Scala Training 3

Pragmatic Test + Scala Collections

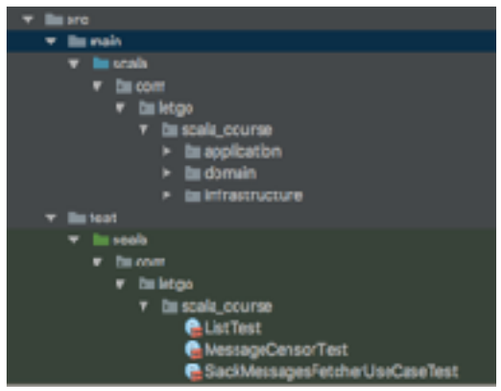
Letgo - 17/02/2017

Gerard Vico
Gerard Madorell

Contents

Pragmatic Test

Pragmatic Test - Where do we put the tests?



By convention, Scala tests are put inside src/test/scala, whereas production code is put in src/main/scala.
By convention, we use the same package structure than production code in test code.

Pragmatic Test - ScalaTest

- Library used for testing in Scala.
- PhpUnit, JUnit… for Scala.
- Different types of syntax: WordSpec, FlatSpec…

Choose one and use only that one for similar type of tests. For example:

- Acceptance test with FeatureSpec
- Behaviour test with WordSpec

It’s like a DSL for testing.

Pragmatic Test - Test Example

```
*/
final class ListTest extends WordSpec with Matchers {
  "A List" should {
    "have a size of zero when it's created as empty" in {
      val emptyList: List[Int] = List.empty[Int]
      emptyList.size shouldBe 0
    }

    "have size of one when it's created with a single element" in {
      val listWithOneElement: List[String] = List("some string")
      listWithOneElement.size shouldBe 1
    }

    "contain an element when it's created with that element inside" in {
      case class SomeCaseClass(someValue: Int)

      val someCaseClass    = SomeCaseClass(10)
      val listWithElement = List(someCaseClass)
      listWithElement.contains(someCaseClass) shouldBe true
    }
  }
}
```

Pragmatic Test - Test Example

Semantics style

```
*/
final class ListTest extends WordSpec with Matchers {
  "A List" should {
    "have a size of zero when it's created as empty" in {
      val emptyList: List[Int] = List.empty[Int]
      emptyList.size shouldBe 0
    }

    "have size of one when it's created with a single element" in {
      val listWithOneElement: List[String] = List("some string")
      listWithOneElement.size shouldBe 1
    }

    "contain an element when it's created with that element inside" in {
      case class SomeCaseClass(someValue: Int)

      val someCaseClass    = SomeCaseClass(10)
      val listWithElement = List(someCaseClass)
      listWithElement.contains(someCaseClass) shouldBe true
    }
  }
}
```

Pragmatic Test - Test Example

Asserts

```
*/
final class ListTest extends WordSpec with Matchers {
  "A List" should {
    "have a size of zero when it's created as empty" in {
      val emptyList: List[Int] = List.empty[Int]
      emptyList.size shouldBe 0
    }

    "have size of one when it's created with a single element" in {
      val listWithOneElement: List[String] = List("some string")
      listWithOneElement.size shouldBe 1
    }

    "contain an element when it's created with that element inside" in {
      case class SomeCaseClass(someValue: Int)

      val someCaseClass    = SomeCaseClass(10)
      val listWithElement = List(someCaseClass)
      listWithElement.contains(someCaseClass) shouldBe true
    }
  }
}
```

Pragmatic Test - Test Example

Test Suite

```
*/
final class ListTest extends WordSpec with Matchers {
  "A List" should {
    "have a size of zero when it's created as empty" in {
      val emptyList: List[Int] = List.empty[Int]
      emptyList.size shouldBe 0
    }

    "have size of one when it's created with a single element" in {
      val listWithOneElement: List[String] = List("some string")
      listWithOneElement.size shouldBe 1
    }

    "contain an element when it's created with that element inside" in {
      case class SomeCaseClass(someValue: Int)

      val someCaseClass   = SomeCaseClass(10)
      val listWithElement = List(someCaseClass)
      listWithElement.contains(someCaseClass) shouldBe true
    }
  }
}
```

Pragmatic Test - Test Example

Test Case

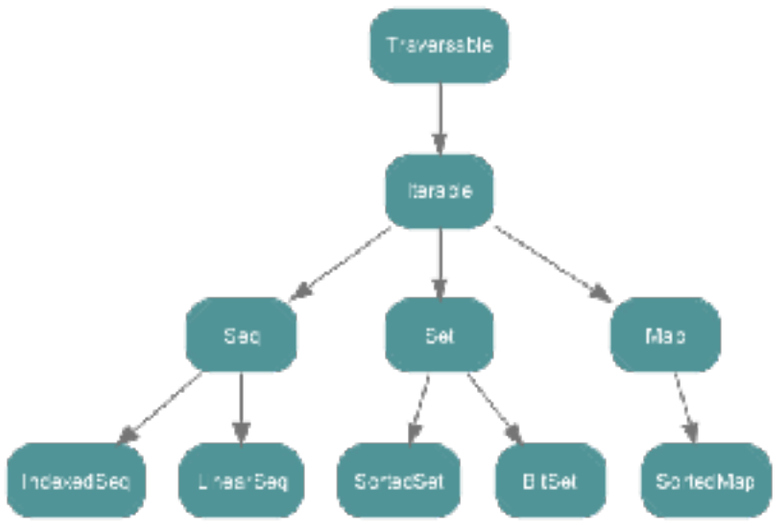
```
final class ListTest extends WordSpec with Matchers {
  "A List" should {
    "have a size of zero when it's created as empty" in {
      val emptyList: List[Int] = List.empty[Int]
      emptyList.size shouldBe 0
    }

    "have size of one when it's created with a single element" in {
      val listWithOneElement: List[String] = List("some string")
      listWithOneElement.size shouldBe 1
    }

    "contain an element when it's created with that element inside" in {
      case class SomeCaseClass(someValue: Int)

      val someCaseClass    = SomeCaseClass(10)
      val listWithElement = List(someCaseClass)
      listWithElement.contains(someCaseClass) shouldBe true
    }
  }
}
```

Pragmatic Test - Test Output

```
[info] ListTest:
[info] A List
[info] - should have a size of zero when it's created as empty (31 milliseconds)
[info] - should have size of one when it's created with a single element (1 millisecond)
[info] - should contain an element when it's created with that element inside (2 milliseconds)
[info] Run completed in 1 second, 704 milliseconds.
[info] Total number of tests run: 3
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 3, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 13 s, completed Feb 15, 2017 12:54:38 PM
```

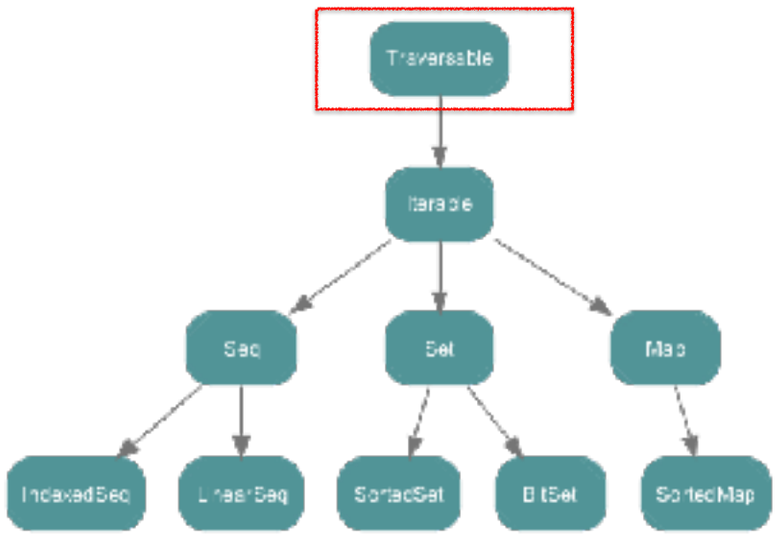
Scala Collections

Collections - Trait tree (interfaces)



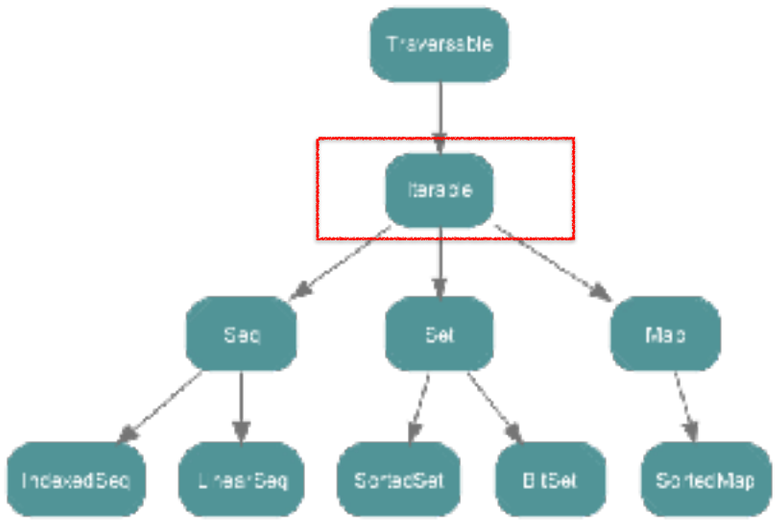
This is the collections framework used in Scala. It follows a hierarchy.
Similar to Java collections hierarchy.

Collections - Trait tree (interfaces)



Traversable -> has an abstract method: foreach()
It gives you map, filter, etc()

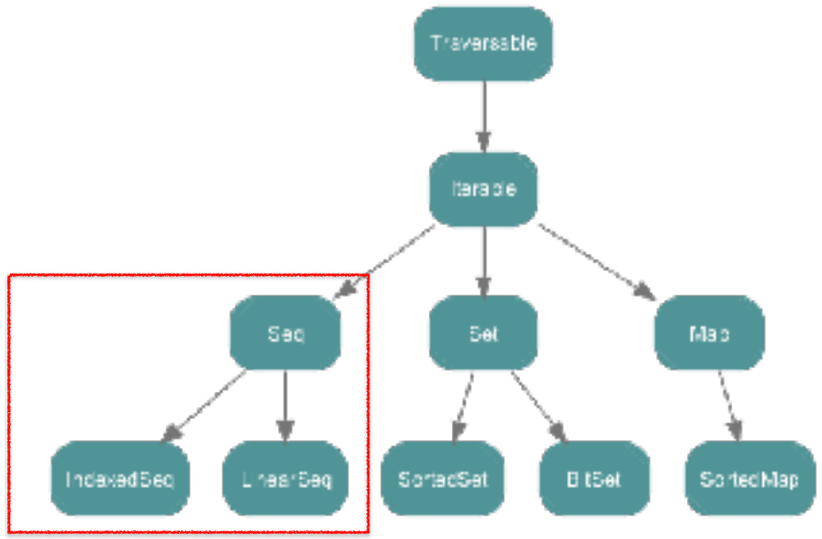
Collections - Trait tree (interfaces)



Iterable -> has an iterator (implements foreach of traversable with an iterator)

```
def foreach[U](f: Elem => U): Unit = {
  val it = iterator
  while (it.hasNext) f(it.next())
}
```

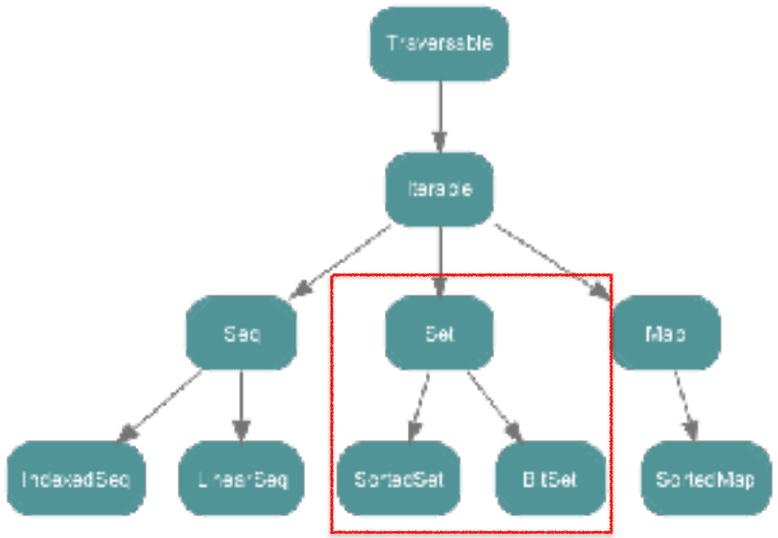
Collections - Trait tree (interfaces)



Sequences = Ordered elements by insertion order with repetitions
Indexed Seq = You can access an element given an index
Linear Seq = You can’t, but it has faster iteration from head to tail

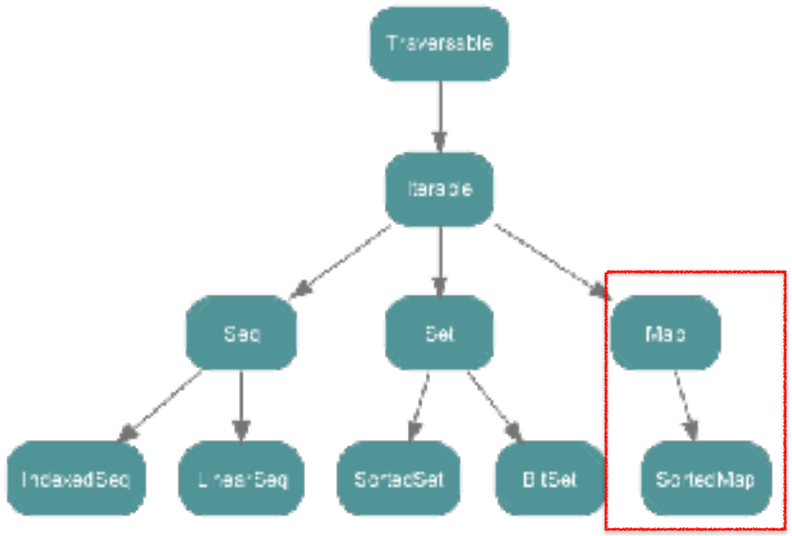
You use LinearSeq or IndexedSeq when you need to have a certain performance characteristics or behaviour, you use the more general Seq when you don't care about the difference.

Collections - Trait tree (interfaces)

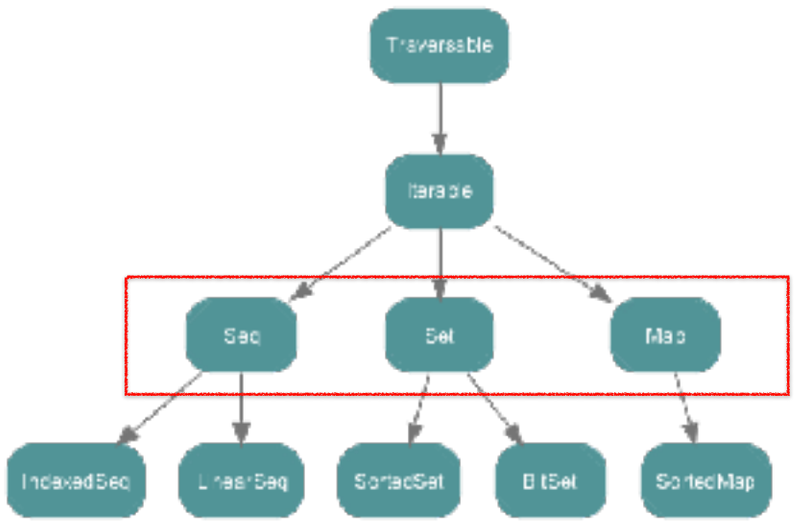


Set = no duplicates
SortedSet = you need order of elements

Collections - Trait tree (interfaces)



Collections - Trait tree (interfaces)



You can use the Seq(), Set() and Map() constructors if you don’t care about the implementation (which is most of the time).
If you need performance tuning, then you can search for a more concrete implementation.
If you want to work specifically with immutable types, use an immutable type for all your methods (for example immutable.List instead of Seq)

Collections - How to instantiate a collection

```
scala> val someSequence = Seq(1, 1, 1, 2, 3, 4)
someSequence: Seq[Int] = List(1, 1, 1, 2, 3, 4)

scala> val someSet = Set(1, 1, 1, 2, 3, 4)
someSet: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 4)

scala> val someMap = Map(1 -> "Cosa", 2 -> "Fina")
someMap: scala.collection.immutable.Map[Int,String] = Map(1 -> Cosa, 2 -> Fina)
```

Collections - Immutable vs Mutable

- Immutable by default!

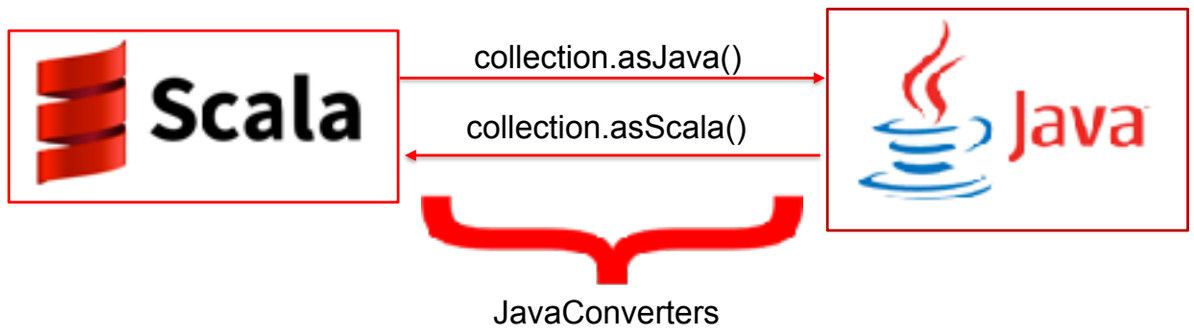
```
scala> val immutableSetByDefault = Set(1)
immutableSetByDefault: scala.collection.immutable.Set[Int] = Set(1)
```

- Have to specify mutability

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val mutableSet = mutable.Set(1)
mutableSet: scala.collection.mutable.Set[Int] = Set(1)
```

Scala.collection -> generic collections / traits
Scala.collection.immutable -> everything immutable
Scala.collection.mutable -> everything mutable

Collections - Conversion Java <-> Scala collections



Use JavaConverters to pass from Scala to Java.
Internally it uses a wrapper, so the collection is still immutable if it was immutable in Scala.

Collections - Conversion Java <-> Scala collections

```
scala> import collection.JavaConverters._
import collection.JavaConverters._
```

```
scala> val someScalaCollection = Seq(1)
someScalaCollection: Seq[Int] = List(1)
```

```
scala> someScalaCollection.asJava
res0: java.util.List[Int] = [1]
```

```
scala> someScalaCollection.asJava.asScala
res1: scala.collection.mutable.Buffer[Int] = Buffer(1)
```

Example of how to convert from Scala to Java and viceversa.

Collections - Conversion Java <-> Scala collections

```
scala> import collection.JavaConverters._
import collection.JavaConverters._
```

```
scala> val someScalaCollection = Seq(1)
someScalaCollection: Seq[Int] = List(1)
```

```
scala> someScalaCollection.asJava
res0: java.util.List[Int] = [1]
```

```
scala> someScalaCollection.asJava.asScala
res1: scala.collection.mutable.Buffer[Int] = Buffer(1)
```

Import java converters

Collections - Conversion Java <-> Scala collections

```
scala> import collection.JavaConverters._
import collection.JavaConverters._
```

```
scala> val someScalaCollection = Seq(1)
someScalaCollection: Seq[Int] = List(1)
```

```
scala> someScalaCollection.asJava
res0: java.util.List[Int] = [1]
```

```
scala> someScalaCollection.asJava.asScala
res1: scala.collection.mutable.Buffer[Int] = Buffer(1)
```

Collections - Conversion Java <-> Scala collections

```
scala> import collection.JavaConverters._
import collection.JavaConverters._
```

```
scala> val someScalaCollection = Seq(1)
someScalaCollection: Seq[Int] = List(1)
```

```
scala> someScalaCollection.asJava
res0: java.util.List[Int] = [1]
```

```
scala> someScalaCollection.asJava.asScala
res1: scala.collection.mutable.Buffer[Int] = Buffer(1)
```

Collections - Conversion Java <-> Scala collections

```
scala> import collection.JavaConverters._
import collection.JavaConverters._
```

```
scala> val someScalaCollection = Seq(1)
someScalaCollection: Seq[Int] = List(1)
```

```
scala> someScalaCollection.asJava
res0: java.util.List[Int] = [1]
```

```
scala> someScalaCollection.asJava.asScala
res1: scala.collection.mutable.Buffer[Int] = Buffer(1)
```

Collections - Tips and tricks - Working with Seqs

- val seq = Seq(1, 2, 3)
- seq :+ 1 -> Seq(1, 2, 3, 1)
- seq +: 1 -> Seq(1, 1, 2, 3)
- seq.length -> 3
- seq(0) -> 1
- seq.reversed -> Seq(3, 2, 1)
- seq.reversed.sorted -> Seq(1, 2, 3)

Collections - Tips and tricks - Working with Sets

- val set = Set(1, 2, 3)
- set + 1 -> Set(1, 2, 3)
- set + 4 -> Set(1, 2, 3, 4)
- set.contains(2) -> true
- set - 2 -> Set(1, 3)

Collections - Tips and tricks - Working with Maps

- val map = Map(“ripo” -> “team”, “pata” -> “negra”)
- map + (“key” -> “value”) -> Map(“ripo” -> “team”, “pata” -> “negra”, “key” -> “value”)
- map - “pata” -> Map(“ripo” -> “team”)
- map.keys -> Iterable(“ripo”, “pata”)
- map.values -> Iterable(“team”, “negra”)
- map.contains(“ripo”) -> true
- map(“ripo”) -> Some(“team”)

Collections - Tips and tricks - Functional Fun

- Can’t spell functional without fun!
- Collections all have functional methods! Wow!
  - someCollection.map
  - someCollection.filter
  - someCollection.foreach
  - someCollection.flatmap
  - someCollection.reduce
  - someCollection.fold
  - ……

Collections - Tips and tricks - Parallelism

- Collections can be transformed to parallelized collections.
  - val someCollection = 1 to 10000000
- Without parallelism:
  - someCollection.map(someHardToComputeFunction)
- With parallelism:
  - someCollection.par.map(someHardToComputeFunction)

Workshop

Workshop

Pragmatic Test Theory Summary

Collection Types Theory Summary

Workshop

- Implement a test for the “countMessagesOfUser” method

```
class MessageAnalyticsService {

  def countMessagesOfUser(messages: Seq[UserMessage], userName: UserName): Int = {
    messages.count(_.userName == userName)
  }
```

```
  "count messages of user" in {

    val msgAnalyticsService = new MessageAnalyticsService()

    // TODO implement test

    true shouldBe false
  }
```

Workshop

- Implement the “groupByUsername” method, it should return a map of user names to sequences of the messages sent by that user.

```
def groupByUserName(messages: Seq[UserMessage]): Map[UserName, Seq[Message]] =
}
```