Functional Programming
and
the Scala Language

Lecture 7

Eugene Zouev
Innopolis University
Spring Semester 2018

Basic Scala Collections:

Set

Map

Implicit conversions

Scala Collections: Common Picture

- Lists are always **immutable** structures
- Arrays are always **mutable** structures
- **Sets** & **Maps** are **mutable & immutable**

More concretely,

- There mutable Sets and immutable Sets;
- There are mutable Maps and immutable Maps.

Other languages (eg, C++):

- `set`/`map` structures exist;
- They are (of course ☺) mutable
- Also, there are `multi_set` & `multi_map` structures.

Scala Collections: Set

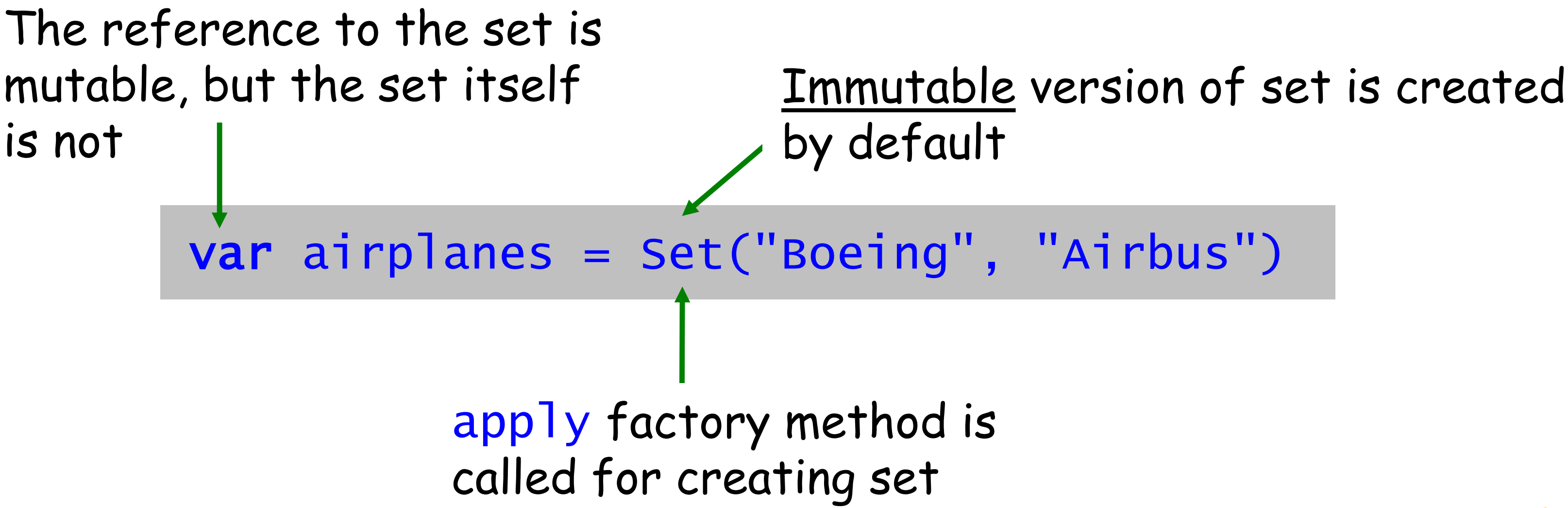
What is "set"?

- A collection of items of the same type where the uniqueness of each element is guaranteed.

What is "uniqueness"?

- The == operator should give false for each pair of set elements.

How to declare and create a set?



The reference to the set is mutable, but the set itself is not

Immutable version of set is created by default

```
var airplanes = Set("Boeing", "Airbus")
```

apply factory method is called for creating set

Scala Collections: Set

```
var airplanes = Set("Boeing", "Airbus")

airplanes += "Bombardier"
```

The += operator doesn’t change the set:
it’s immutable. Instead, it creates a **new set**.
Here, += is exactly like this:

```
airplanes = airplanes + "Bombardier"
```

Some examples of operators on sets:

```
airplanes.contains("Cessna")  // false
airplanes.size                // 3
airplanes ++ Set("TU")        // join
airplanes & Set("IL")         // intersection
...
```

Scala Collections: Set

Where are mutable sets?

```
var airplanes = Set("Boeing", “Airbus")
```

```
var airplanes = scala.collection.immutable.Set("Boeing", "Airbus")
```

Two ways for creating mutable sets:

```
val numbers = scala.collection.mutable.Set(1, 3, 5, 7)
```

```
import scala.collections.mutable
val numbers = Set(1, 3, 5, 7)
```

```
numbers += 9
```

Here, += operator doesn't create a new set; it updates the existing set “in place”.

Scala Collections: Set

Example

```
// 1. Taking the initial text:
val text = "See Spot run. Run, Spot. Run!“

// 2. Splitting it into separate words:
val arrWords = text.split("[ !,.]+")
  // The result is:
  // Array[String]("See","Spot","run","Run","Spot","Run")

// 3. Creating the mutable set: initially empty
val setWords = scala.collection.mutable.Set.empty[String]

// 4. Adding words from array to the set:
for ( word <- arrWords )
  setWords += word.toLowerCase
```

Returns array of strings

Result:

```
setWords: Set("see","run","spot")
```

Scala Collections: Map

Associative array

What is "map"?

- A collection of pairs (Key,Value) where the uniqueness of each first element over all pairs is guaranteed.

What is "uniqueness"?

- The `==` operator should give `false` for comparing first elements (Key) for all pairs.

  The type of each first element in map pairs is the same; so about the type of the second pair element.

M.Odersky:

Maps let you **associate a value with each element of the collection.** Using a map looks similar to using an array, except that instead of indexing with integers counting from 0, you can use **any kind of key**.

If you import the `scala.collection.mutable` package, you can create an empty mutable map like this:

```
   val map = mutable.Map.empty[String, Int]
```

Scala Collections: Map

How to declare and create a map?

```
val map = mutable.Map.empty[String, Int]
```

Empty map

```
import scala.collection.mutable.Map
val treasureMap = Map[Int, String]()

treasureMap(1) = "Go to island."
treasureMap(2) = "Find big X on ground."
treasureMap(3) = "Dig."
println(treasureMap(2))
```

Empty map

Looks like indexing array elements…

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

`apply` factory method again ☺

```
countries("Germany") = "Berlin"
countries("Austria") = "Wien"
```

Scala Collections: Map

Example

```
import scala.collection.mutable.Map
val treasureMap = Map[Int, String]()

treasureMap(1) = "Go to island."
treasureMap(2) = "Find big X on ground."
treasureMap(3) = "Dig."
println(treasureMap(2))
```

```
import scala.collection.mutable.Map

val treasureMap = Map[Int, String]()

treasureMap += (1 -> "Go to island.")
treasureMap += (2 -> "Find big X on ground.")
treasureMap += (3 -> "Dig.")

println(treasureMap(2))
```

Scala Collections: Map

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??

See some final slides for the explanation

Scala Collections: Map

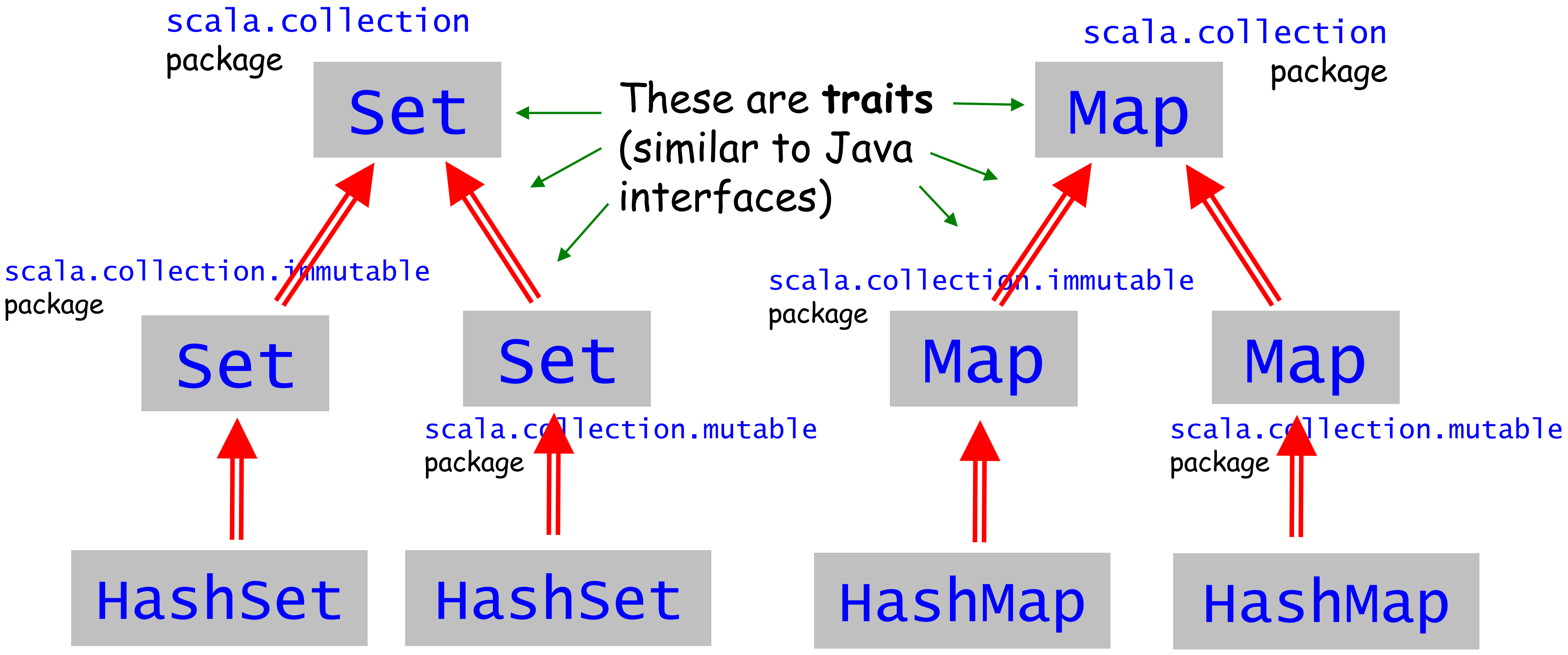
Example: a method that counts the number of times each word occurs in a string

```
def countWords(text: String) = {
  val counts = mutable.Map.empty[String, Int]
  for (rawWord <- text.split("[ ,!.]+"))
  {
    val word = rawWord.toLowerCase
    val oldCount =
          if ( counts.contains(word) )
            counts(word)
          else
            0
    counts += (word -> (oldCount + 1))
  }
  counts
}
countWords("See Spot run! Run, Spot. Run!")
  // Result: Map(see -> 1, run -> 3, spot -> 2)
```

counts

Word	Number of times
see	1
run	3
spot	2

Sets & Maps: Common Picture



scala.collection package

Set

scala.collection package

Map

These are **traits** (similar to Java interfaces)

scala.collection.immutable package

Set

Set

scala.collection.mutable package

scala.collection.immutable package

Map

Map

scala.collection.mutable package

HashSet

HashSet

HashMap

HashMap

Implementations: **classes**

Sets & Maps: Common Picture

Set/Map vs HashSet/HashMap: What’s the difference?

- “Hash” versions are more efficient; they contain hash tables inside for faster search.
- The approach: for <u>mutable</u> Sets/Maps “hashed” versions are always used

```
import scala.collections.mutable
var airplanes = Set("Boeing", “Airbus")
```

Internally, the `HashSet.apply` factory method is called for creating set

- For <u>immutable</u> Sets/Maps the approach is a bit smarter (see next slide).

Sets & Maps: Common Picture

If your immutable set/map contains less than 5 items then special "simple" versions are used by default. Otherwise hashed versions are used.

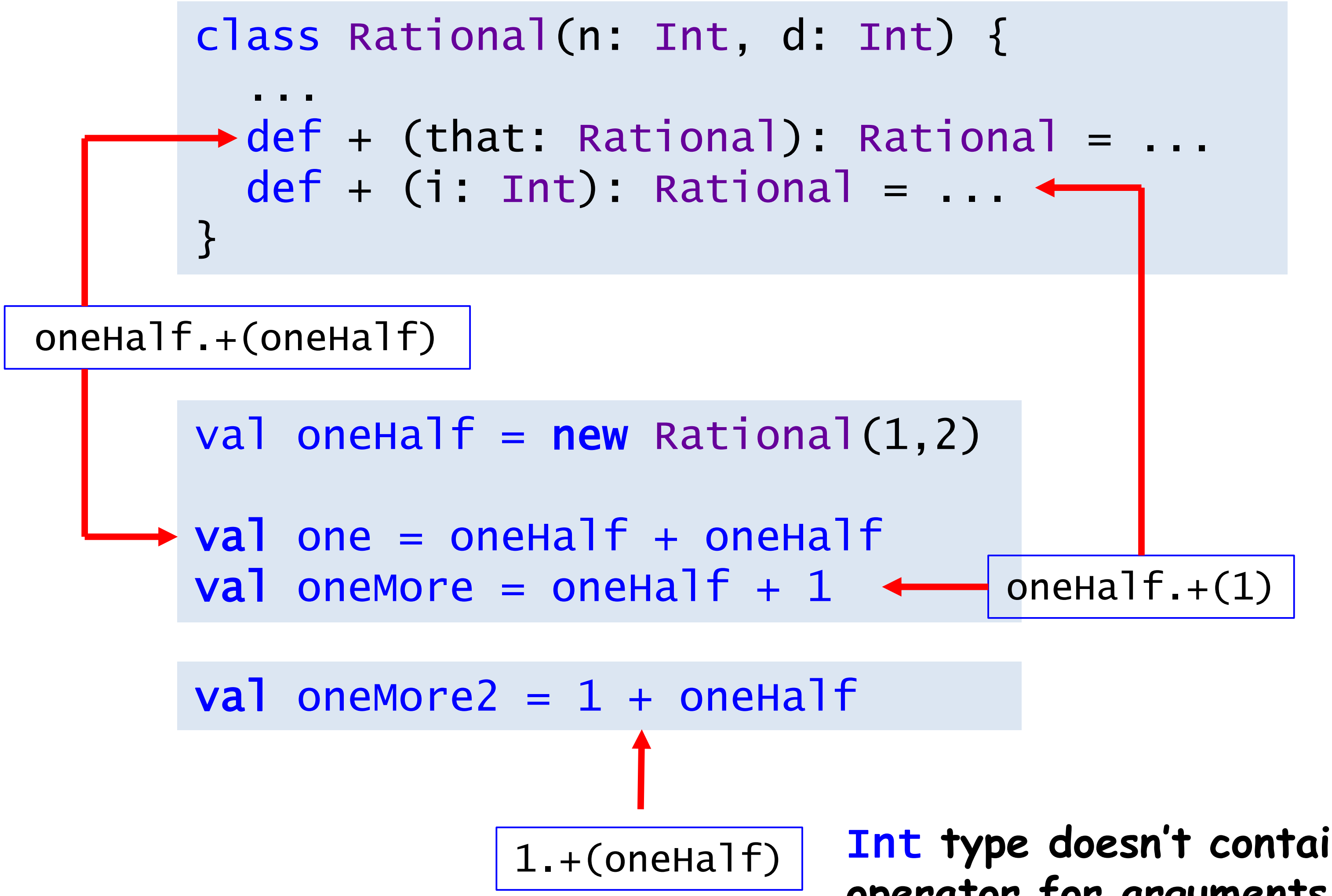
Number of elements	Implementation
0	`scala.collection.immutable.EmptySet`
1	`scala.collection.immutable.Set1`
2	`scala.collection.immutable.Set2`
3	`scala.collection.immutable.Set3`
4	`scala.collection.immutable.Set4`
5 or more	`scala.collection.immutable.HashSet`

The same is with immutable version of Map

Implicit conversions

Map: A Problem with ->

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 Rationals.

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

```
1 + oneHalf
```

→

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

Implicit Conversions and ->

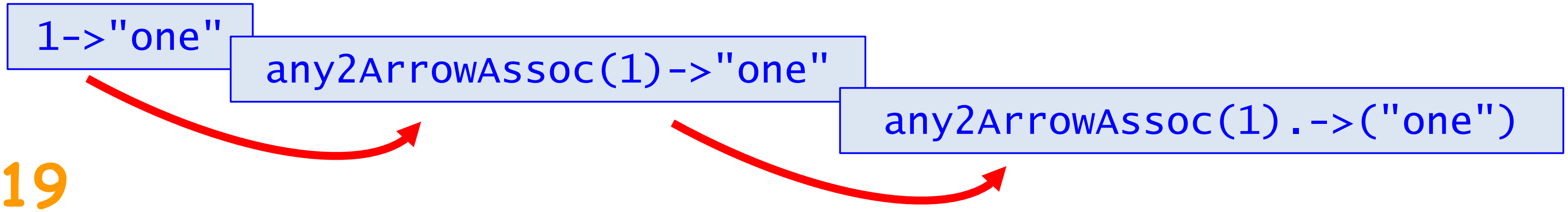
```
Map(1->"one", 2->"two", 3->"three")
```

```
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.