# Class 5: Scala Collections for Spark Programming

New York University

**Summer 2017**

## Agenda

1. Review
2. Scala Collections for Spark Programming
3. **Sets**: Creating a Collection of Unique Elements
4. **Lists and ListBuffers**: Fast Access to Head of Collection
5. **Arrays**: Fast Access to Arbitrary Elements
6. **Maps**: Fast Access with a Key
7. Common Collection Type Conversions
8. Midterm Exam Review

## Agenda

1. Review
2. Scala Collections for Spark Programming
3. **Sets**: Creating a Collection of Unique Elements
4. **Lists and ListBuffers**: Fast Access to Head of Collection
5. **Arrays**: Fast Access to Arbitrary Elements
6. **Maps**: Fast Access with a Key
7. Common Collection Type Conversions
8. Midterm Exam Review

- We've looked at Tuples in the previous section
  - Tuples are not part of the collection hierarchy of Scala
- Collections in Scala are defined by classes that inherit methods from parent classes forming a *Collections Hierarchy*
- A Collection is an object instantiated from a Collection class
- *Knowing where Collection classes reside in the Collections Hierarchy helps distinguish the purpose and features of different kinds of Collections*

- A collection in package **`scala.collection`** can be either mutable or immutable
  - For instance, **`collection.IndexedSeq[T]`** is a superclass of both:

    **`collection.immutable.IndexedSeq[T]`** and

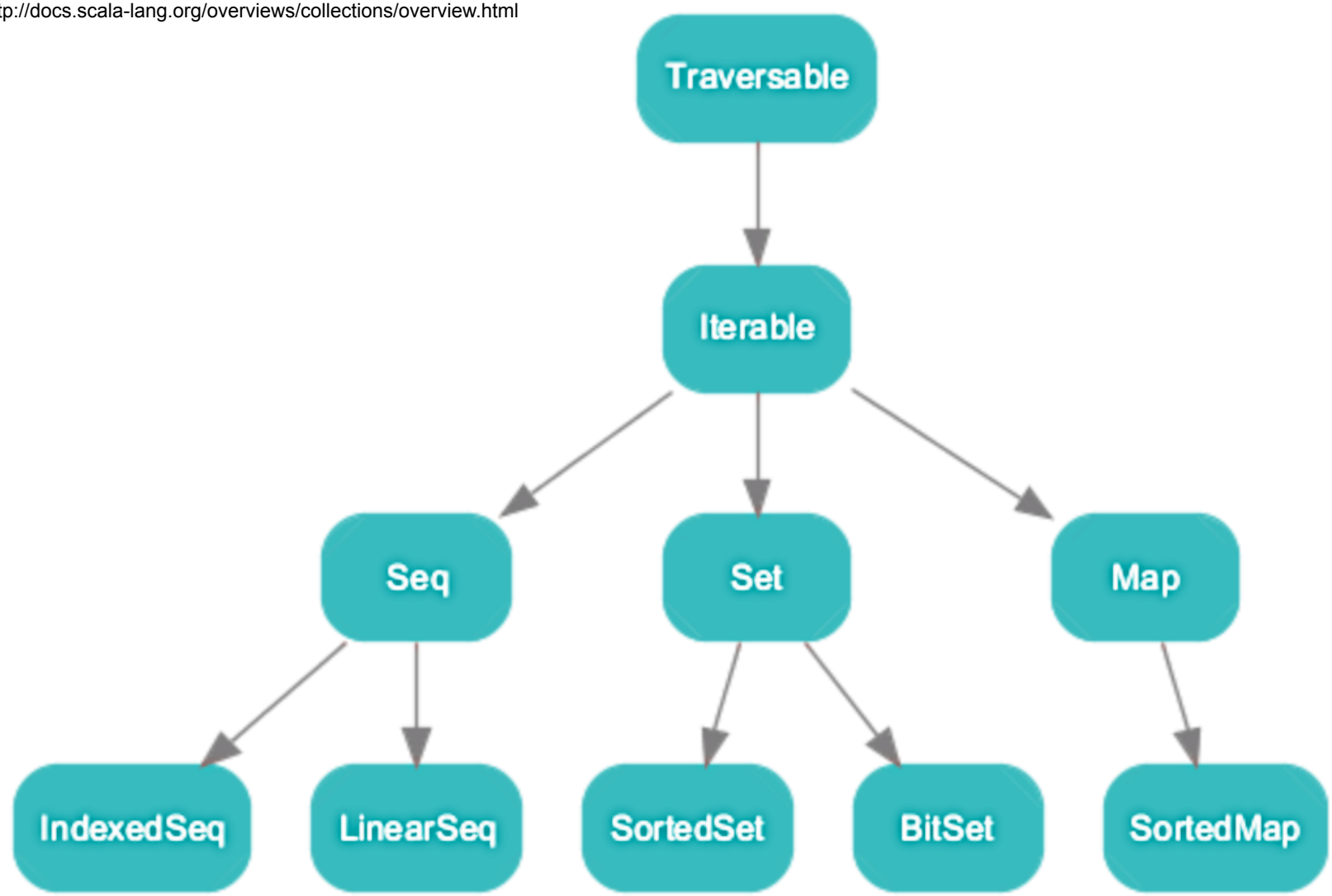
    **`collection.mutable.IndexedSeq[T]`**

- Generally, the root collections in package **`scala.collection`** define the same interface as the immutable collections
  - The mutable collections in package **`scala.collection.mutable`** add some *side-effecting* modification operations to this immutable interface

- The difference between root collections and immutable collections:
  - Clients of an immutable collection have a *guarantee* that nobody can mutate the collection
  - Clients of a root collection only *promise* not to change the collection themselves

- Even though the static type of such a collection provides no operations for modifying the collection, it might still be possible that the run-time type is a mutable collection which can be changed by other clients

http://docs.scala-lang.org/overviews/collections/overview.html



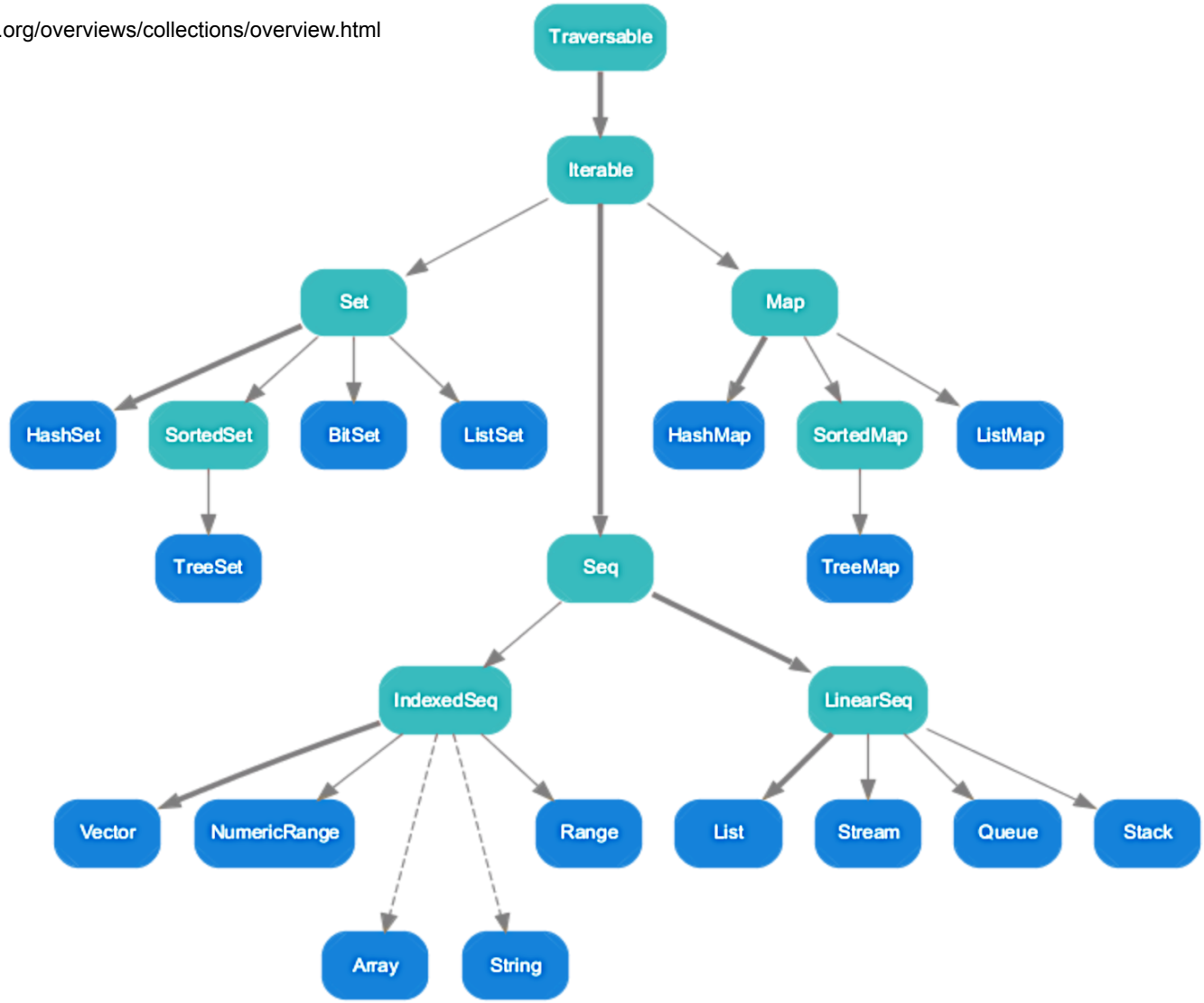
- In Scala, there are a large number of collection classes available and Spark leverages many of them
  - Collection classes are optimized for use in particular circumstances
    - For fast head or tail access
    - For fast update

- Collection classes vary in the methods they support
  - Immutable Collection classes are defined in package **`scala.collection.immutable`**
  - Mutable Collection classes are defined in package **`scala.collection.mutable`**

- Immutable Collections
  - Cannot be updated or extended in place
  - Immutable collections never change
  - There are operations that simulate additions, removals, or updates, but those operations will in each case return a *new collection* and leave the old collection unchanged

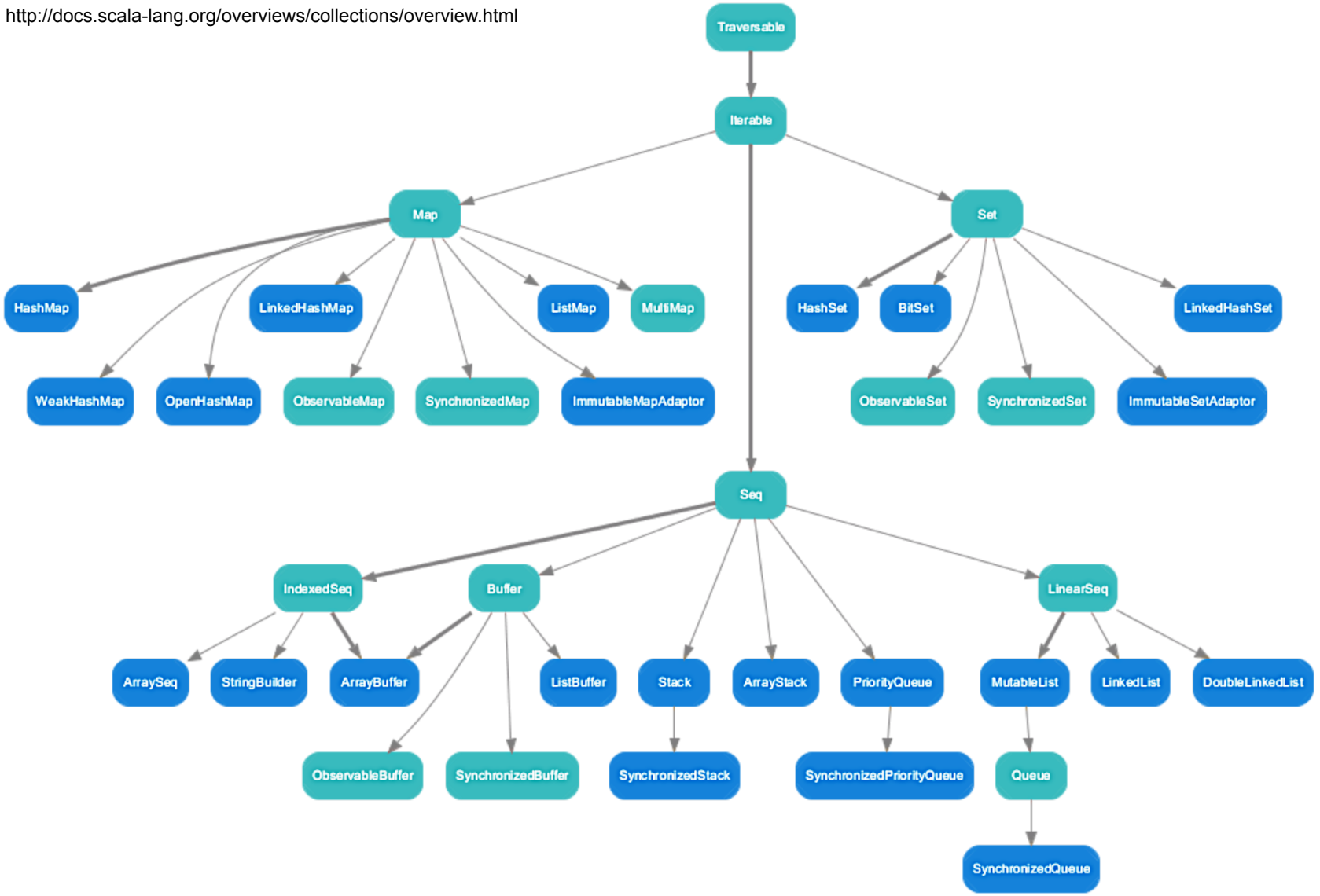
- Immutable Collections
  - A collection in package scala.collection.immutable is guaranteed to be immutable for everyone
  - Such a collection will never change after it is created
  - Therefore, you can rely on the fact that accessing the same collection value repeatedly at different points in time will always yield a collection with the same elements
  - Remember that using immutable objects and collections is preferred when working in Scala and Spark

http://docs.scala-lang.org/overviews/collections/overview.html



- Mutable Collections
  - *Can* be updated or extended in place
  - This means you can change, add, or remove elements of a collection *as a side effect*
  - A collection in package **`scala.collection.mutable`** has some operations that change the collection in place

http://docs.scala-lang.org/overviews/collections/overview.html



- By default, Scala always picks immutable collections
- For example, if you create a collection of type **`Set`** without any explicit mention of mutability or immutability desired, the **`Set`** will be of an immutable type
- If you choose an **`Iterable`**, an immutable **`Iterable`** collection is created
- This is due to the default bindings imported from the Scala package
- **To get the mutable default versions, you need to write explicitly `collection.mutable.Set`, or `collection.mutable.Iterable`**

- To use both mutable and immutable versions of collections, use:

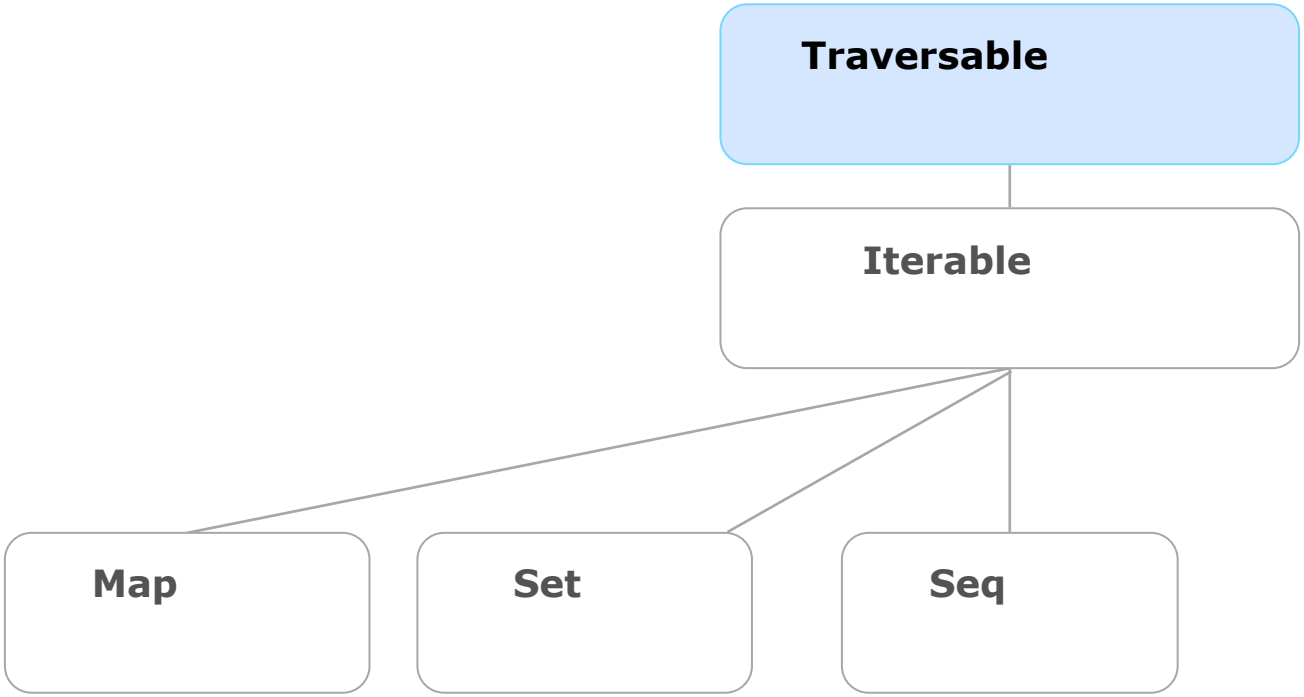
```
import scala.collection.mutable
```

- Then, to create a mutable **`Set`** use **`mutable.Set`**
- To create an immutable **`Set`**, simply specify it as **`Set`**

- **`Traversable`** provides the very important **`foreach`** method which facilitates parallel and distributed processing
  - **`foreach`** performs a specified action on all members of the collection
- Scala will apply the function you supply to **`foreach`** to each element
  - *Allows the platform to parallelize processing and improve performance*

```
val modelTrav = Traversable("MeToo", "Ronin", "iFruit")

modelTrav.foreach(println)
> MeToo
> Ronin
> iFruit
```

The **`Traversable`** **`foreach`** method receives a function as a parameter; for example **`println`**, which will be called once for each element in the collection.



While you are guaranteed that all elements of the collection will be processed, **you are not guaranteed of an order of processing**

- It's easy to imagine the work being allocated amongst four processors (threads), each producing results in parallel
- And taking that one step further with Spark, where processing spans servers

```
val modelTrav =
Traversable("MeToo","Ronin","Sorrento","Titanic","iFruit")
> modelTrav: Traversable[String] = List(MeToo, Ronin,
Sorrento, Titanic, iFruit)

modelTrav.foreach(println)
> MeToo
> Ronin
> Sorrento
> Titanic
> iFruit
```

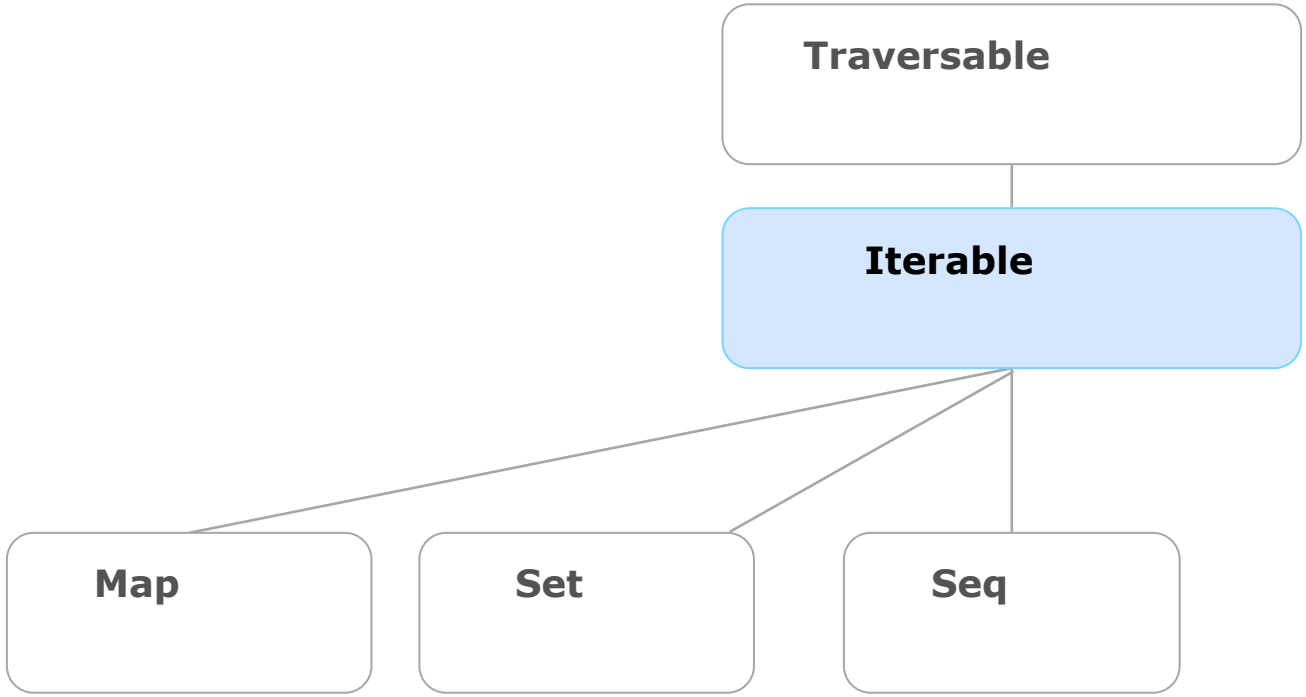
The **`Traversable.foreach`** method receives a function as a parameter, for example **`println`**, which will be called once for each element in the collection

- All Collection types derive from the **`Traversable`** abstract type
- You pass your function in, and Scala is responsible for applying it over the entire collection
  - Further, Spark extends Scala's reach across a cluster of machines

- **`Iterable`** adds the ability to iterate through each element, one at a time
  - Data is resident in memory only as it is used

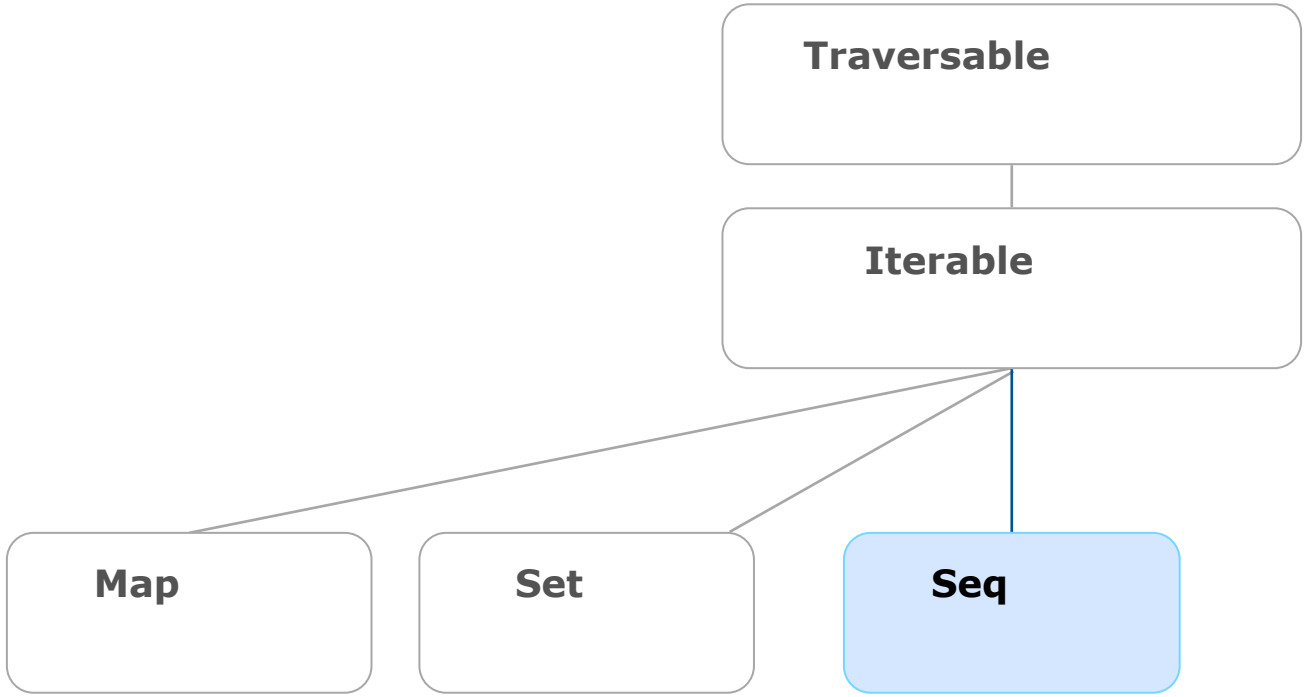
```
val models = Iterable("MeToo", "Ronin", "iFruit")
> models: Iterable[String] = List(MeToo, Ronin, iFruit)

val modelIter = models.iterator
> ModelIter: Iterator[String] = non-empty iterator

modelIter.next
> String = MeToo

modelIter.next
> String = Ronin

modelIter.next
> String = iFruit
```

The **`iterator`** method returns an **`Iterator`** object, which provides a way to traverse each element in sequence, *one time*



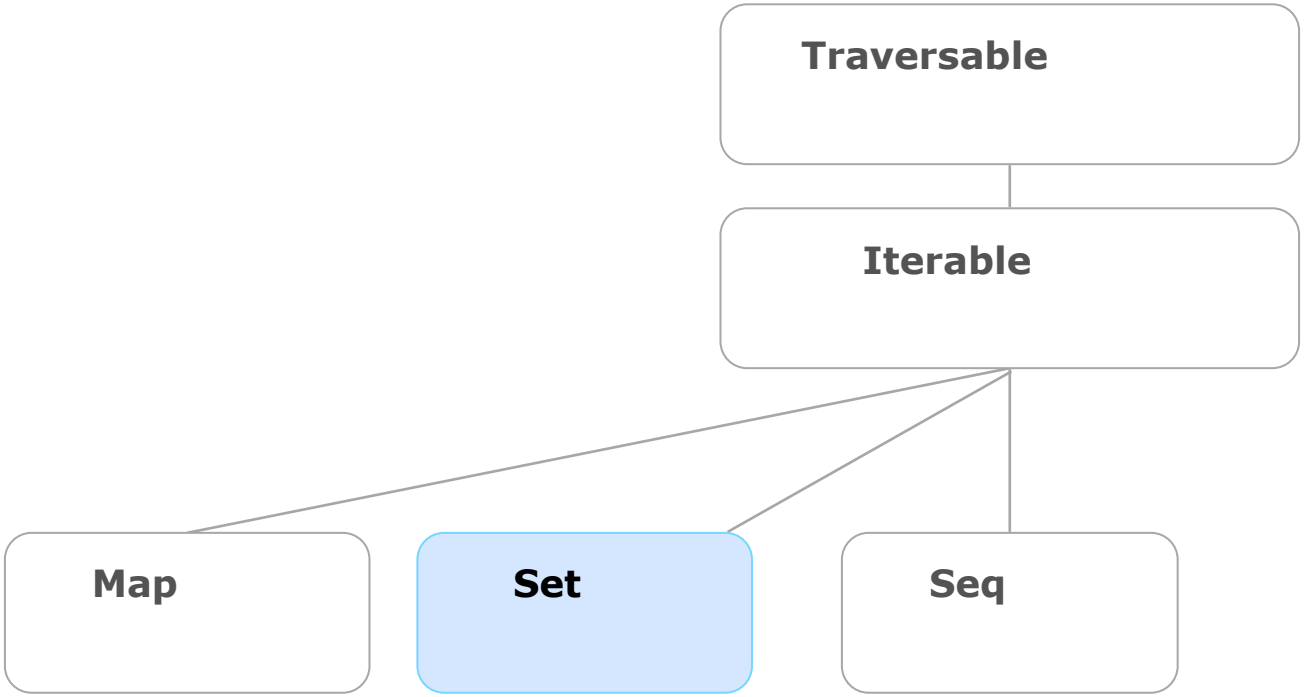
- **`Seq`** adds the ability to access each element at a fixed offset (index)
- First element is at index 0
- **`Seq(n)`** returns the value of the element at offset ***`n`***

```
val mySeq = Seq("MeToo", "Ronin", "iFruit")
> mySeq: Seq[String] = List(MeToo, Ronin, iFruit)

mySeq(1)
> String = Ronin
```



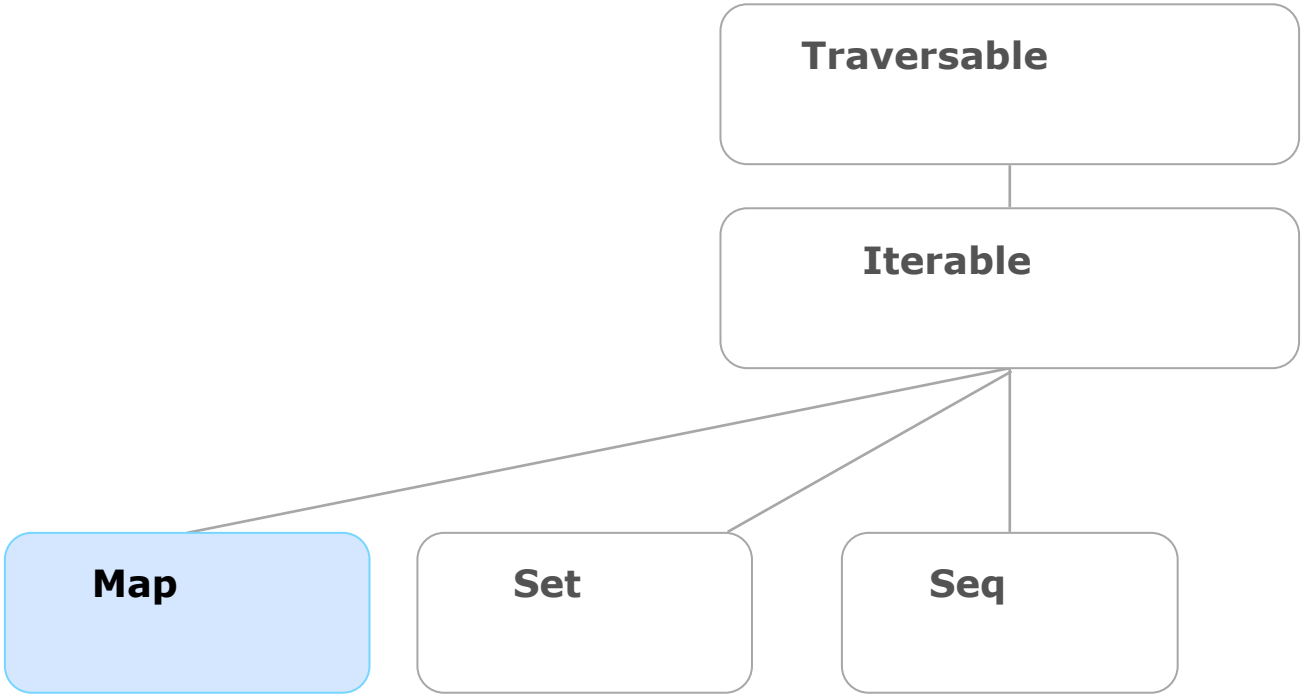
- **`Set`** removes duplicates
- Does not change ordering
- **`Set(`*`value`*`)`** returns **`true`** or **`false`**

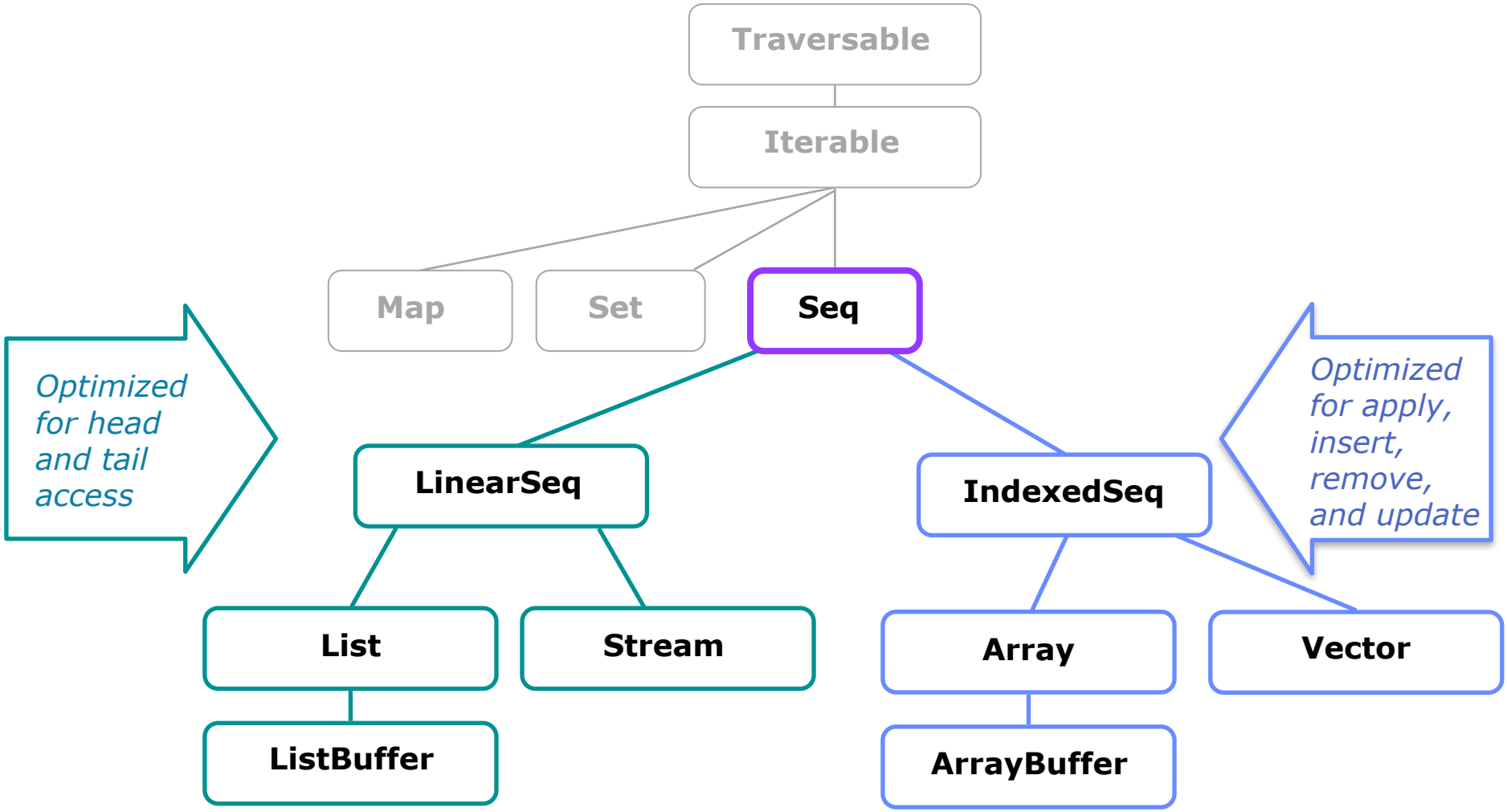
```
val mySet = Set("MeToo", "Ronin", "iFruit")
> mySet: scala.collection.immutable.Set[String] =
  Set(MeToo, Ronin, iFruit)

mySet("Banana")
> Boolean = false
```



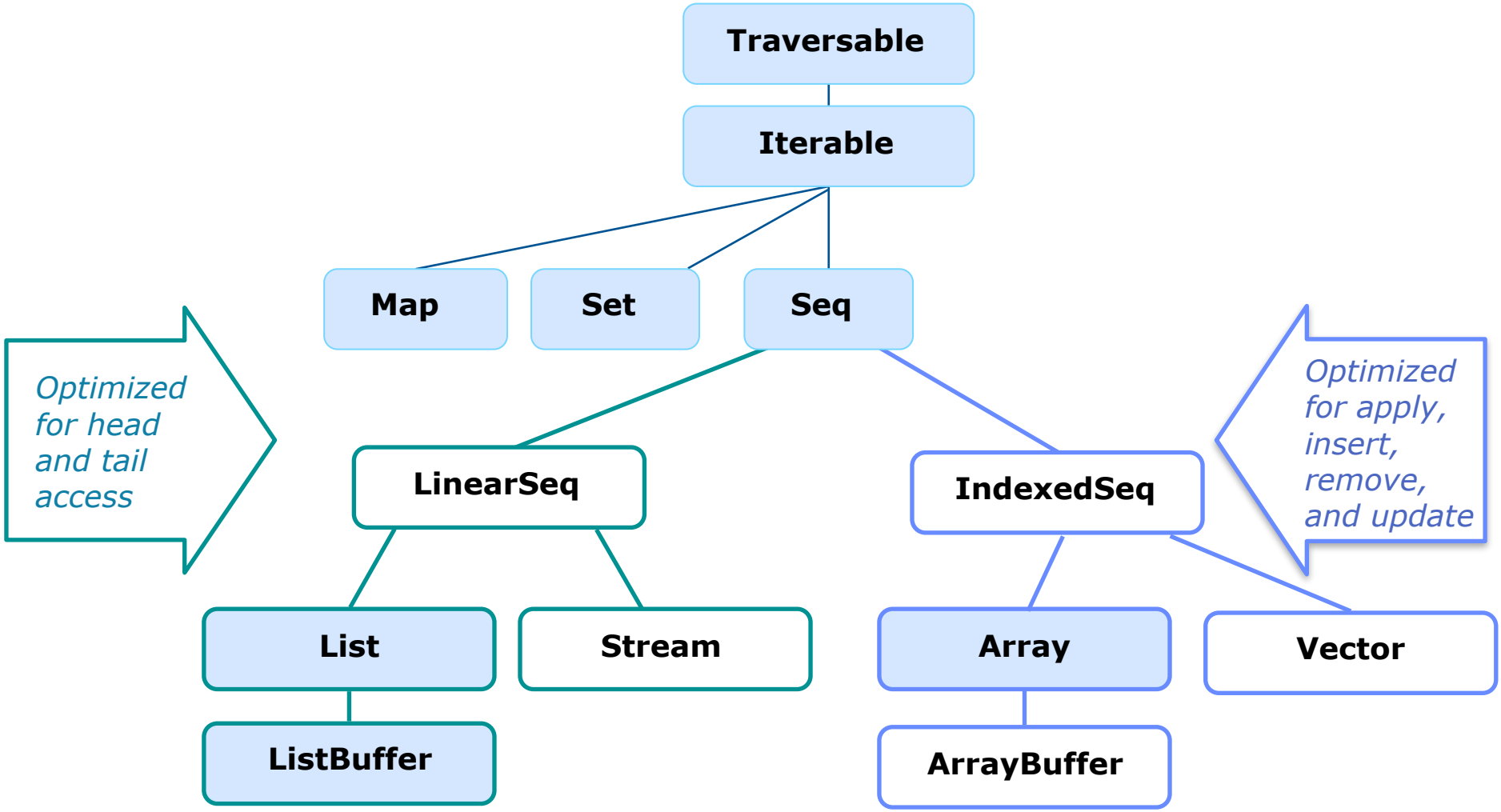
- **`Map`** stores (key → value) pairs

```
val wifiStatus = Map(
"disabled"  -> "Wifi off",
"enabled"   -> "Wifi on but disconnected",
"connected" -> "Wifi on and connected")

wifiStatus("enabled")
> String = Wifi on but disconnected
```






*Buffers are mutable versions – supporting insert, remove, append methods*

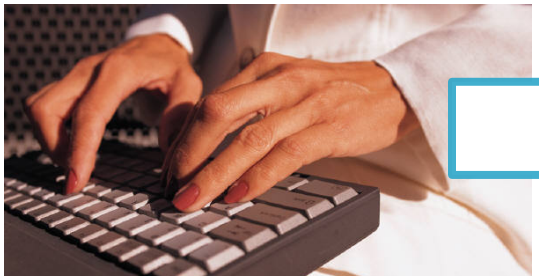



*Buffers are mutable versions – supporting insert, remove, append methods*

- We have covered the different collection types, but the elements of a collection also have a type
- Element type may be specified explicitly or inferred

```
val myMap: Map[Int,String] = Map(1 -> "a", 2 -> "b")

val myMap = Map(1 -> "a", 2 -> "b")
```

- Scala collections include methods for processing all items in a collection without returning each item to the calling program
- By processing all items and only returning the result, Scala can optimize the program for distributed processing



```
collection.foreach()
```

**Platform**

***How to***
- ***iterate***
- ***process data***

## Agenda

1. Review
2. Scala Collections for Spark Programming
3. **Sets**: Creating a Collection of Unique Elements
4. **Lists and ListBuffers**: Fast Access to Head of Collection
5. **Arrays**: Fast Access to Arbitrary Elements
6. **Maps**: Fast Access with a Key
7. Common Collection Type Conversions
8. Midterm Exam Review

- A **`Set`** is an **`Iterable`** that contains no duplicate elements

```
val mySet = Set("Titanic", "Sorrento", "Ronin",
"Titanic", "Sorrento", "Ronin")
> mySet: scala.collection.immutable.Set[String] =
Set(Titanic, Sorrento, Ronin)

mySet.size
> Int = 3

mySet("Ronin")
> Boolean = true
```

- **drop** removes the first ***n*** elements

```
val mySet = Set("Titanic", "Sorrento", "Ronin")

val myset2 = mySet.drop(1)
> myset2: scala.collection.immutable.Set[String] =
        Set(Sorrento, Ronin)

mySet
> mySet: scala.collection.immutable.Set[String] =
   Set(Titanic, Sorrento, Ronin)
```

## Agenda

1. Review
2. Scala Collections for Spark Programming
3. **Sets**: Creating a Collection of Unique Elements
4. **Lists and ListBuffers**: Fast Access to Head of Collection
5. **Arrays**: Fast Access to Arbitrary Elements
6. **Maps**: Fast Access with a Key
7. Common Collection Type Conversions
8. Midterm Exam Review

- A **`List`** is a finite immutable sequence
  - Very commonly used in Scala programming
  - Accessing the first element and adding an element to the front of the list are constant-time operations
- A **`List`** literal can be constructed using **`::`** (cons operator) and **`Nil`**

```
val newList = "a" :: "b" :: "c" :: Nil
> newList: List[String] = List(a, b, c)
```

- Create a list using the **`List`** keyword
  - An alternative to using the cons operator and **`Nil`**
- Elements of a **`List`** can be accessed using an index

```
val models = List("Titanic", "Sorrento", "Ronin")
> models: List[String] = List(Titanic, Sorrento, Ronin)

models(1)
> String = Sorrento
```

- **`Lists`** can contain a single data type or type **`Any`**

```
val randomlist = List("iFruit", 3, "Ronin", 5.2)

> randomlist: List[Any] = List(iFruit, 3, Ronin, 5.2)
```

- **`Lists`** can contain Collection and **`Tuple`** elements as well as simple types

```
val devices = List(("Sorrento", 10), ("Sorrento", 20),
("iFruit", 30))

> devices: List[(String, Int)] = List((Sorrento,10),
(Sorrento,20), (iFruit,30))
```

```
val myList: List[Int] = List(1, 5, 7, 1, 3, 2)
> myList: List[Int] = List(1, 5, 7, 1, 3, 2)

myList.sum
> Int = 19

myList.max
> Int = 7

myList.take(3)
> List[Int] = List(1, 5, 7)

myList.sorted
> List[Int] = List(1, 1, 2, 3, 5, 7)

myList.reverse
> List[Int] = List(2, 3, 1, 7, 5, 1)
```

```
val myListA = List("iFruit", "Sorrento", "Ronin")
val myListB = List("iFruit", "MeToo",    "Ronin")

val myListC = myListA.union(myListB)
> myListC: List[String] = List(iFruit, Sorrento, Ronin,
iFruit, MeToo, Ronin)

val myListD = myListA ++ myListB
> myListD: List[String] = List(iFruit, Sorrento, Ronin,
iFruit, MeToo, Ronin)

myListC == myListD
> Boolean = true

val myListC = myListA.intersect(myListB)
> myListC: List[String] = List(iFruit, Ronin)
```

- Operations using the lists leave the original lists unchanged

```
myListA ++ myListB

myListA
> res14: List[String] = List(iFruit, Sorrento, Ronin)

myListB
> res15: List[String] = List(iFruit, MeToo, Ronin)
```

- Use :+ to append to a list

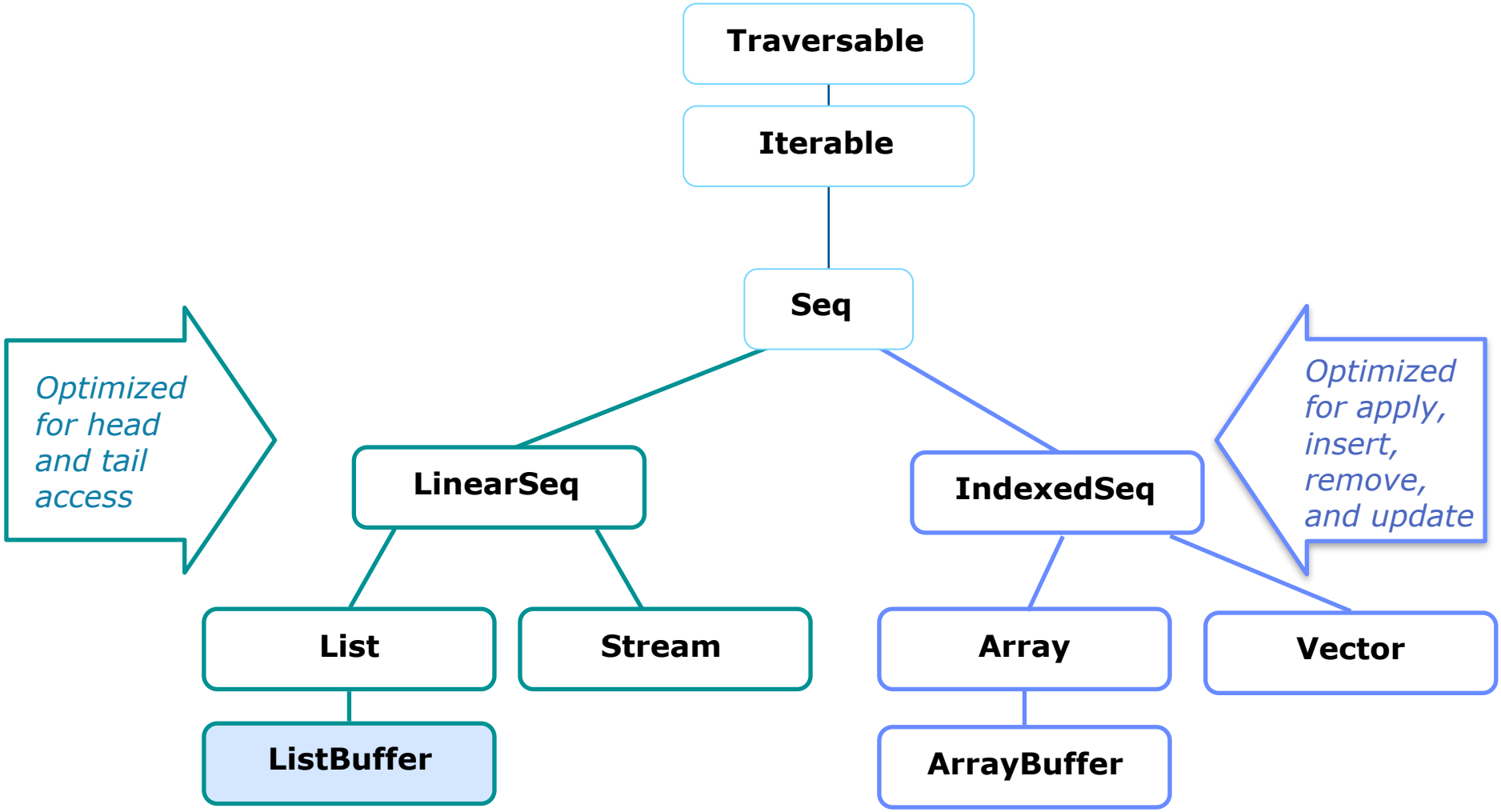
```
val myListE = myListA :+ "xPhone"
> myListE: List[String] = List(iFruit, Sorrento, Ronin,
xPhone)
```

- A **`ListBuffer`** is the mutable form of a **`List`**
- A **`ListBuffer`** provides constant time prepend and append operations
- Use **`-=`** to remove the first occurrence of a value, other values remain

```
val listBuf =
scala.collection.mutable.ListBuffer.empty[Int]

listBuf += 17
listBuf += 29
listBuf += 45
> listBuf.type = ListBuffer(17, 29, 45)

listBuf -= 17
> listBuf.type = ListBuffer(29, 45)
```

- **`ListBuffer`** is mutable with respect to its elements, however, attempts to reassign the pointer address are not allowed if it was declared with **`val`**

```
import scala.collection.mutable.ListBuffer

val listBuf2 = ListBuffer("abc")

listBuf2 += "def"
> listBuf2.type = ListBuffer(abc, def)

listBuf = listBuf2
> error: reassignment to val listBuf = listBuf2
```




*Buffers are mutable versions – supporting insert, remove, append methods*

- Use **`var`** to create a mutable and reassignable **`ListBuffer`**

```
var listBufVar = ListBuffer("one")
listBufVar += "banana"
> listBufVar.type = ListBuffer(one, banana)

listBuf2
> scala.collection.mutable.ListBuffer[String] =
ListBuffer(abc, def)

listBufVar = listBuf2

listBufVar
>scala.collection.mutable.ListBuffer[String] =
ListBuffer(abc, def)
```

- Review this example carefully
  - What is happening when **`listBuf2`** is modified?

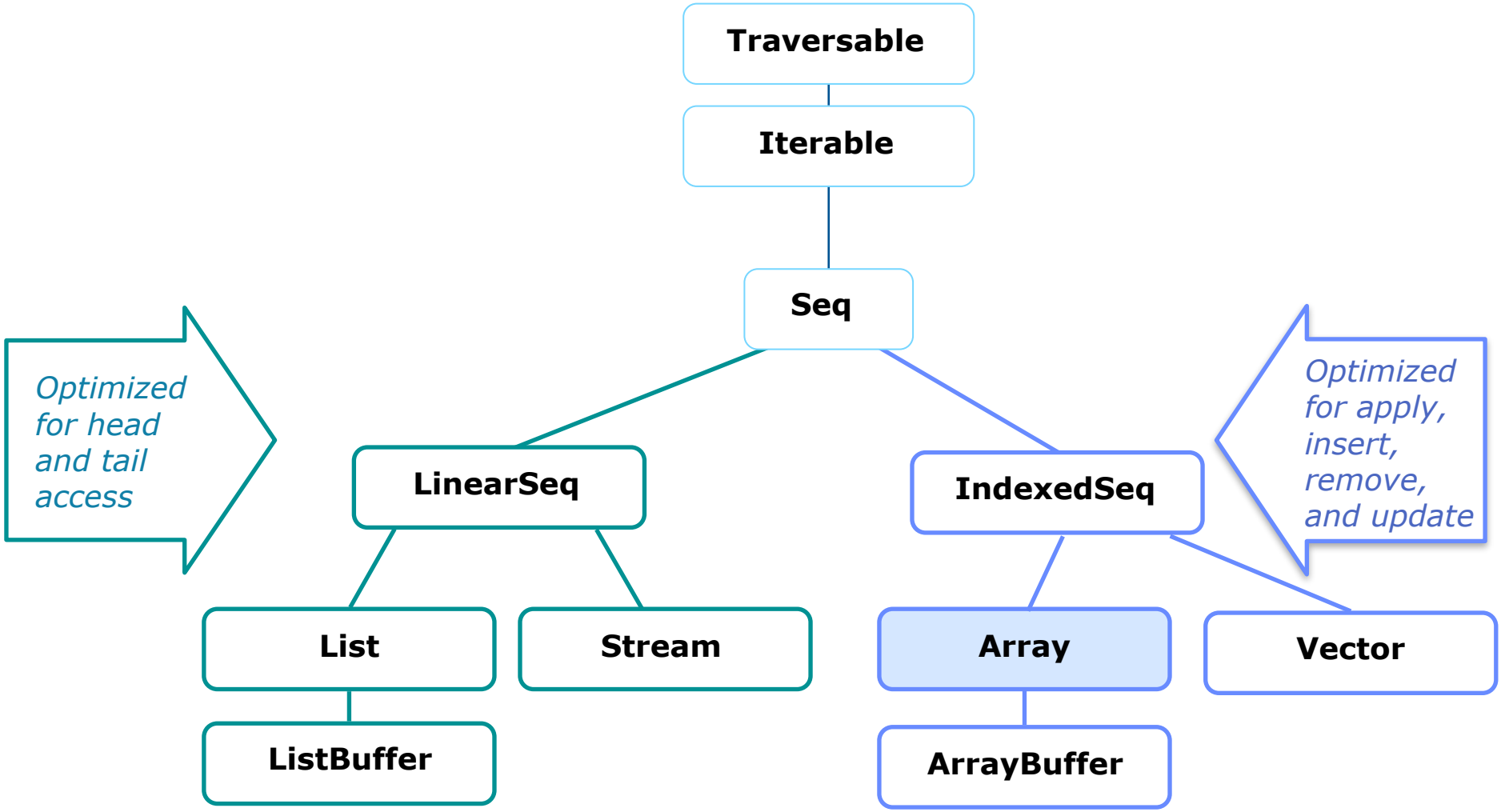
```
listBuf2 += "xyz"

listBuf2
> scala.collection.mutable.ListBuffer[String] =
ListBuffer(abc, def, xyz)

listBufVar
> scala.collection.mutable.ListBuffer[String] =
ListBuffer(abc, def, xyz)
```

## Agenda

1. Review
2. Scala Collections for Spark Programming
3. **Sets**: Creating a Collection of Unique Elements
4. **Lists and ListBuffers**: Fast Access to Head of Collection
5. **Arrays**: Fast Access to Arbitrary Elements
6. **Maps**: Fast Access with a Key
7. Common Collection Type Conversions
8. Midterm Exam Review

- An **`Array`** is mutable but not resizable
  - Created with a fixed number of elements
    - You cannot change the number of elements in the array
    - You *can* update the value of an existing element
  - Array elements can be of a single type or **`Any`**

```
val devs = Array("iFruit", "MeToo", "Ronin")
> devs: Array[String] = Array(iFruit, MeToo, Ronin)

devs(2) = "Ronin"

devs
> Array[String] = Array(iFruit, MeToo, Titanic)
```



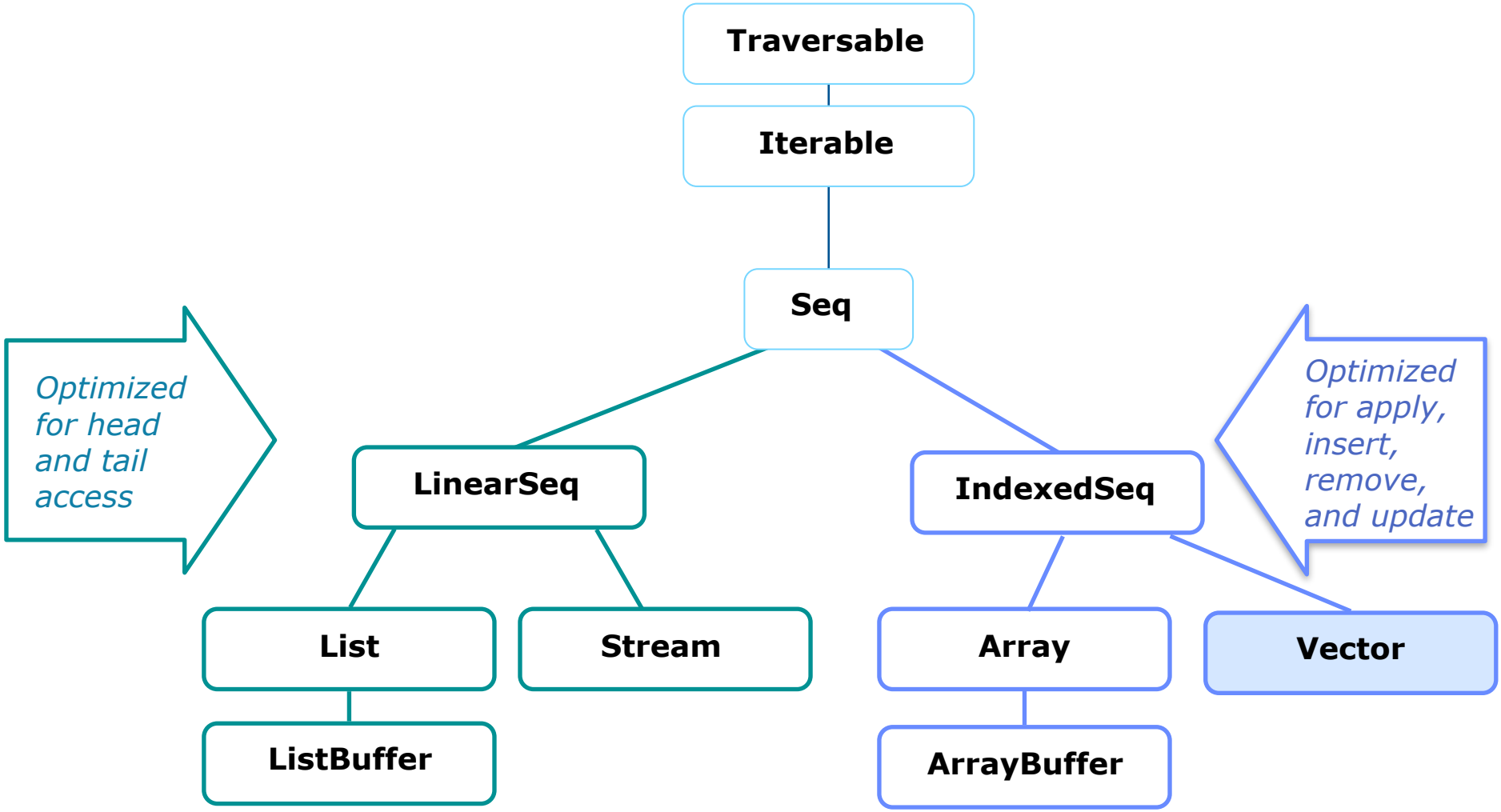

*Buffers are mutable versions – supporting insert, remove, append methods*

- Arrays are fixed in both size and type
- The assignment below shows that the value being assigned must match the type of the elements in the array

```
val devices: Array[String] = new Array[String](4)
devices.update(0, "Sorrento")
devices
> Array[String] = Array(Sorrento, null, null, null)

devices(0) = "Titanic"
devices
> Array[String] = Array(Titanic, null, null, null)

devices(1) = 256
> error: type mismatch; found: Int(256) required: String

devices.length
> Int = 4
```

- **`Vector`**, **`Array`**, and **`List`** all inherit from the **`Seq`** type
  - **`List`** belongs to to the **`LinearSeq`** branch of **`Seq`**
  - **`Vector`**, **`Array`**, and **`String`** belong to the **`IndexedSeq`** branch

- A **`Vector`** is more efficient for random access than a **`List`**
  - Allows access to any element in effectively constant time
  - Strikes a good balance between random selection and update speed




*Buffers are mutable versions – supporting insert, remove, append methods*

- **`Vector`** is immutable, modifications are not made in place

```
val vec = Vector(1, 18, 6)
> scala.collection.immutable.Vector[Int] = Vector(1, 18, 6)

vec.updated(1, 30)
> scala.collection.immutable.Vector[Int] = Vector(1, 30, 6)
```

- Unlike **`Array`**, a **`Vector`** has flexible size

```
var vec = Vector(1, 6, 21)
> scala.collection.immutable.Vector[Int] = Vector(1, 6, 21)

vec = vec :+ 5
> Vector(1, 6, 21, 5)

vec = 77 +: vec
> Vector(77, 1, 6, 21, 5)
```

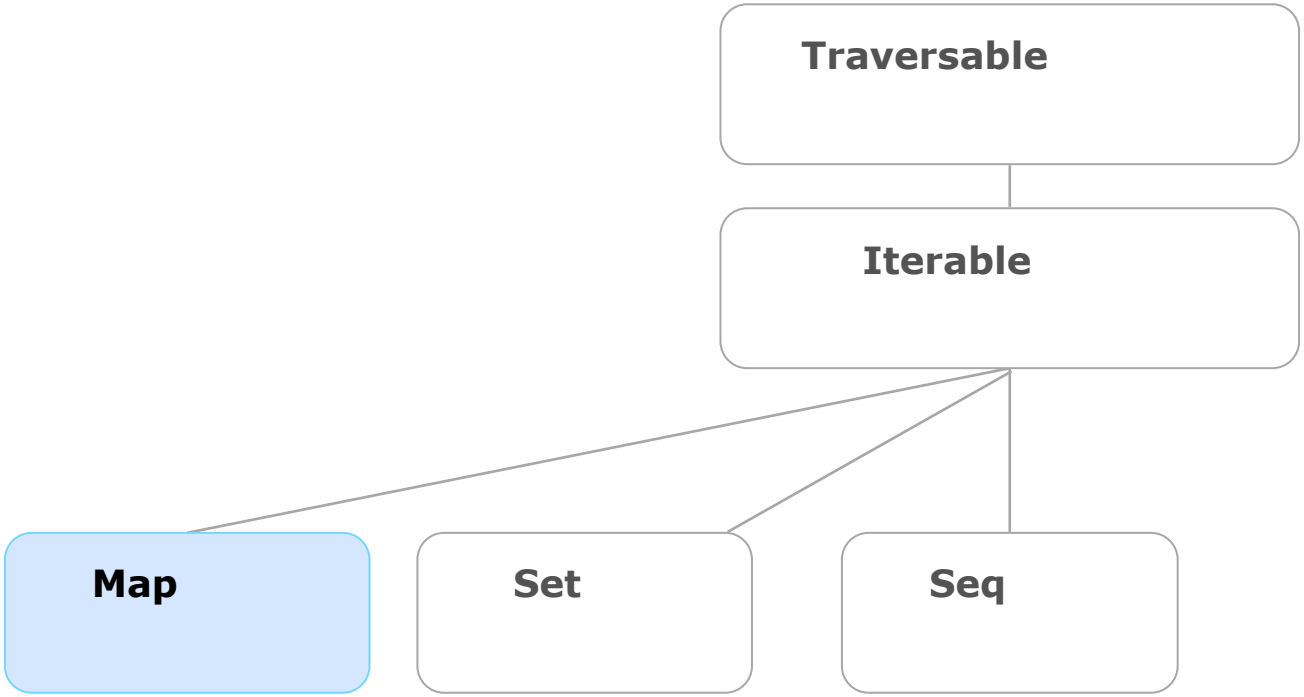
## Agenda

1. Review
2. Scala Collections for Spark Programming
3. **Sets**: Creating a Collection of Unique Elements
4. **Lists and ListBuffers**: Fast Access to Head of Collection
5. **Arrays**: Fast Access to Arbitrary Elements
6. **Maps**: Fast Access with a Key
7. Common Collection Type Conversions
8. Midterm Exam Review

- A **`Map`** is a collection of key-value pairs
  - Immutable by default – values are not modified in place

- Declare a map variable using either of these techniques
  - **`Map((key1, value1), (key2, value2))`**
  - **`Map(key1 -> value1, key2 -> value2)`**

- Keys and values
  - Keys are unique and may only appear once; values are not unique



- Commonly used for in-memory tables requiring fast access

- Used to associate names with values
  - Single record buffer of data
  - Parameters required for calling an API

```
val phoneStatus = Map(
("DTS"       -> "2014-03-15:10:10:31"),
("Brand"     -> "Titanic"),
("Model"     -> "4000"),
("UID"       -> "1882b564-c7e0-4315-aa24-228c0155ee1b"),
("DevTemp"   -> 58),
("AmbTemp"   -> 36),
("Battery"   -> 39),
("Signal"    -> 31),
("CPU"       -> 15),
("Memory"    -> 0),
("GPS"       -> true ),
("Bluetooth" -> "enabled"),
("WiFi"      -> "enabled"),
("Latitude"  -> 40.69206648),
("Longitude" -> -119.4216429))
```

- The values are associated with keys that are easily understood string names.
- For example, to determine if the WiFi is turned on, access `phoneStatus("WiFi")`

```
phoneStatus.contains("DTS")
> Boolean = true

phoneStatus.keys
> Iterable[String] = Set(AmbTemp, GPS, Memory,
  Battery, Latitude, Signal, Longitude, DevTemp,
  Model, WiFi, UID, CPU, DTS, Brand, Bluetooth)

phoneStatus.values
> Iterable[Any] = MapLike(36, true, 0, 39,
  40.69206648, 31, -119.4216429, 58, 4000, enabled,
  1882b564-c7e0-4315-aa24-228c0155ee1b, 15,
  2014-03-15:10:10:31, Titanic, enabled)
```

- Use **get** or **getOrElse** to avoid an exception for non-existent keys

```
phoneStatus("DTS")
> Any = 2014-03-15:10:10:31

phoneStatus("key_does_not_exist")
> java.util.NoSuchElementException: key not found:
key_does_not_exist ...

phoneStatus.get("key_does_not_exist")
> Option[Any] = None

phoneStatus.get("DTS")
> Option[Any] = Some(2014-03-15:10:10:31)

phoneStatus.getOrElse("key_does_not_exist", "No Key")
> Any = No Key
```

- We cannot change value for **`Wifi`** to **`disabled`**

```
phoneStatus("Wifi") = "disabled"
> error: value update is not a member of
  scala.collection.immutable.Map[String,String]
```

- Changing a value requires explicitly creating a mutable **`Map`**

```
val mutRec = scala.collection.mutable.Map(("Brand" ->
"Titanic"), ("Model" -> "4000"), ("Wifi" -> "enabled"))
> scala.collection.mutable.Map[String,String] =
  Map(Wifi-> enabled, Model -> 4000, Brand -> Titanic)

mutRec("Wifi") = "disabled"
mutRec
> scala.collection.mutable.Map[String,String] =
 Map(Wifi-> disabled, Model -> 4000, Brand -> Titanic)
```

## Agenda

1. Review
2. Scala Collections for Spark Programming
3. **Sets**: Creating a Collection of Unique Elements
4. **Lists and ListBuffers**: Fast Access to Head of Collection
5. **Arrays**: Fast Access to Arbitrary Elements
6. **Maps**: Fast Access with a Key
7. Common Collection Type Conversions
8. Midterm Exam Review

- Scala provides several methods for converting between collection types in Scala Spark programs

```
val myList = List("Titanic", "F01L", "enabled", 32)

val myArray = myList.toArray
> myArray: Array[Any] = Array(Titanic, F01L, enabled, 32)

val myIterable = myList.toIterable
> myIterable: Iterable[Any] = List(Titanic, F01L, enabled,
  32)

val myList2 = myIterable.toList
> myList2: List[Any] = List(Titanic, F01L, enabled, 32)

val myList3 = myArray.toList
> myList3: List[Any] = List(Titanic, F01L, enabled, 32)
```

- Example converting from Tuple6 to List

```
val myTup = (4, "MeToo", "1.0", 37.5, 41.3, "Enabled")
> myTup: (Int, String, String, Double, Double, String) =
  (4,MeToo,1.0,37.5,41.3,Enabled)

myTup.getClass
> Class[_ <: (Int, String, String, Double, Double, String)]
  = class scala.Tuple6

val myList = myTup.productIterator.toList
> myList: List[Any] = List(4, MeToo, 1.0, 37.5, 41.3,
  Enabled)
```

- Strings in Scala are treated as collections similar to **Arrays**
- Strings can be converted to other **Collection** types

```
val myStr = "A Banana"
myStr(2)
> Char = B

myStr.toArray
> Array[Char] = Array(A,  , B, a, n, a, n, a)

myStr.toList
> List[Char] = List(A,  , B, a, n, a, n, a)

myStr.toSet
> scala.collection.immutable.Set[Char] = Set(n, A, a,  , B)
```

- **`Tuple`**
  - Fixed size: **`Tuple2`**, **`Tuple3`**, …, **`Tuple22`**
  - Not part of the collection library
  - Created at compile time, which restricts their flexibility
- **`List`**
  - Flexible size
  - Elements are immutable, so they cannot be changed by assignment
  - Fast addition and removal at head
  - Slow access to arbitrary indexes
- **`ListBuffer`**
  - Flexible size
  - Elements are mutable
  - Constant time append and prepend operations

- **`Array`**
  - Created with a fixed number of elements and not resizable
  - Fast access to arbitrary indexes
- **`Map`**
  - For working with key-value pairs
    To create a mutable **`Map`**, import **`scala.collection.mutable`** explicitly and declare the **`Map`** as **`mutable.Map`**

## Agenda

1. Review
2. Scala Collections for Spark Programming
3. **Sets**: Creating a Collection of Unique Elements
4. **Lists and ListBuffers**: Fast Access to Head of Collection
5. **Arrays**: Fast Access to Arbitrary Elements
6. **Maps**: Fast Access with a Key
7. Common Collection Type Conversions
8. Midterm Exam Review

## Homework

See the homework packet for details.