Introduction to Scala workshop

About you

About me

What we will NOT be covering

What is functional programming?

Building blocks
Expressions and immutability
Functions

Basic data structures
Case classes
Pattern Matching

Application
Functional combinators

Expressions and immutability

```
scala> 1 + 1

res0: Int = 2
```

```
scala> val a = 1 + 1
a: Int = 2

scala> a = 3
error: reassignment to val
```

```
scala> val a = 10
a: Int = 10

scala> { val a = 5; a }
res3: Int = 5

scala> a
res1: Int = 10
```

```
scala> var b = 1 + 1
b: Int = 2

scala> b = 3
b: Int = 3

scala> b = "hello"
error: type mismatch;
 found   : String("hello")
 required: Int
```

```
scala> var b: Any = 1
b: Any = 1

scala> b = "hello"
b: Any = hello
```

```
scala> val mySet = mutable.Set(
                        "Scala", "Java", “Haskell")

scala> mySet += “Ruby”
res7: mySet.type = Set(Java, Scala, Haskell, Ruby)

scala> mySet.size
res1: Int = 4
```

```
scala> val mySet = mutable.Set(
                        “Scala", “Java","Haskell”)

mySet: scala.collection.Set[String]
     = Set(Scala, Java, Haskell)


scala> mySet = Set(“Scala”, “Java”)

<console>:8: error: reassignment to val
       mySet = Set("Scala", "Java")
```

```
scala> val mySet = Set("Scala", “Java", “Haskell”)
mySet: Set[String]
     = Set(Scala, Java, Haskell)

scala> val updatedSet = mySet + “Ruby”
updatedSet: Set[String]
          = Set(Scala, Java, Haskell, Ruby)

scala> mySet.size
res1: Int = 3

scala> updatedSet.size
res1: Int = 4
```

```
scala> def a = 10
a: Int

scala> a
res26: Int = 10
```

Functions

```
def max(x: Int, y: Int): Int = {
 if(x > y) x
 else y
}

max: (x: Int, y: Int)Int
```

```
def max(x: Int, y: Int): Int = {
 if(x > y) x
 else y
}

max: (x: Int, y: Int)Int

scala> max(2, 4)
res0: Int = 4
```

```
max: (x: Int, y: Int)Int

(Int, Int) => Int
```

```scala
def factorial(n: Int): Int = {
  def go(n: Int, acc: Int): Int =
    if (n <= 0) acc
    else go(n-1, n*acc)
  go(n, 1)
}
```

```
def sumInts(a: Int, b: Int): Int =
  if (a > b) 0
  else a + sumInts(a + 1, b)



def square(x: Int): Int = x * x

def sumSquares(a: Int, b: Int): Int =
  if (a > b) 0
  else square(a) + sumSquares(a + 1, b)



def powerOfTwo(x: Int): Int =
  if (x == 0) 1 else 2 * powerOfTwo(x - 1)

def sumPowersOfTwo(a: Int, b: Int): Int =
  if (a > b) 0
  else powerOfTwo(a) + sumPowersOfTwo(a + 1, b)
```

```
def sum(f: Int => Int, a: Int, b: Int): Int =
 if (a > b) 0
 else f(a) + sum(f, a + 1, b)

def id(x: Int): Int = x

def square(x: Int): Int = x * x

def powerOfTwo(x: Int): Int =
  if (x == 0) 1 else 2 * powerOfTwo(x - 1)


def sumInts(a: Int, b: Int): Int =
  sum(id, a, b)

def sumSquares(a: Int, b: Int): Int =
  sum(square, a, b)

def sumPowersOfTwo(a: Int, b: Int): Int =
  sum(powerOfTwo, a, b)
```

```
(x: Int, y: Int) => x * y
```

```
def sum(f: Int => Int, a: Int, b: Int): Int =
   if (a > b) 0
   else f(a) + sum(f, a + 1, b)


def sumInts(a: Int, b: Int): Int =
   sum(x => x, a, b)
```

```
def sumSquares(a: Int, b: Int): Int =
   sum(x => x * x, a, b)


val powerOfTwo = (x: Int) =>
  if (x == 0) 1 else 2 * powerOfTwo(x - 1)

def sumPowersOfTwo(a: Int, b: Int): Int =
  sum(powerOfTwo, a, b)
```

```
val a = (x: Int, y: Int) => x * y

val a = {
  def f (x: Int, y: Int) = x * y
  f _
}
```

```
object MyModule {

  def max(x: Int, y: Int): Int = {
   if(x > y) x
   else y
  }

 def calculate() = {
    val a = 2
    val b = 10
    max(a, b)
  }
}

scala> val biggest = MyModule.calculate()
```

Definition of KOAN

: a paradox to be meditated upon that is used to train Zen Buddhist monks to abandon ultimate dependence on reason and to force them into gaining sudden intuitive enlightenment

Definition of KOAN

: a paradox to be meditated upon that is used to train Zen Buddhist monks to **abandon** ultimate dependence on **reason** and to force them into gaining sudden intuitive enlightenment

Basic Data structures

Tuple

```
scala> val ab = (5, “Hello")
ab: (Int, String) = (5, Hello)

scala> val abc = (5.5, "World", List(1, 2, 3))
abc: (Double, String, List[Int])
   = (5.5, World, List(1, 2, 3))

scala> ab._1
res9: Int = 5

scala> ab._2
res10: Boolean = true

scala> abc._3
res11: List[Int] = List(1, 2, 3)
```

List

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

scala> val strings = List("a", "b", "c")
strings: List[String] = List(a, b, c)

scala> val numbers2 = 1 :: 2 :: 3 :: Nil
numbers2: List[Int] = List(1, 2, 3)

scala> numbers(0)
res12: Int = 1
```

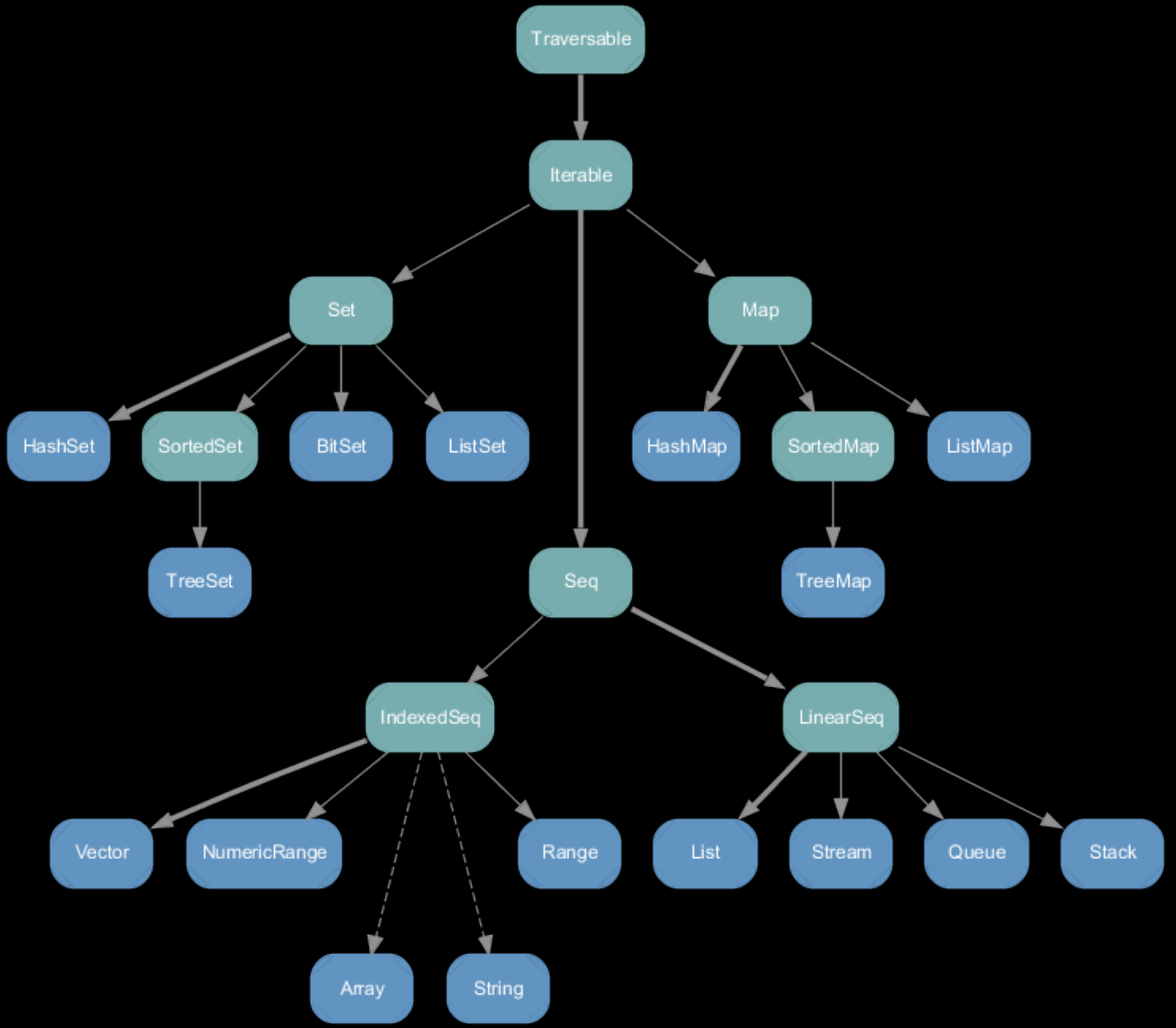
Map

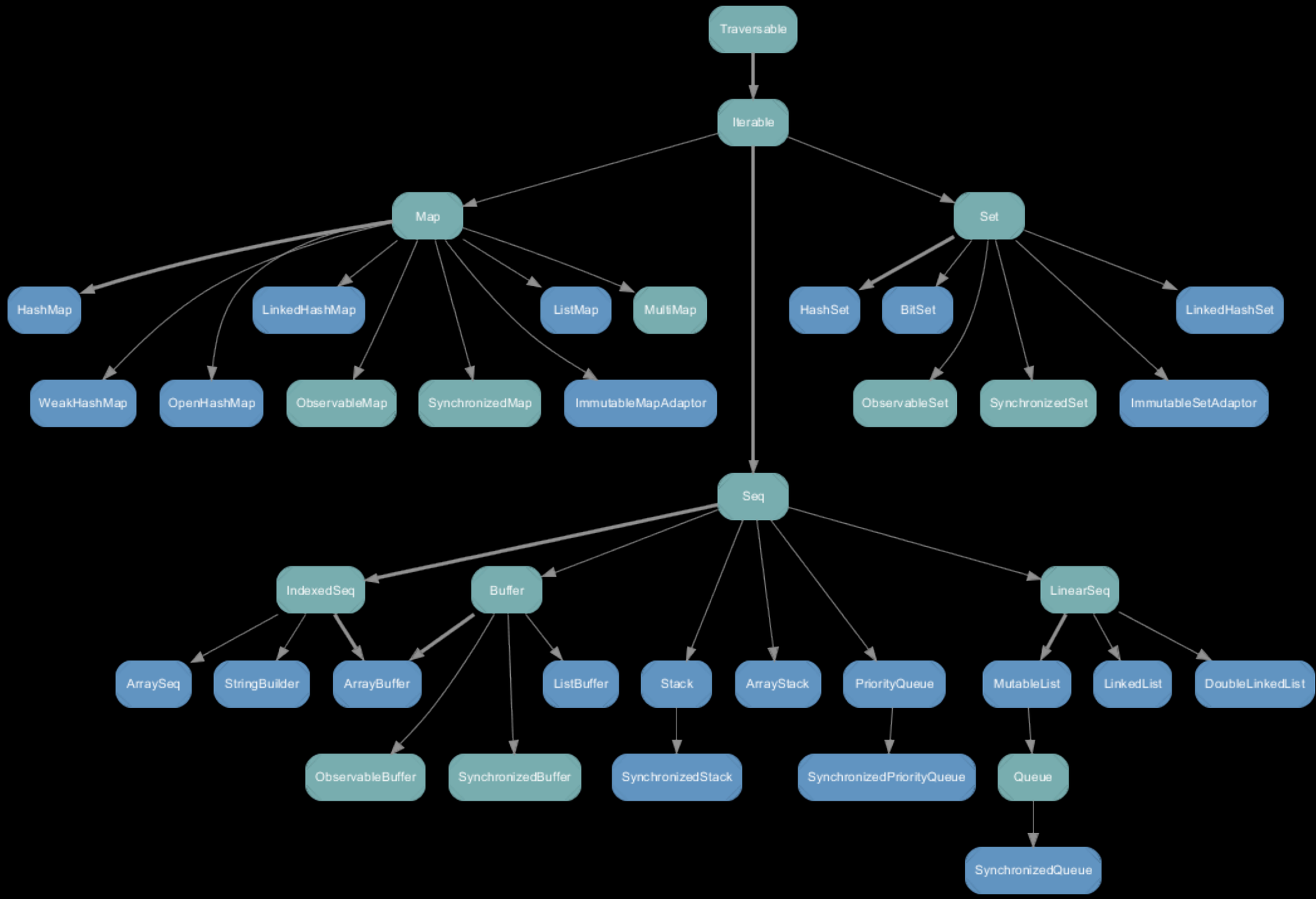
```
scala> val map = Map(1 -> “Hello")
map: immutable.Map[Int,String]
   = Map(1 -> Hello)

scala> val helloWorld = map + (2 -> “World")
helloWorld: immutable.Map[Int,String]
          = Map(1 -> Hello, 2 -> World)
```

```
scala> val world = helloWorld - 1
world: immutable.Map[Int,String]
     = Map(2 -> World)
```

Set

```
scala> Set(1, 1, 2)
res0: scala.collection.immutable.Set[Int]
    = Set(1, 2)
```


Traversable
Iterable
Set
Map
HashSet
SortedSet
BitSet
ListSet
HashMap
SortedMap
ListMap
TreeSet
Seq
TreeMap
IndexedSeq
LinearSeq
Vector
NumericRange
Range
List
Stream
Queue
Stack
Array
String


Traversable
Iterable
Map
Set
HashMap
LinkedHashMap
ListMap
MultiMap
WeakHashMap
OpenHashMap
ObservableMap
SynchronizedMap
ImmutableMapAdaptor
HashSet
BitSet
LinkedHashSet
ObservableSet
SynchronizedSet
ImmutableSetAdaptor
Seq
IndexedSeq
Buffer
LinearSeq
ArraySeq
StringBuilder
ArrayBuffer
ListBuffer
Stack
ArrayStack
PriorityQueue
MutableList
LinkedList
DoubleLinkedList
ObservableBuffer
SynchronizedBuffer
SynchronizedStack
SynchronizedPriorityQueue
Queue
SynchronizedQueue

Case classes

```
case class Person(
   firstname: String,
   lastname: String
)
```

```
case class Person(
  firstname: String,
  lastname:String
)

val p = Person(“Alfred”, “Lysebraate”)

p.firstname
p.lastname
```

```
case class Person(
  firstname: String,
  lastname:String
)

val p = Person(
  “Alfred”,
  “Lysebraate”
)
```

```
val update = p.copy(lastname = “Sandvik”)
```

```
case class Address(
   street: String,
   country: String
)

case class Person(
   firstname: String,
   lastname:String,
   address: Address
)
```

```
case class Address(
  street: String,
  country: String
)

case class Person(
   firstname: String,
   lastname:String,
   address: Address
)
```

```
val p = Person(“Alfred”, “Lysebraate”,
Address(“First Streeth”, “Norway”))

p.address.street
```

```
sealed trait WorkDay
case object Monday extends WorkDay
case object Tuesday extends WorkDay
case object Wednesday extends WorkDay
case object Thursday extends WorkDay
case object Friday extends WorkDay
```

Pattern Matching

```
val anInteger = 5

anInteger match {
   case 1 => println(“Number One”)
   case 2 => println(“Runner up”)
   case _ => println(“Everyone else”)
}
```

```
something match {
   case i: Int => println(“Found a number: ” + i)
   case s: String => println(“Found a string: ” + s)
   case _ => println(“Found unsupported type”)
}
```

```
myObject match {
   case i: Int if i == 0 => println(“It’s Zero”)
   case i: Int if i > 5 => println(“Bigger than 5”)
   case other => …
}
```

```
val acceptedType = myObject match {
  case i: Int => true
  case s: String => true
  case other => false
}
```

```
val userAndPassword = (“alfred”, “secret”)
val organizationKey = “abd4kgo3wgbo2”

login match {
  case (user, password) => check(username, password)
  case authKey => api.check(authKey)
  case _=> illegalLogin()
}
```

```
val a = List(“a”, “b”, “c”)

val res = a match {
  case Nil => “list to short”
  case _ :: Nil => “list to short”
  case _ :: second :: _ => second
  case _ => “list to big”
}

res: String = b
```

```
case class Address(street: String, country: String)
case class Person(u: String, p: String, a: Address)

countrySupported match {
  case Person(_,_, Address(_,country)) => {
   isCountrySupported(country)
  }
  case _ => false
}
```

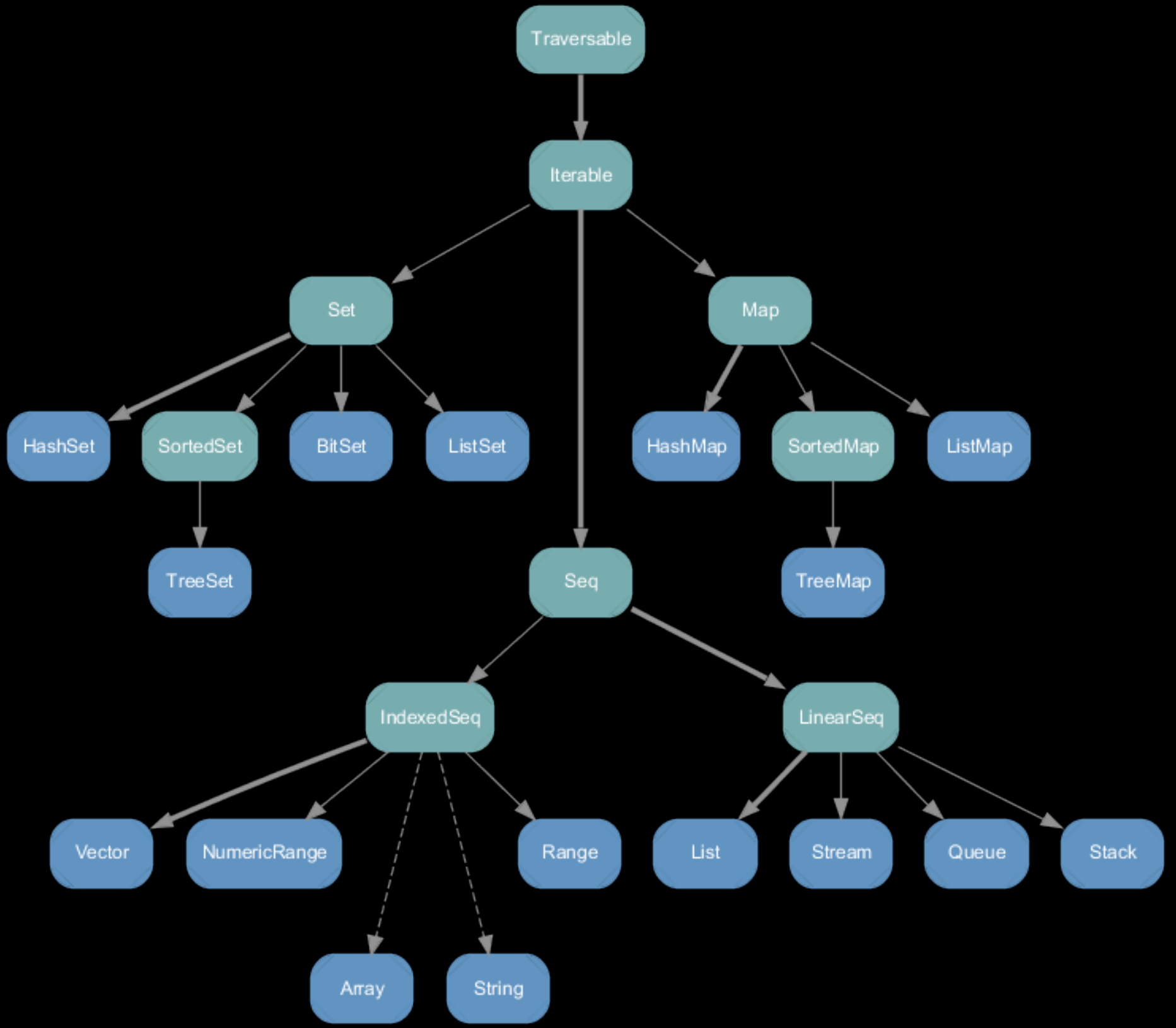
```
workDay match {
  case Monday => …
  case Tuesday => …
  case Wednesday => …
  case Friday => …
}


<console>:13: warning: match may not be exhaustive.
It would fail on the following input: Thursday
       workDay match {
       ^
```

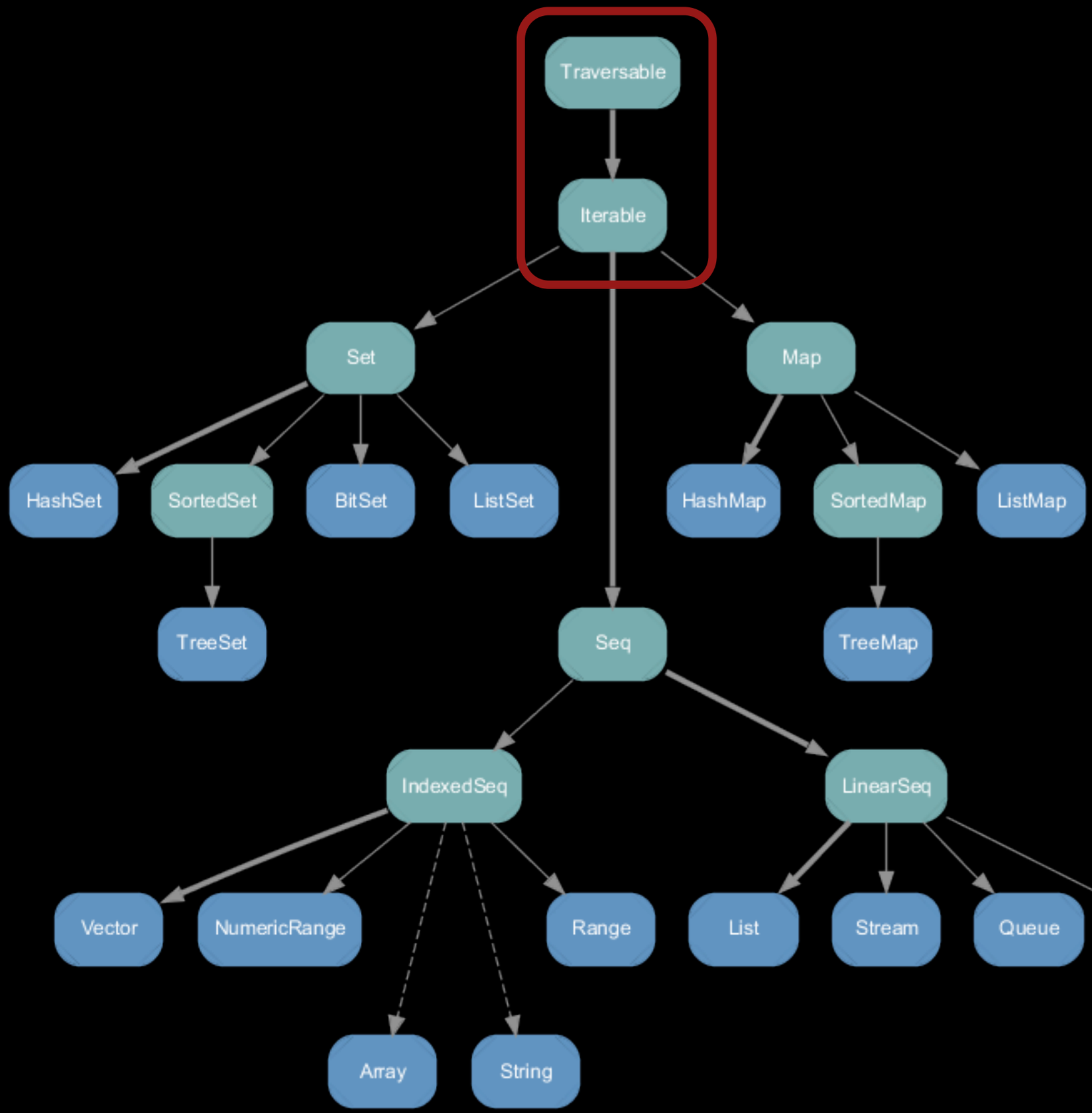
Koans

Functional Combinators

Polymorphic Methods

```
def dup[T](x: T, n: Int): List[T] = {
  if (n == 0) Nil
  else x :: dup(x, n - 1)
}

scala> dup[Int](3, 4))
res25: List[Int] = List(3, 3, 3, 3)

scala> dup("three", 3)
res26: List[String] = List(“three”, “three”, “three”)
```


Traversable
Iterable
Set
Map
HashSet
SortedSet
BitSet
ListSet
HashMap
SortedMap
ListMap
TreeSet
Seq
TreeMap
IndexedSeq
LinearSeq
Vector
NumericRange
Range
List
Stream
Queue
Stack
Array
String


Traversable
Iterable
Set
Map
HashSet
SortedSet
BitSet
ListSet
HashMap
SortedMap
ListMap
TreeSet
Seq
TreeMap
IndexedSeq
LinearSeq
Vector
NumericRange
Range
List
Stream
Queue
Stack
Array
String

Map

```
trait List[+A]{
  def map[B](f:A => B):List[B] = ...
}

scala> val list = List(1, 2, 3, 4)

scala> list.map(element => element * 2)
res27: List[Int] = List(2, 4, 6, 8)
```

Filter & FilterNot

```
trait List[+A]{
  def filter(f:A => Boolean):List[A] = ...
}

scala> val list = List(1, 2, 3, 4)

scala> list.filter(element => element % 2 == 0)
res28: List[Int] = List(2, 4)

scala> list.filterNot(element => element % 2 == 0)
res29: List[Int] = List(1, 3)
```

FlatMap

```
trait List[+A]{
  def flatMap[B](f:A => Traversable[B]):List[B] = ...
}

case class Person(val pets:List[String])

scala> val family = List(Person(List("Dog", "Cat")),
Person(List("Fish")))
```

```
scala> val familyPets = family.map(p => p.pets)
familyPets: List[List[String]]
          = List(List(Dog, Cat), List(Fish))
```

FlatMap

```
trait List[+A]{
  def flatMap[B](f:A => Traversable[B]):List[B] = ...
}

case class Person(val pets:List[String])

scala> val family = List(Person(List("Dog", "Cat")),
Person(List("Fish")))
```

```
scala> val familyPets = family.map(p => p.pets)
familyPets: List[List[String]]
          = List(List(Dog, Cat), List(Fish))

scala> familyPets.reduce(_ ++ _)
         res7: List[String] = List(Dog, Cat, Fish)
```

FlatMap

```
trait List[+A]{
  def flatMap[B](f:A => Traversable[B]):List[B] = ...
}

case class Person(val pets:List[String])

scala> val family = List(Person(List("Dog", "Cat")),
Person(List("Fish")))

scala> val familyPets = family.flatMap(p => p.pets)
familyPets: List[String] = List(Dog, Cat, Fish)

scala> val strings = List("a,b,c", “d,e,f”
                          .flatMap(s => s.split(“,”))

strings: List[String] = List(a, b, c, d, e, f)
```

And More

```
val first10 = list.take(10)

val dropped = list.drop(5)

val containsTwo = List(1, 2, 3).contains(2)

val families :Map[String, List[Person]] =
persons.groupBy(_.lastname)
```

```
val firstname:Map[String, List[String]] =
  families.mapValues(persons =>
persons.map(_.firstname))

val sorted:Seq[(String, List[String])] =
  firstname.toSeq.sortBy(_._1.size)
```

Koans

Summary

Immutability

Functions as first-class citizens &
Higher order functions

Immutable functional data structures and functions that do not change state

Pattern matching is a tool to control flow and extract data

Few abstractions and operations can solve a whole lot of problems

Come join us at flatMap(Oslo)