

State of Scala



Venkat Subramaniam

venkats@agiledeveloper.com

[@venkat_s](https://twitter.com/venkat_s)

Evolutions of Scala

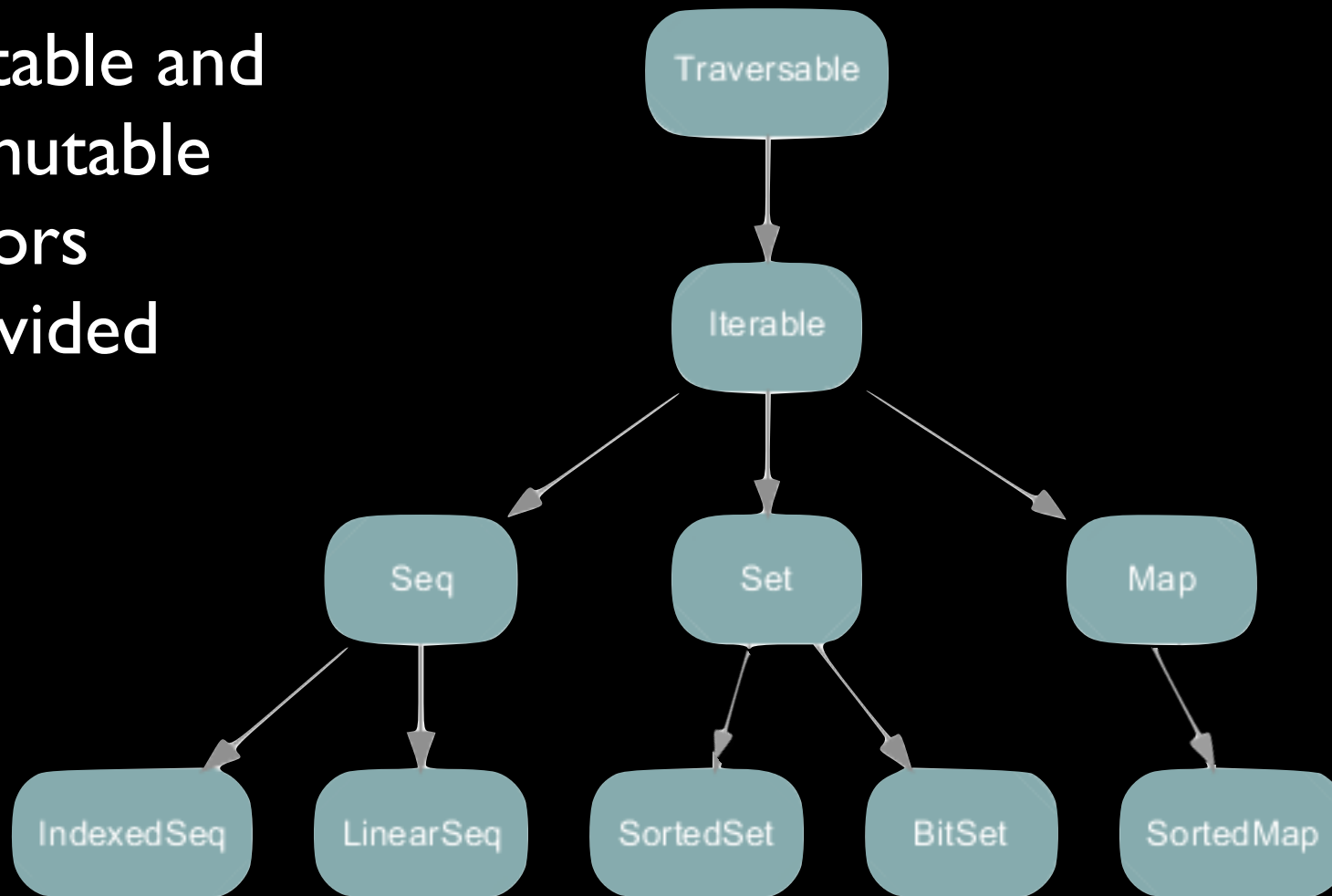
Scala has evolved over the years

Scala 2.8 has some interesting additions and changes

- ★ Collections
- ★ Named and Default Arguments
- ★ package objects
- ★ Chained package clauses
- ★ tailrec annotation
- ★ Type Specialization

Collections

Mutable and
Immutable
flavors
provided



Source: <http://www.scala-lang.org>

Traversable Trait

Top of the Collections hierarchy

Provides quite a few common methods that are available on all collections

Methods to add elements, split and partition the collection

Immutable Collections

List—constant time op on head of the list

Stream—like list but lazy evaluation, so infinite length

Vector—improves on List for constant time op on any element—implemented using Tries

Stack, Queue, Range, Hash Tries, Red-black tree, BitSet, ListMap,

Mutable Collections

ArrayBuffer

ListBuffer

StringBuffer

LinkedList

Double Linked List

Mutable List

Queue

Array Sequence, Stack, ArrayStack, HashTable, Weak
HashMap, Concurrent Map, Mutable BitSet

Stream

```
def isPrime(number : Int) : Boolean = {
  if (number < 2) return false

  for(i <- 2 to math.sqrt(number).toInt)
    if(number % i == 0) return false

  return true
}

def primes() = {
  def nextPrime(number : Int) : Stream[Int] = {
    if(isPrime(number)) {
      number #:: nextPrime(number + 1)
    } else {
      nextPrime(number + 1)
    }
  }

  nextPrime(2)
}

val nPrimes = primes().take(10)
println(nPrimes.toList)
```

#:: and take

```
List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

Views

Collections come in two flavors

Strict collections—elements are evaluated when you create

Lazy collections—elements are evaluated on demand

Most collections are strict, Stream is lazy

You can, however, use views to convert a strict collection into a lazy one

Views

Views provide good modularity

It separates the operations on the collection and can even chain them while providing lazyness

Views

```
val names = List("Jack", "Jill", "Bill", "Bob", "Brad", "Bond")

def len(n : String) = {
  println(n)
  (n, n.length)
}

println(names map len find { _._2 == 3})

println("With views now")

println(names.view map len find { _._2 == 3} )
```

```
Jack
Jill
Bill
Bob
Brad
Bond
Some((Bob,3))
With views now
Jack
Jill
Bill
Bob
Some((Bob,3))
```

Named and Default Args

You can provide default values for arguments

You can refer to parameters using names

```
def power(number : Int, exponent : Int = 2) =  
    (math.pow(number, exponent))
```

```
println(power(2, 3))
```

```
println(power(2))
```

```
println(power(exponent = 4, number = 3))
```

8.0

4.0

81.0

Named Arguments

The name you specify may be a named parameter or a variable in scope—it can't be both

If you place in parenthesis, it is not considered a named argument

Expressions are evaluated in the order you place them

Overriding methods can use different parameter names

Name is based on type checking

package objects...

In the past, package can have traits, classes, objects

You can now put methods, etc. in package scope and provide an easier reach

```
//Utility.scala
package com.agiledeveloper.util

object Utility {
  def power(number : Int, exponent : Int) =
    math.pow(number, exponent)
}

import com.agiledeveloper.util.Utility._

object Use {
  def main(args : Array[String]) = {
    println(power(2, 4))
  }
}
```

package objects to rescue

```
//package.scala
package com.agiledeveloper

package object util {
  def power(number : Int, exponent : Int) =
    math.pow(number, exponent)
}


import com.agiledeveloper.util._

object Use {
  def main(args : Array[String]) = {
    println(power(2, 4))
  }
}
```

You can extend the package objects from traits/classes and acquire their methods into the package

Chained package clauses

```
package foo {  
  package bar {  
    ...  
  }  
}
```



```
package foo  
package bar  
...
```

Less noisy

Lightweight

One big difference, package declaration now brings only the tail package into scope

package foo.bar will not bring in foo into scope

A critical change necessary due to name conflicts

tailrec

Scala has offered limited tail recursion

You had no indication if the recursion was tail recursive

Now you can assert if it is tail recursive

If compiler could not optimize for tail recursion, you get an error

tailrec

```
import scala.annotation.tailrec

def fact(number : Int) = {
  @tailrec def factImpl(number : Int, factorial : Int) : Int = {
    if (number <= 1)
      factorial
    else
      1 * factImpl(number - 1, factorial * number)
  }

  factImpl(number, 1)
}

println(fact(5))
```

```
error: could not optimize @tailrec annotated method:
it contains a recursive call not in tail position
```

```
  @tailrec def factImpl(number : Int, factorial : Int) : Int = {
    ^
one error found
```

If you remove `1 *` the error goes away

Type Specialization

Generic types substitute types with upper bound types

For primitive types, this involves boxing/unboxing overhead

You can avoid this by using `@specialized`

This will specialize for all primitive types

You can ask for specific types using `@specialized(type1, type2,...)`

For unspecialized types, it will use the regular type erasure

Specialization happens only if there is at least one parameter of specialized type or its array

Type Specialization

```
def mult1[T](input : T) = input
def mult2[@specialized(Int) T](input : T) = input

val start1 = System.nanoTime
mult1(2)
val end1 = System.nanoTime
println(end1 - start1)

val start2 = System.nanoTime
mult2(2)
val end2 = System.nanoTime
println(end2 - start2)
```

References

<http://www.scala-lang.org/node/7009>

Thank You!

Venkat Subramaniam
venkats@agiledeveloper.com
twitter: venkat_s

