf
```

1.+(oneHalf)

Error!!

Int type doesn't contain + operator for arguments of type Rational!!

Solution: Implicit Conversions

Solution for `Rational`

```
class Rational(n: Int, d: Int) {
  ...
  def + (that: Rational): Rational = ...
  def + (i: Int): Rational = ...
}
```

```
implicit def intToRational(x: Int) =
                              new Rational(x,1)
```

Implicit conversion

How this gets interpreted:

```
val oneMore2 = 1 + oneHalf
```

Step 1: compiler tries to find `+` operator in `Int` class; there are some but none of them accepts `Rational`s.

Step 2: compiler looks for an implicit conversion `Int->`*`SomeType`*, provided that *`SomeType`* has `+` operator applicable to `Rational`s. YESS, there is one!

`1 + oneHalf` → `intToRational(1) + oneHalf`

Implicit Conversions and ->

```
Map(1->"one", 2->"two", 3->"three")
```

```
package scala
object Predef
{
  class ArrowAssoc[A](x: A)
  {
    def ->[B](y: B): Tuple2[A, B] = Tuple2(x, y)
  }
  implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] =
       new ArrowAssoc(x)
  ...
}
```

M.Odersky:

`->` is a method of the class `ArrowAssoc`, a class defined inside the standard Scala preamble (`scala.Predef`). The preamble also defines **an implicit conversion** from `Any` to `ArrowAssoc`.

When you write `1->"one"`, the compiler inserts **a conversion** from `1` to `ArrowAssoc` so that the `->` method can be found.

`1->"one"` → `any2ArrowAssoc(1)->"one"` → `any2ArrowAssoc(1).->("one")`

Implicit Conversions: Rules

Marking Rule:
Only **definitions** marked implicit are available. The `implicit` keyword is used to mark which declarations the compiler may use as implicits.

Scope Rule:
An inserted implicit conversion must be **in scope** as a single identifier... The Scala compiler will only consider implicit conversions that are in scope. To make an implicit conversion available, therefore, you must in some way **bring it into scope**. Moreover, ... the implicit conversion must be in scope as a single identifier. The compiler will not insert a conversion of the form `someVariable.convert`.

One-at-a-time Rule:
Only one implicit is tried. The compiler will never rewrite `x + y` to `convert1(convert2(x)) + y`.

Explicits-First Rule:
Whenever code type checks as it is written, no implicits are attempted. The compiler will not change code that already works.

Naming an implicit conversion:
Implicit conversions can have arbitrary names.

Extractors

Extractors

Typical task: **email address processing**

Suppose we need to check if an input string contains an email address and get access to user name and domain name of the address.

Straightforward (imperative) solution:

```
def isEmail(s: String): Boolean ...
def domain(s: String): String ...
def user(s: String): String ...
```

Use:

```
if ( isEmail(s) )
   println(user(s) + " AT " + domain(s))
else
   println("not an email address")
```

Extractors

Advanced solution: case classes & pattern matching

Suppose we have a pattern like `Email(user,domain)` that can be used for matching:

```
s match {
   case Email(user,domain) =>
             println(user(s) + " AT " + domain(s))
   case _ => println("not an email address")
}
```

Why we **cannot** do like as written above??

- What is the type of the object than gets matched? `String`.
  But type `String` is not a case class; i.e., string do not have representation like `Email(user,domain)`.

Extractors

Before we come to extractors let's come back to `apply` method:

```
class EMail(user: String, domain: String)

object EMail {
    def apply(u: String, d: String) = new EMail(u,d)
}
```

```
val email =
    EMail("john.lord","innopolis.ru")
```

Implicit call to `apply` method

The method constructs and object from components

Extractors

The solution: extractors

Definition:
Extractor is the object that has unapply method.

Extractor is used for checking whether a string meets a condition and for **splitting it into parts**.

```
object Email {
   // Injection method (optional)
   def apply(user: String, domain: String) =
      user + "@" + domain
   // Exctraction method (mandatory)
   def unapply(str: String): Option[(String, String)] =
   {
      val parts = str split "@"
      if ( parts.length == 2 ) Some(parts(0), parts(1))
      else None
   }
}
```

Extractors

Some important remarks:

```
object Email {
   ...
   // Exctraction method (mandatory)
   def unapply(str: String): Option[(String, String)] =
   {
      val parts = str split "@"
      if ( parts.length == 2 ) Some(parts(0), parts(1))
      else None
   }
}
```

This is shorthand; the full form is `Some((parts(0),parts(1))`

Why `Option`, `Some` and `None`?

- For the case when the input string **is not an email address**

Variance

Variance

Common explanation

Suppose there are two related classes:

```
class Base { ... }
class Derived extends Base { ... }
```

…And we have declared two **generic** classes:

```
class Queue[Base] { ... }
class Queue[Derived] { ... }
```

The question:
What is relationship between two queues?

Variance: Explanation

Base

Derived

Covariance

Queue[Base]

Queue[Derived]

Typical for most cases; intuitively obvious.

Contravariance

Queue[Base]

Queue[Derived]

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

Invariance

Queue[Base]

Queue[Derived]

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

Variance

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)
}
```

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]
```

Variance: Notation

```scala
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

```scala
class Queue[T]
```

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

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

Covariance: Example

```
class Queue[+T]
{
   ...
}

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

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

q1 = q2  // correct
```

Actually, this is not completely correct example; see next lecture ☺

Contravariance: Example

Counterexample: why and when contravariance needed.

Trivial class:

```
class OutputChannel[+T] {
  def write(x: T) = ...
}
```

Covariant class

```
val ch1: OutputChannel[String]
ch.write("something")                      // OK
val ch2: OutputChannel[AnyRef] = ch1  // Why not?
ch.write(Array(1,2,3))                    // unsafe!!
```

```
class OutputChannel[-T] {
  def write(x: T) = ...
}
```

Contravariant class

```
val ch1: OutputChannel[String]
ch.write("something")                      // OK
val ch2: OutputChannel[AnyRef] = ch1  // Compiler error!
ch.write(Array(1,2,3))                    // Won't run
```