Scala Collections

Georgia Tech

Scala Collections

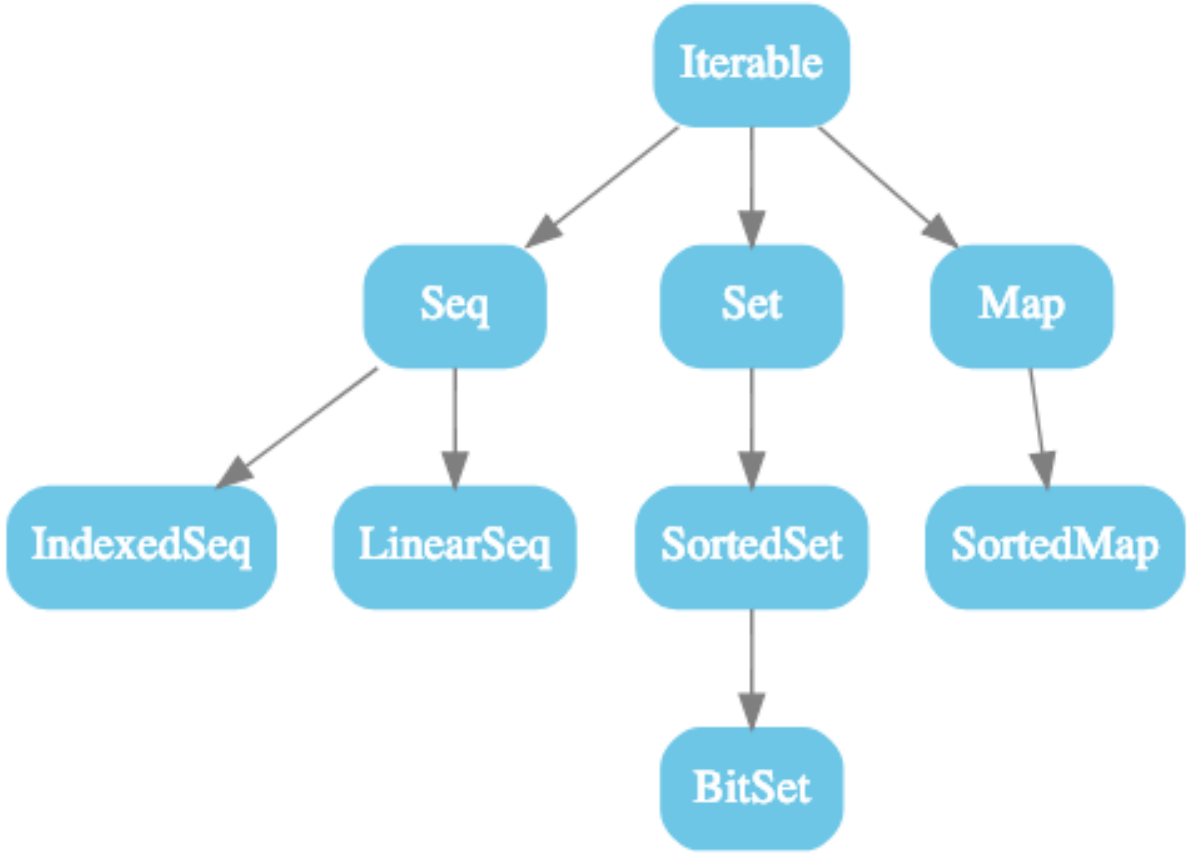

Figure 1: Abstract classes and traits in `scala.collection`

From https://docs.scala-lang.org/overviews/collections-2.13/overview.html

Georgia Tech

Immutable Collections

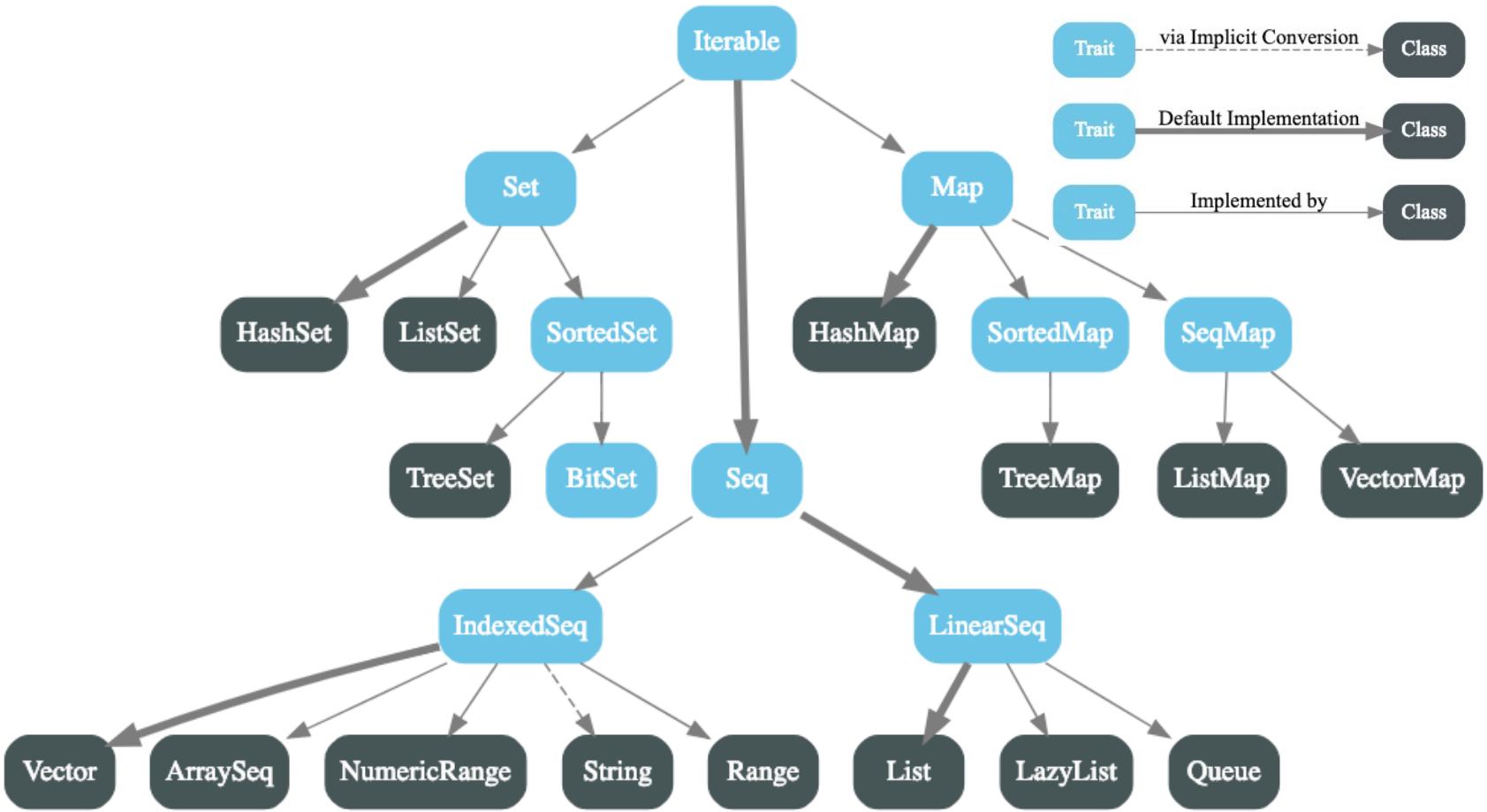

Figure 2: Immutable collections in `scala.collection.immutable`

Mutable Collections

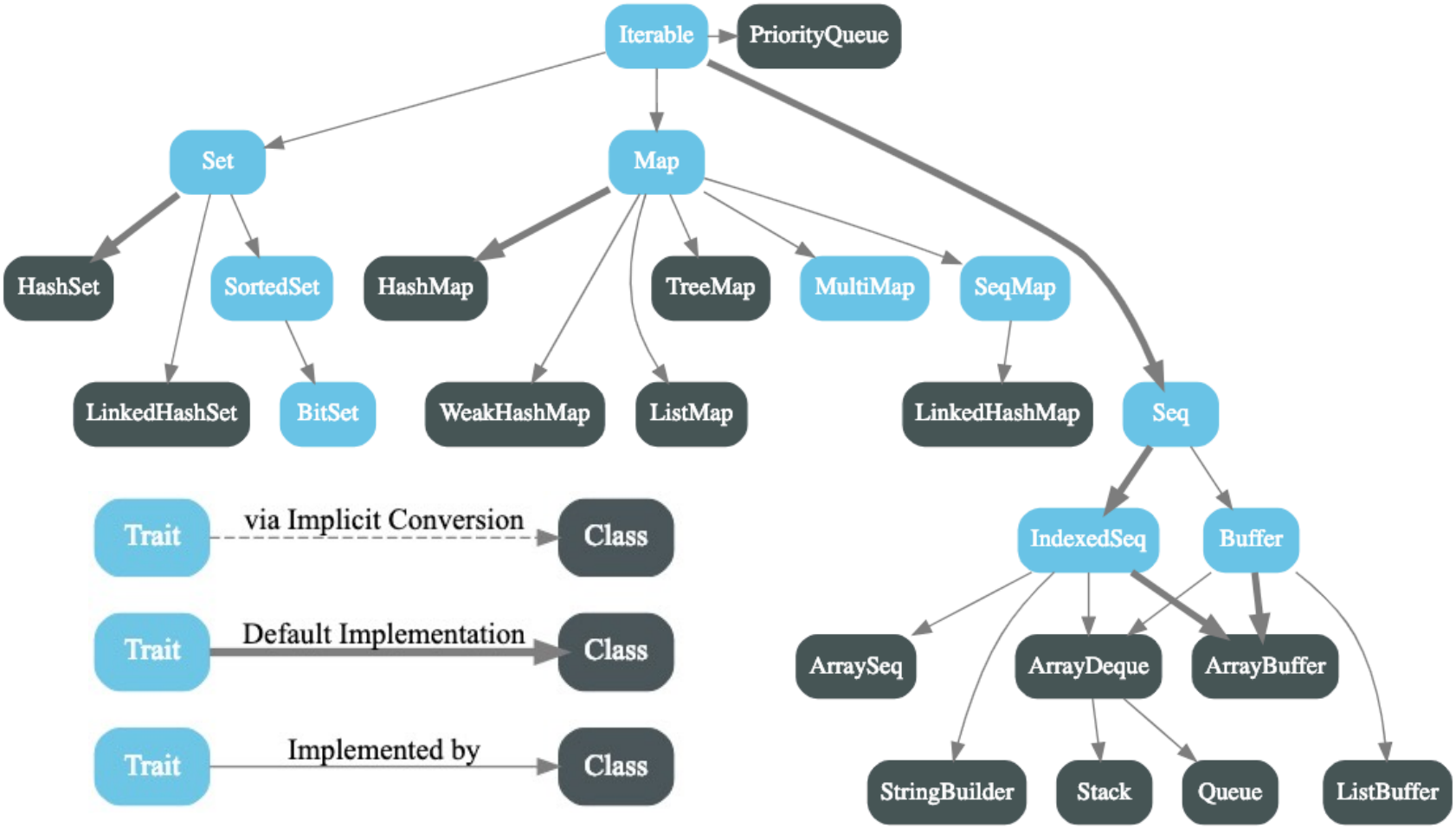

Figure 3: Mutable collections in `scala.collections.mutable`

Type Aliases in `scala` and `scala.Predef`

`scala.Predef` provides type aliases for commonly used collections, such as `List` (and constructors `::` and `Nil`), `Set`, and `Map`.

`scala` provides type aliases for `IndexedSeq`, `Seq`, `Iterable` (and `Iterator`) and `Vector`.

All of these alias to collections in `scala.collection.immutable`, so these companion object factories create immutable collections:

```
1  Iterable("x", "y", "z")
2  List(1, 2, 3)
3  Vector(1, 2, 3)
4  Seq(1.0, 2.0)
5  IndexedSeq(1.0, 2.0)
6  Set(1, 2, 3)
7  Map("x" -> 24, "y" -> 25, "z" -> 26)
```

Georgia Tech

Collection-like Classes in `scala`

The `scala` package contains collection-like classes which are not part of the collections framework: `Array` and the TupleN classes.

Arrays are mutable fixed-sized Sequences of like-typed elements which map one-to-one with Java arrays, except Scala arrays are generic.

```
1 val zs: Array[Int] = Array(1, 2, 3)
2 zs(0) = 42
3 zs == Array(42, 2, 3)
```

Georgia Tech

Tuples

A tuple is an immutable fixed-size collection of values of mixed types. The `Tuple` companion object factory method creates instances of classes named `Tuple1` through `Tuple22`, so the largest tuple you can create is 22 elements in size.

Tuples are about convenience:

```
1 val dog = ("Chloe", 6)
2 dog._1 == "Chloe"
3 dog._2 == 6
4 val (name, age) = dog
5 name == "Chloe"
6 age == 6
```

Be careful: if you leave off the parenthesis you don't get destructuring bind:

```
1 val name, age = dog
2 name == ("Chloe", 6)
3 age == ("Chloe", 6)
```

Georgia Tech

Idiomatic Functional Lists

List construction:

```
1  val nums: List[Int] = List(1, 2, 3, 4)
2
3  val leer = Nil  // Nil is an empty list constant
4  val vide = List()
```

Lists are homogeneous (elements have same type) and generic.

- `List` is not a type – no raw collections in Scala
- `List[T]` is a generic type, or type constructor
- `List[Int]` is a type because an argument for `T` is provided

Due to type inference, these are equivalent:

```
1  val nums = List(1, 2, 3, 4)
2  val nums: List[Int] = List[Int](1, 2, 3, 4)
```

Georgia Tech

Basic List Operations

Given `xs == List(1,2,3)`,

- `xs.head` returns the first element of a list: `1`
- `xs.tail` returns a list consisting of all elements except the first: `List(2,3)`
- `xs.isEmpty` returns true if the list is empty: `false`

Using these basic operations and functional list idioms, we can implement insertion sort as:

```
1  def insertionSort(xs: List[Int]): List[Int] =
2    if (xs.isEmpty) Nil
3    else insert(xs.head, insertionSort(xs.tail))
4
5  def insert(x: Int, xs: List[Int]): List[Int] =
6    if (xs.isEmpty || x <= xs.head) x :: xs
7    else xs.head :: insert(x, xs.tail)
```

Georgia Tech

List Patterns

The `List` constructor can be used for a destructuring bind:

```
scala> val List(a, b, c) = List("apples", "bananas", "kiwis")
a: String = apples
b: String = bananas
c: String = kiwis
```

Recall that you can "cons" an element to the head of a list, so the list above could be constructed like:

```
scala> val fruits = "apples"::"bananas"::"kiwis"::Nil
fruits: List[String] = List(apples, bananas, kiwis)
```

Georgia Tech

consing Lists

Recall that `::` is a method that associates to the right, that is, it's invoked on its right operand.

```
scala> val head = "apple"
head: String = apple

scala> val tail = List("bananas", "kiwis")
tail: List[String] = List(bananas, kiwis)

scala> head::tail
res0: List[String] = List(apple, bananas, kiwis)

scala> tail.::(head)
res1: List[String] = List(apple, bananas, kiwis)
```

Georgia Tech

Pattern Matching on List Structure

Scala allows infix operators like `::` to be used in pattern matches. So we could rewrite insertion sort as:

```
1  def insertionSort(xs: List[Int]): List[Int] = xs match {
2    case List()  => List()
3    case h :: t => insert(h, insertionSort(t))
4  }
5  def insert(x: Int, xs: List[Int]): List[Int] = xs match {
6    case List() => List(x)
7    case h :: t => if (x <= h) x :: xs else h :: insert(x, t)
8  }
```

When you read list pattern matching code in functional languages read `h` as "head" and `t` as "tail".

Now let's look at the most general collections operations: those defined on `Iterable`

Size-related Methods on `Iterable`

- `xs.isEmpty` Tests whether the collection is empty.
- `xs.nonEmpty` Tests whether the collection contains elements.
- `xs.size` The number of elements in the collection.
- `xs.knownSize` The number of elements, if this one takes constant time to compute, otherwise -1.
- `xs.sizeCompare(ys)` Returns a negative value if xs is shorter than the ys collection, a positive value if it is longer, and 0 if they have the same size. Works even if the collection is infinite, for example LazyList.from(1) sizeCompare List(1, 2) returns a positive value.
- `xs.sizeCompare(n)` Returns a negative value if xs is shorter than n, a positive value if it is longer, and 0 if it is of size n. Works even if the collection is infinite, for example LazyList.from(1) sizeCompare 42 returns a positive value.
- `xs.sizeIs < 42`, `xs.sizeIs != 42`, etc. Provides a more convenient syntax for xs.sizeCompare(42) < 0, xs.sizeCompare(42) $!= 0$, etc., respectively.

Element Retrieval Methods on `Iterable`

- `xs.head` The first element of the collection (or, some element, if no order is defined).
- `xs.headOption` The first element of xs in an option value, or None if xs is empty.
- `xs.last` The last element of the collection (or, some element, if no order is defined).
- `xs.lastOption` The last element of xs in an option value, or None if xs is empty.

Georgia Tech

Subcollection Methods on `Iterable`

- `xs.tail` The rest of the collection except xs.head.
- `xs.init` The rest of the collection except xs.last.
- `xs.slice(from, to)` A collection consisting of elements in some index range of xs (from from up to, and excluding to).
- `xs take n` A collection consisting of the first n elements of xs (or, some arbitrary n elements, if no order is defined).
- `xs drop n` The rest of the collection except xs take n.
- `xs takeWhile p` The longest prefix of elements in the collection that all satisfy p.
- `xs dropWhile p` The collection without the longest prefix of elements that all satisfy p.
- `xs takeRight n` A collection consisting of the last n elements of xs (or, some arbitrary n elements, if no order is defined).
- `xs dropRight n` The rest of the collection except xs takeRight n.
- `xs filter p` The collection consisting of those elements of xs that satisfy the predicate p.
- `xs withFilter p` A non-strict filter of this collection. Subsequent calls to map, flatMap, foreach, and withFilter will only apply to

Georgia Tech

Mapping Methods on `Iterable`

- `xs map f` The collection obtained from applying the function f to every element in xs.
- `xs flatMap f` The collection obtained from applying the collection-valued function f to every element in xs and concatenating the results.
- `xs collect f` The collection obtained from applying the partial function f to every element in xs for which it is defined and collecting the results.

Georgia Tech

Folding Methods on `Iterable`

- `xs.foldLeft(z)(op)` Apply binary operation op between successive elements of xs, going left to right and starting with z.
- `xs.foldRight(z)(op)` Apply binary operation op between successive elements of xs, going right to left and ending with z.
- `xs reduceLeft op` Apply binary operation op between successive elements of non-empty collection xs, going left to right.
- `xs reduceRight op` Apply binary operation op between successive elements of non-empty collection xs, going right to left.

Convenience Folds

- `xs.sum` The sum of the numeric element values of collection xs.
- `xs.product` The product of the numeric element values of collection xs.
- `xs.min` The minimum of the ordered element values of collection xs.
- `xs.max` The maximum of the ordered element values of collection xs.
- `xs.minOption` Like min but returns None if xs is empty.

Georgia Tech

Zipping Methods on `Iterable`

- `xs zip ys` A collection of pairs of corresponding elements from xs and ys.
- `xs.zipAll(ys, x, y)` A collection of pairs of corresponding elements from xs and ys, where the shorter sequence is extended to match the longer one by appending elements x or y.
- `xs.zipWithIndex` An collection of pairs of elements from xs with their indices.

Georgia Tech

Conversion Methods on `Iterable`

- `xs.toArray` Converts the collection to an array.
- `xs.toList` Converts the collection to a list.
- `xs.toIterable` Converts the collection to an iterable.
- `xs.toSeq` Converts the collection to a sequence.
- `xs.toIndexedSeq` Converts the collection to an indexed sequence.
- `xs.toSet` Converts the collection to a set.
- `xs.toMap` Converts the collection of key/value pairs to a map. If the collection does not have pairs as elements, calling this operation results in a static type error.
- `xs.to(SortedSet)` Generic conversion operation that takes a collection factory as parameter.

Georgia Tech

Sets and Maps

Sets are immutable by default, so we "add" to them with reassignment

```
1 var trooperSet = Set("Thorny", "Farva", "Mac", "Mac")
2 trooperSet == Set("Thorny", "Farva", "Mac")
3 trooperSet += "Rabbit"
4 trooperSet.contains("Rabbit")
```

Map elements created with 2-tuples, which are usually created with ->

```
1 var majors = Map(
2   ("CS", "Computer Science"),
3   "CM" -> "Computational Media",
4   "EE" -> "Electrical Engineering"
5 )
6 majors += "IE" -> "Industrial Engineering"
7 majors("IE")
8 majors.getOrElse("AA", "Unknown Major")
```

-> uses implicit conversion to create `Tuple2` instances.

Georgia Tech