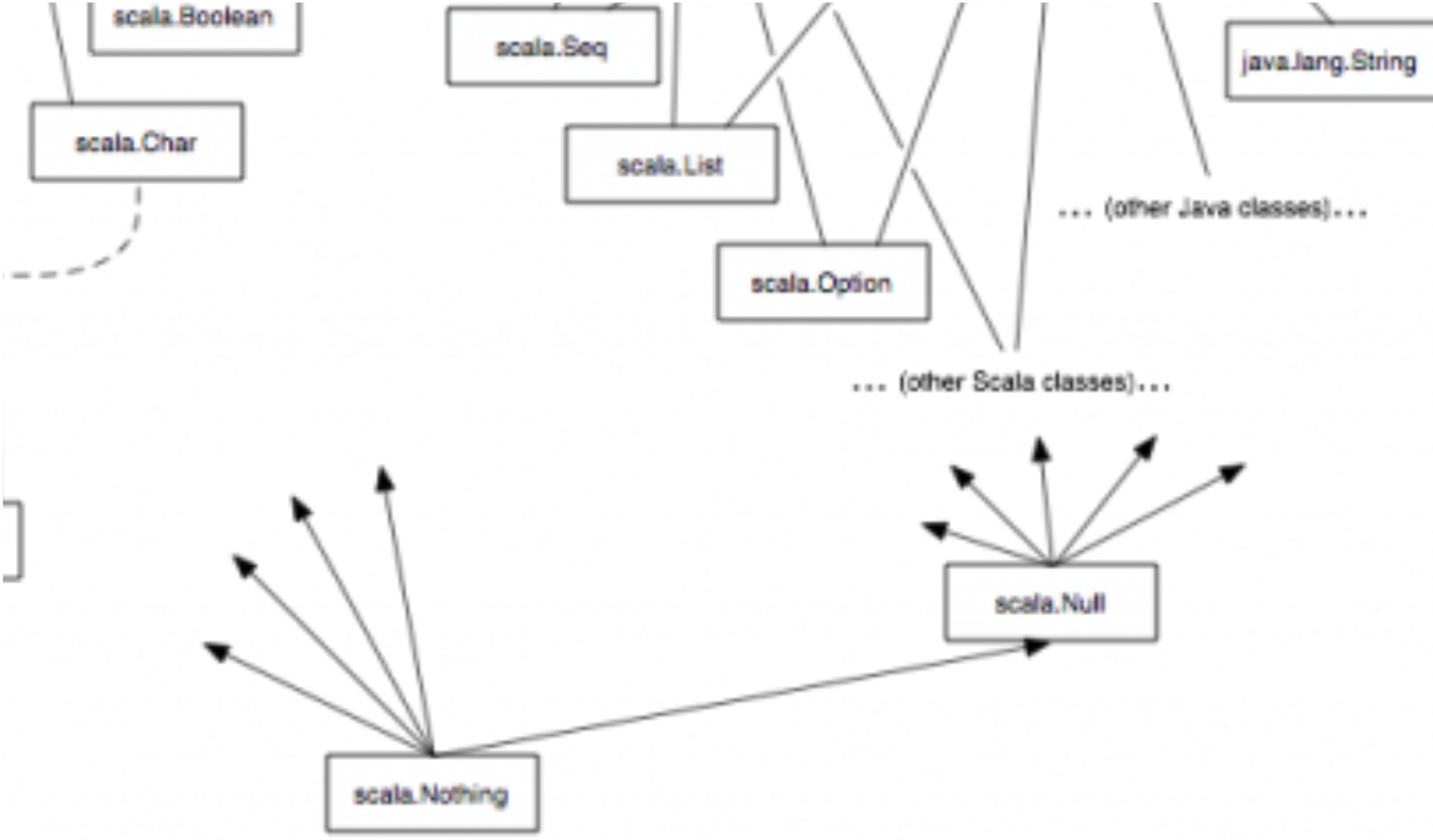
Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC

Equality in Scala


scala.Boolean
scala.Char
scala.Seq
scala.List
scala.Option
java.lang.String
... (other Java classes)...
... (other Scala classes)...
scala.Null
scala.Nothing

Equality in Scala

- The method `eq` on values of type `AnyRef` checks that two objects exist in the same place

Equality in Scala

- The method == checks the “natural” equality relation on a type
- For `AnyRefs`:

```
final def ==(that: Any): Boolean =
  if (null eq this) null eq that
  else this equals that
```

Equality in Scala

- The inherited `equals` method is the same as `eq` on values of type **AnyRef**
- We can override the inherited definition
- Case classes override automatically

Pitfalls in Overriding Equals

- Wrong signature
- Not defining an equivalence relation
- Defining structural equality on mutable datatypes
- Not overriding `hashCode`

The Signature for Equals

```
def equals(that: Any): Boolean
```

Using another signature will result in static overloading.

Not Defining an Equivalence Relation

- Equivalence relations are:
  - Reflexive
  - Symmetric
  - Transitive
- To respect symmetry, we are forced to check that the *dynamic types* of two objects are identical

Ensuring Symmetry

```
class Point(val x: Int, val y: Int) {
  override def equals(that: Any): Boolean = …
}

class ColoredPoint(red: Int, blue: Int, green: Int, x: Int, y: Int)
extends Point(x,y)
```

Ensuring Symmetry

```
class Point(val x: Int, val y: Int) {
  override def equals(that: Any): Boolean = {
    if (this.getClass != that.getClass) false
    else {
      val _point = that.asInstanceOf[Point]
      (_point.x == x) && (_point.y == y)
    }
  }
}

class ColoredPoint(red: Int, blue: Int, green: Int, x: Int, y: Int)
extends Point(x,y)
```

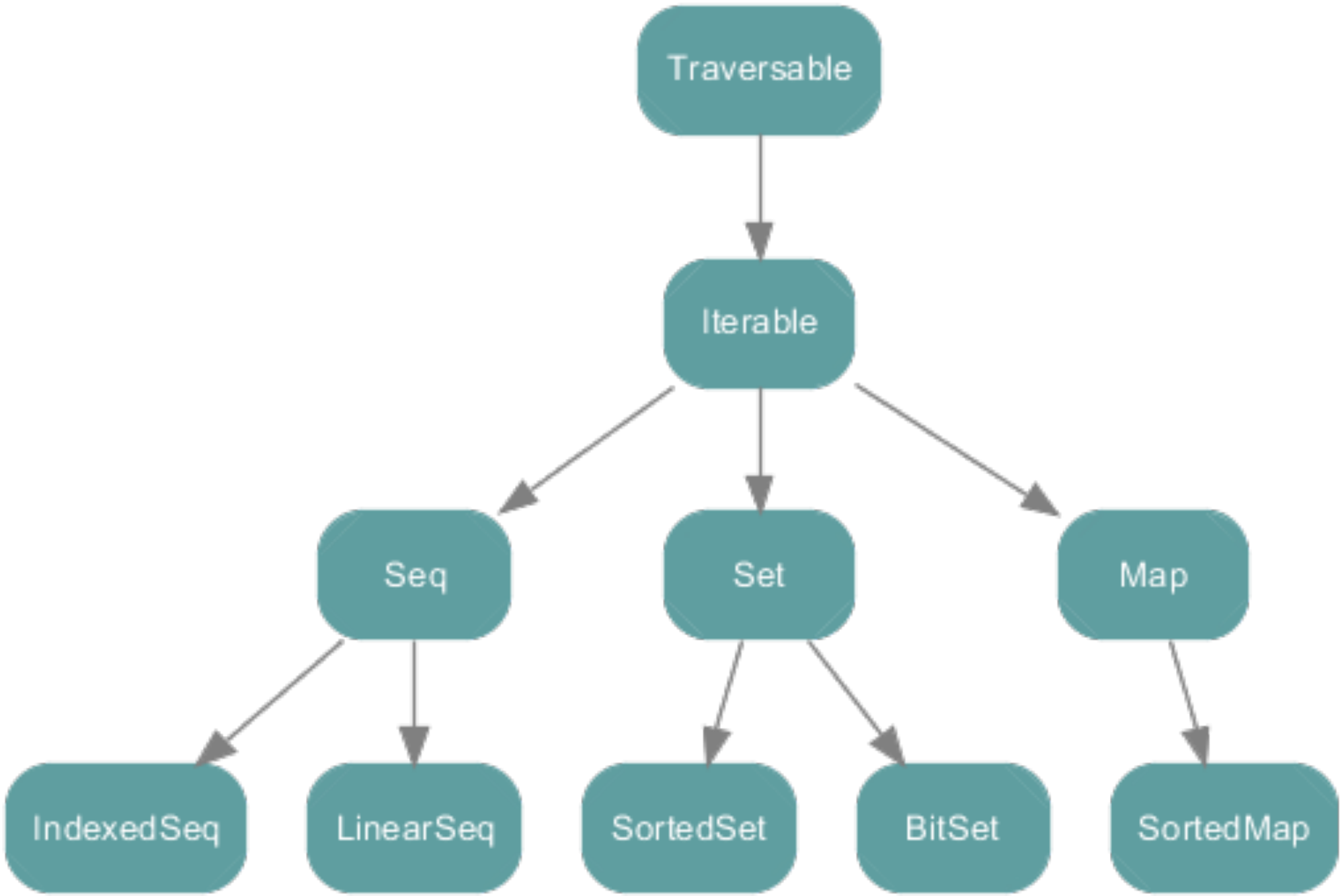
Defining Structural Equality on Mutable Datatypes

Just say no.

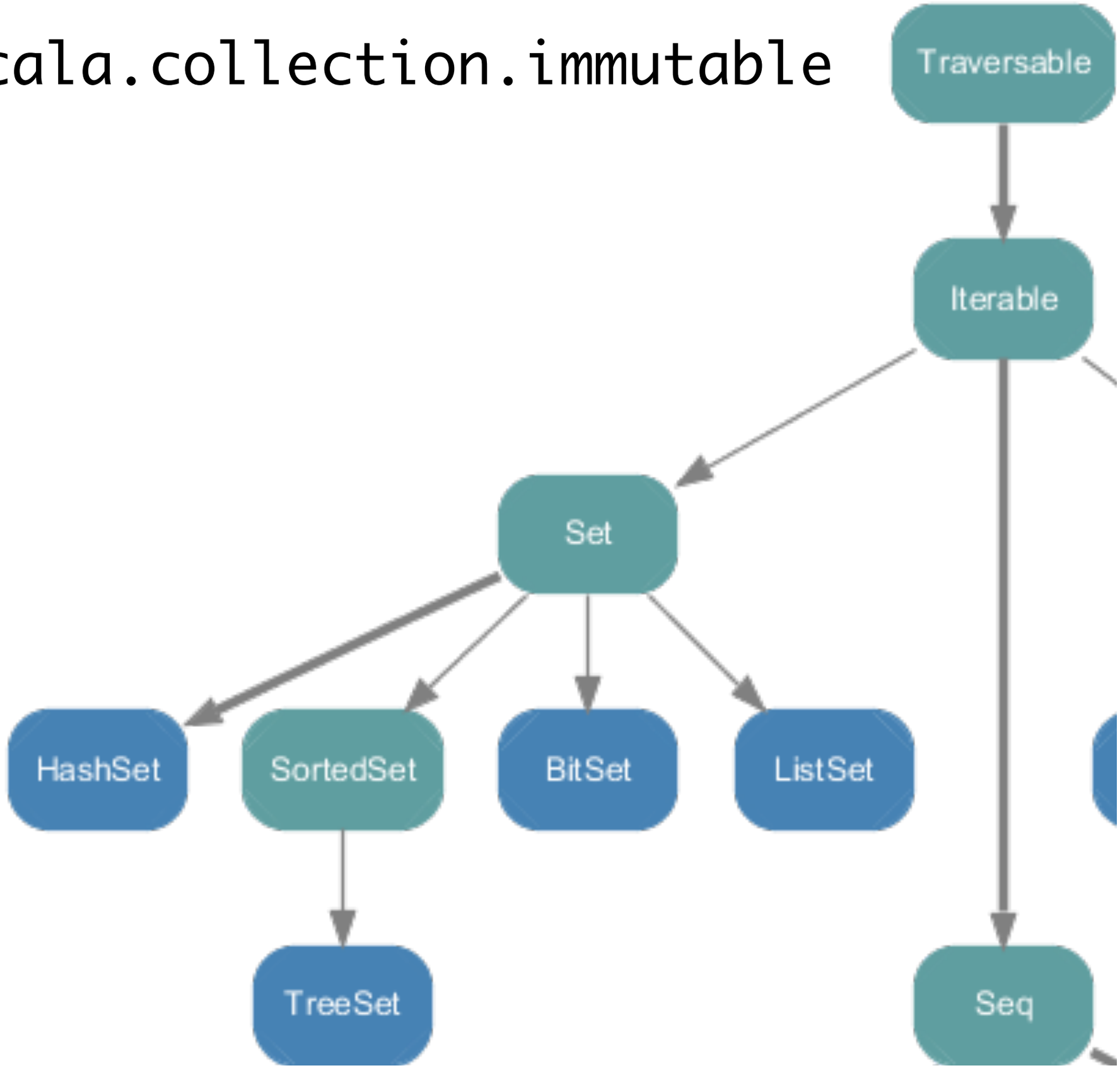
Scala

Collections Classes

Collections in Scala



scala.collection.immutable

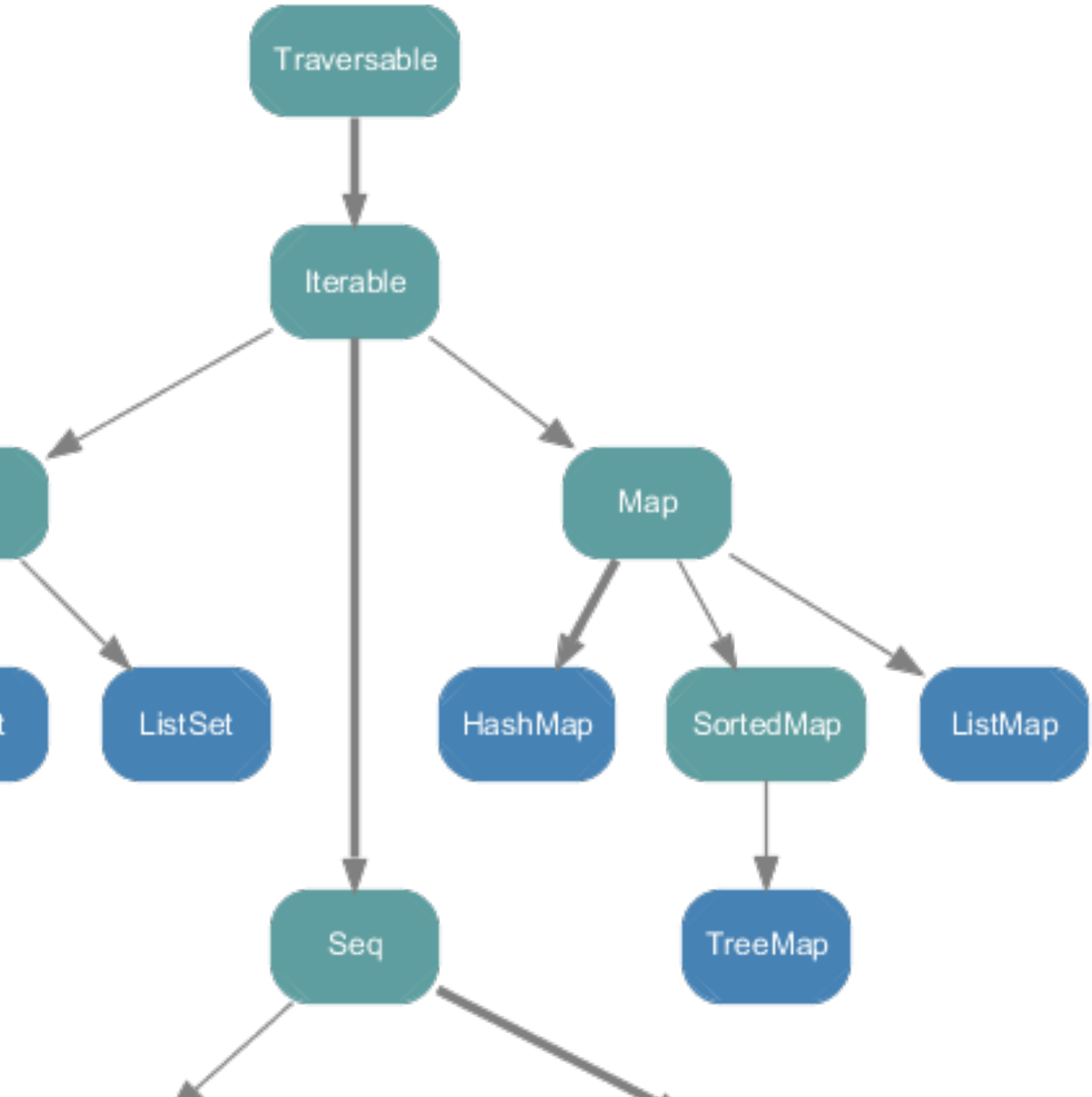


Sorted Sets

- Sorted sets are non-repeating ordered collections of elements

- Canonical implementation is the `TreeSet` implementation (which uses red-black trees)

scala.collection.immutable

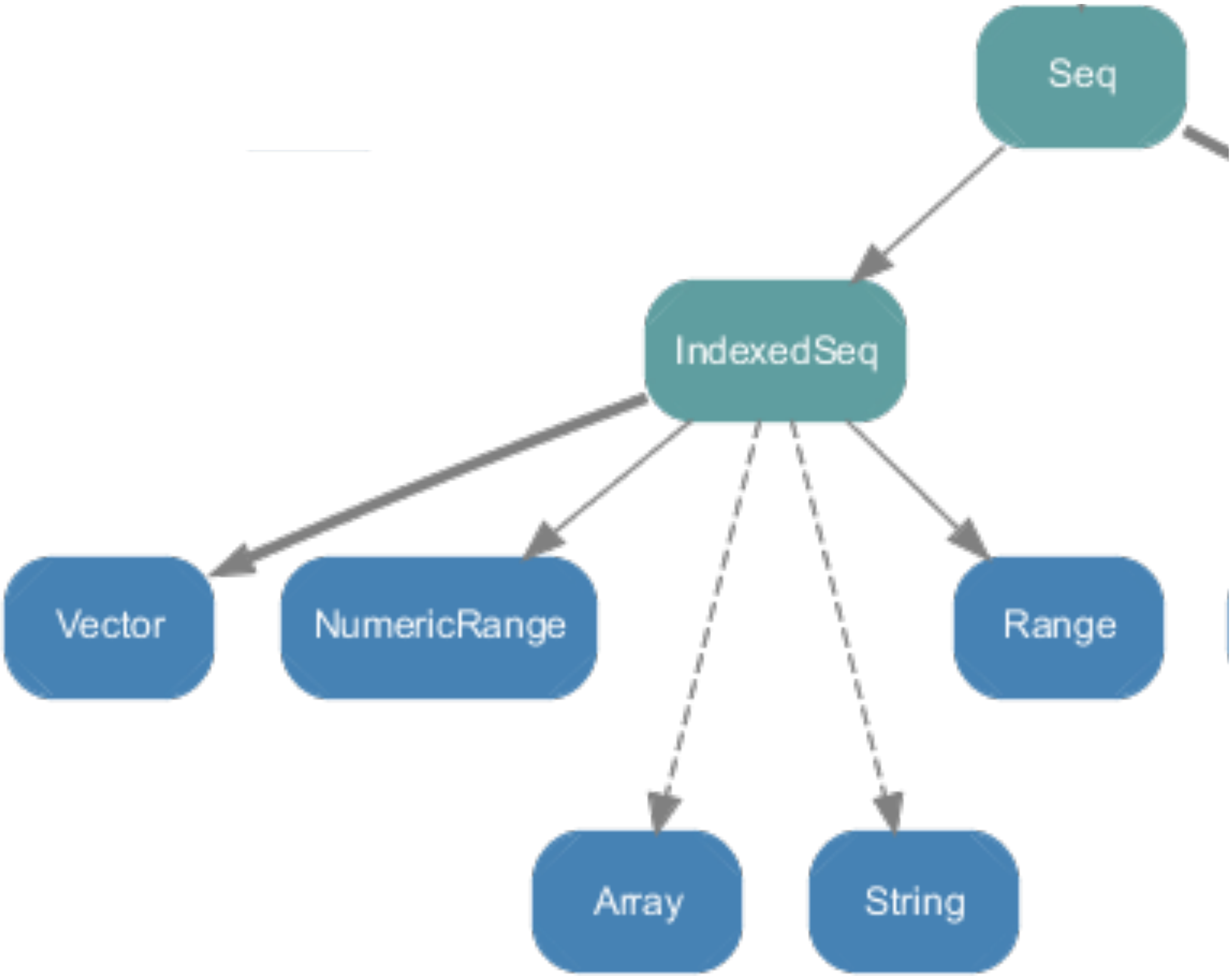


Indexed vs Linear Sequences

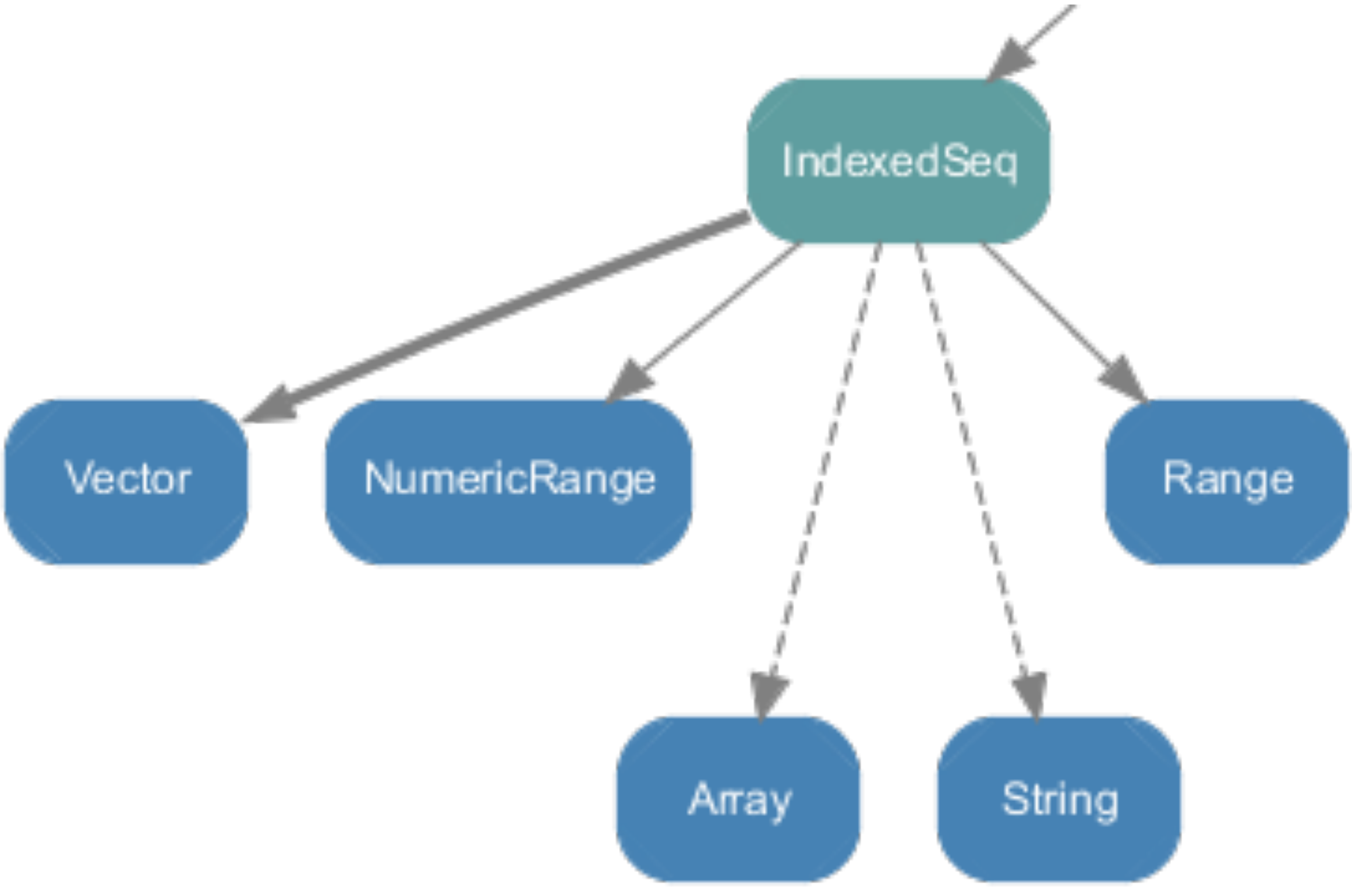
- Linear sequences are intended for recursive descent via `head` and `tail` (as with Lists)

- Indexed sequences are intended for random access to positions (as with Arrays)

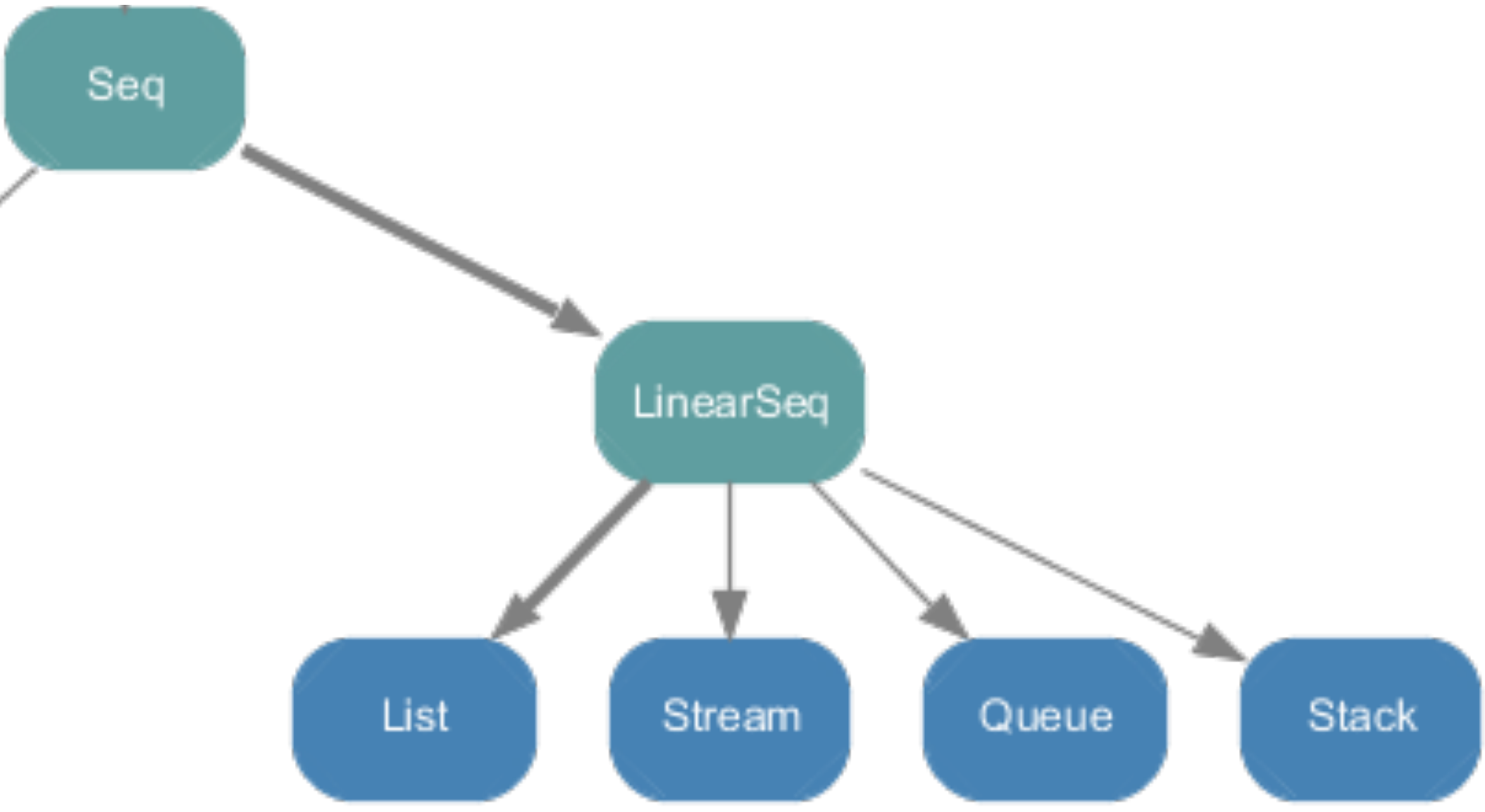
scala.collection.immutable



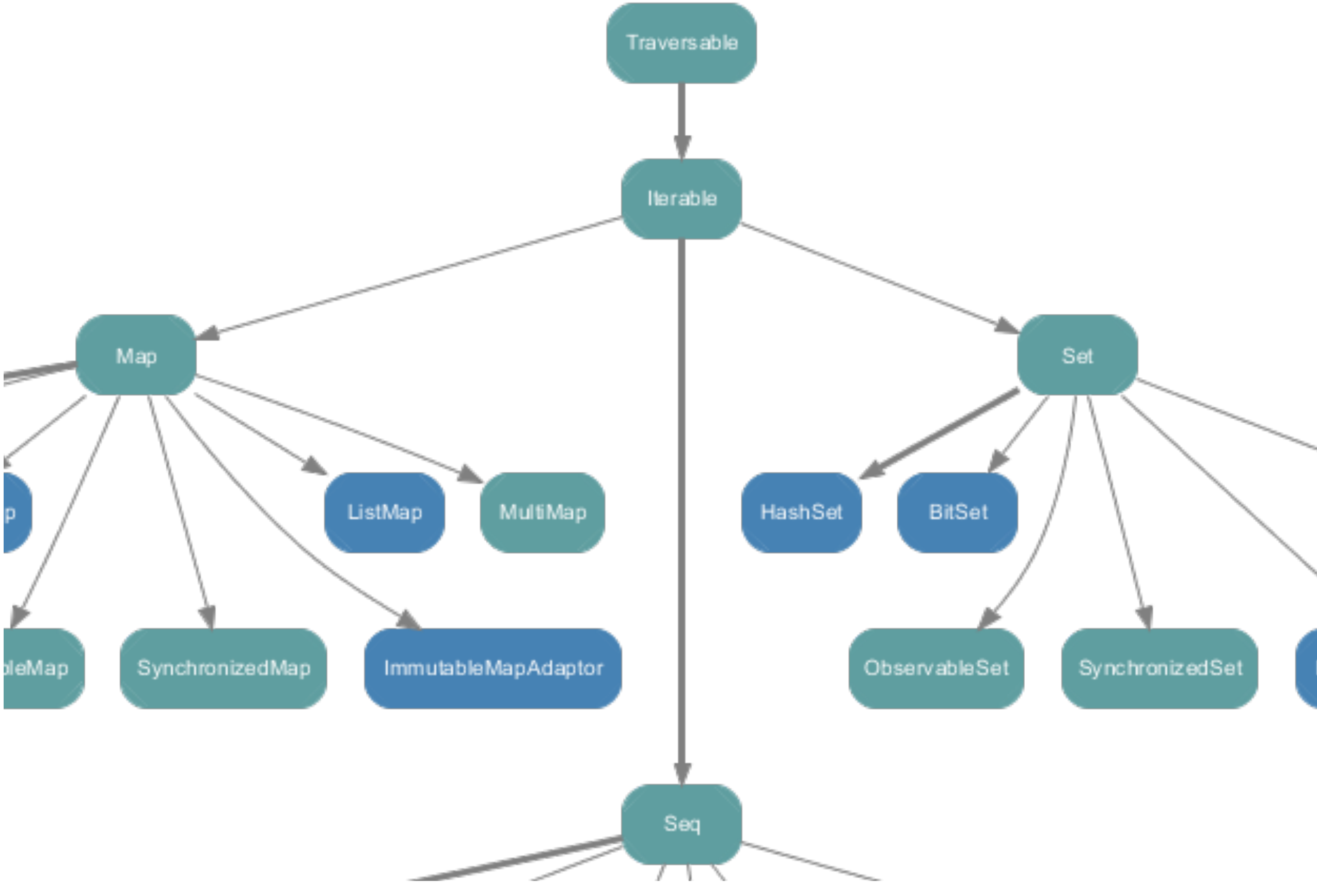
scala.collection.immutable



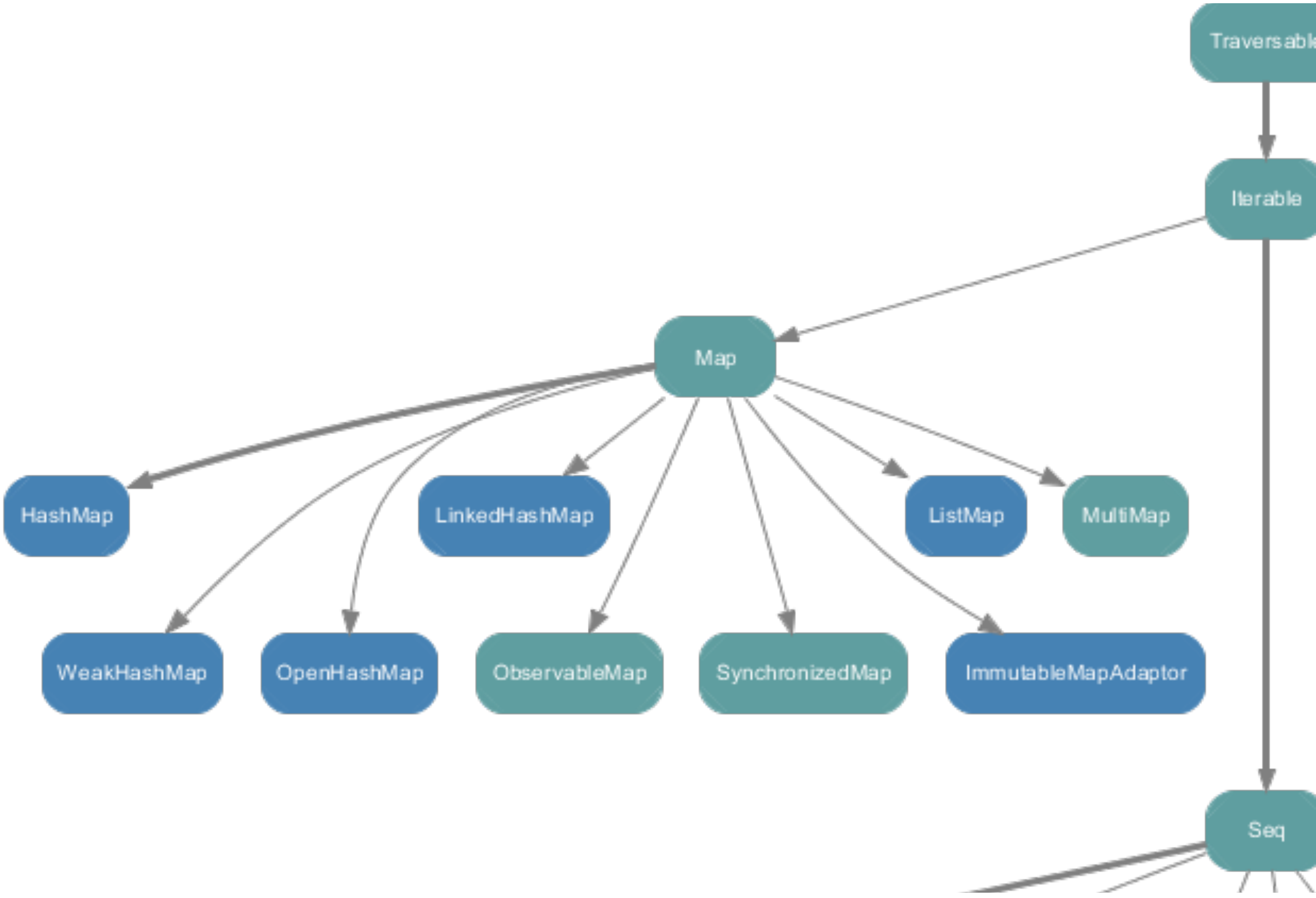
scala.collection.immutable



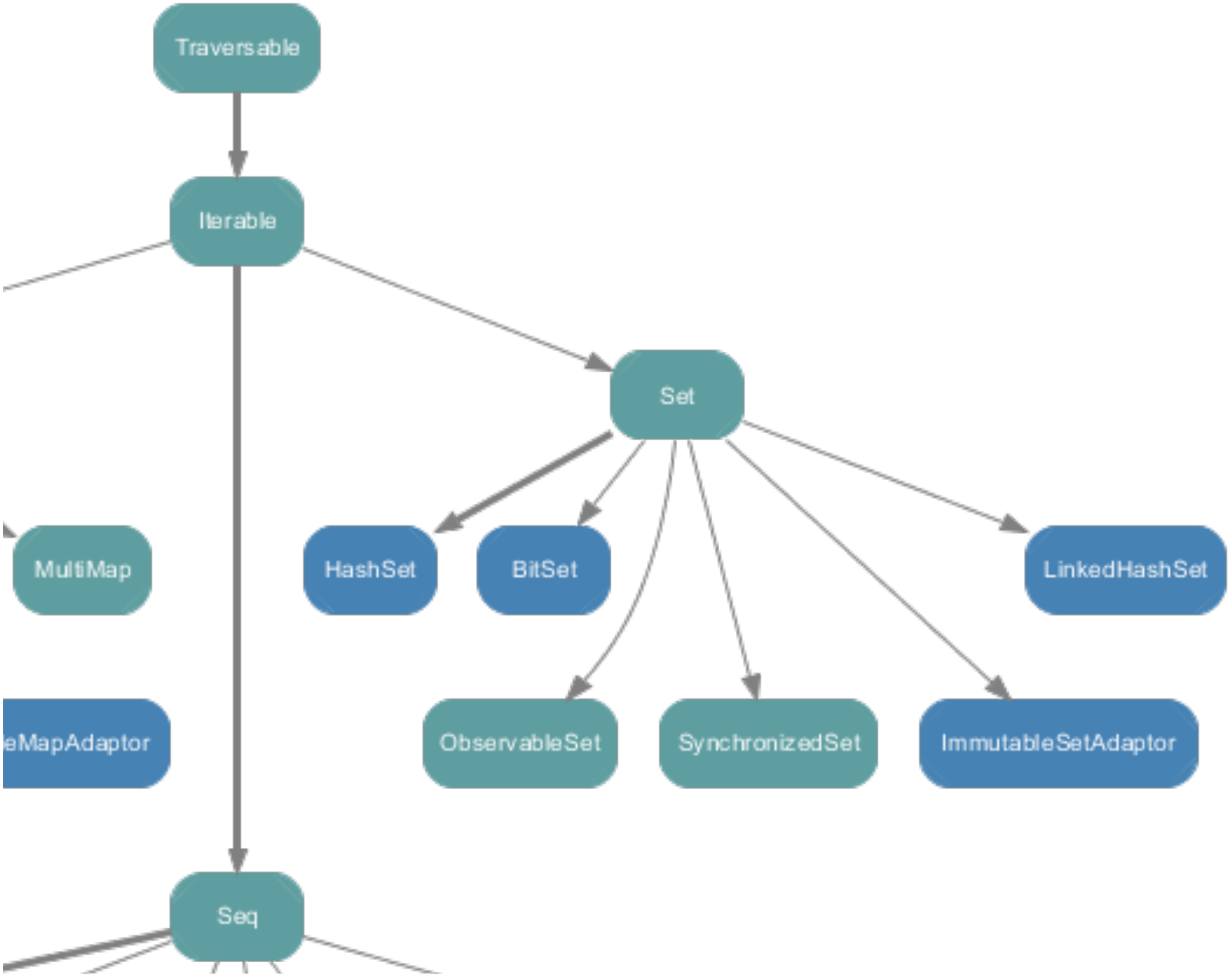
scala.collection.mutable



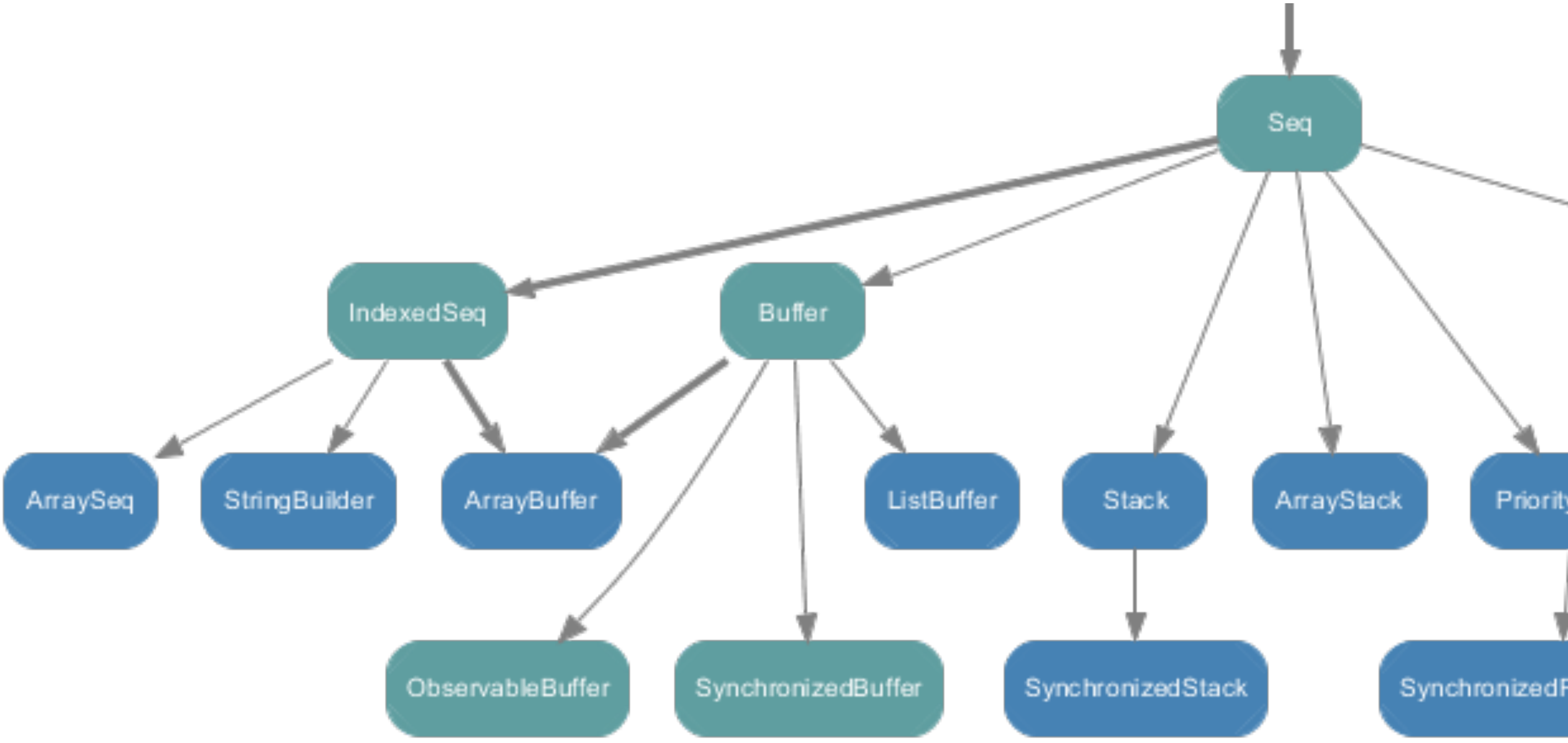
scala.collection.mutable



scala.collection.mutable



scala.collection.mutable



ListBuffers

- In the mutable package
- Constant time prepend and append operations

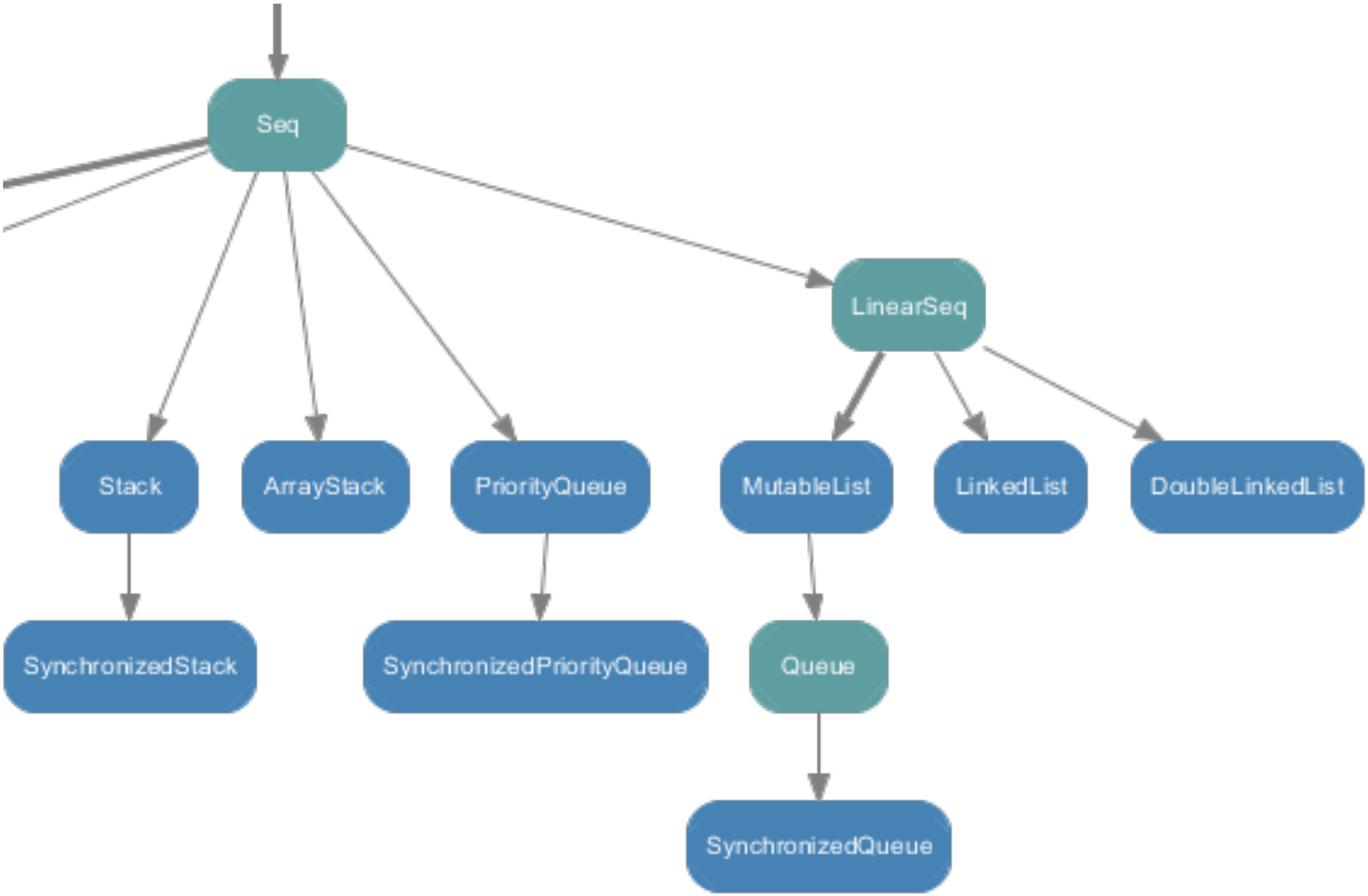
- Append with `+=`
- Prepend with `+=:`
- Obtain a list by invoking `toList`

ArrayBuffers

- Like an array, but with prepend and append

- Prepending and appending on constant time on average but occasionally require linear time

scala.collection.mutable



Trait Traversable

```
def foreach[U](f: Elem => U)
```

Sets and Maps

- Mutable and immutable versions of these collections are available
- By default, you get the immutable versions
- Add and subtract elements using += and -=
- Add and subtract whole collections using ++= and —=

Using Both Mutable and Immutable Datatypes at Once

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

Then mutable variants of a collection type such can be referred to with short qualified names such as:

```
mutable.Set
```

Memoization

Fibonacci Numbers

```
def fib(n: Int): Int = {
  require (n >= 0)
  if (n == 0) 0
  else if (n == 1) 1
  else fib(n - 1) + fib(n - 2)
} ensuring (_ >= 0)
```

Fibonacci Numbers

```
val memoFib: Int => Int =
  memoize {
    (n: Int) => {
      require (n >= 0)
      if (n == 0) 0
      else if (n == 1) 1
      else memoFib(n - 1) + memoFib(n - 2)
    } ensuring (_ >= 0)
  }
```

Memoize

```
def memoize(f: Int => Int) = {
  val table = mutable.Map[Int,Int]()
  (n: Int) =>
    table.getOrElse(n, {
      val result = f(n)
      table += (n -> result)
      result
    })
}
```

Impact of Effects on the Design Recipe

Impact of Effects on the Design Recipe

- Now that functions have effects:
  - The documentation should discuss the observable effects
  - Examples should include observable effects
  - Tests should check that effects occur as expected

Testing Effects

- A common approach to testing in the context of effects is *mocking:*
  - The external objects and APIs our tested code interfaces with are implemented as mock objects that behave just well enough to enable the test
  - Typically, mock objects should perform contained and reversible actions!

Purely Functional State

Rolling a Die

- Suppose we want to implement a function that simulates the rolling of a six-sided die

- The result of calling the function should be a random number from 1 to 6

Rolling a Die

```
def rollDie: Int = {
  val rng = new scala.util.Random
  rng.nextInt(6)
}
```

The call to nextInt will return a value from 0 to 5, not 1 to 6..

Stateful Programs and Debugging

- Because of the state encapsulated in our random number generator:
  - Repeatability of testing is hard
  - Bugs are difficult to reduce
- We would like to use effects when necessary without losing the benefits of referential transparency