Implementing the Scala 2.13 collections

Stefan Zeiger, Lightbend

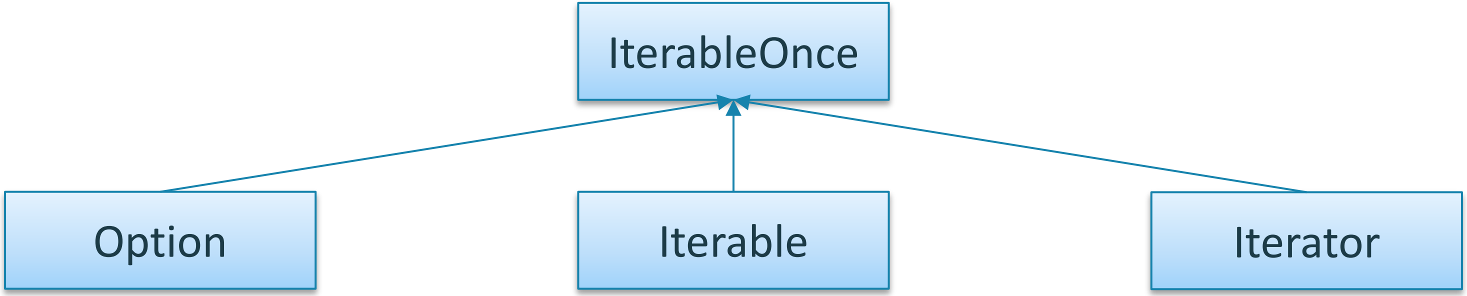
Scala 2.13

- Redesigned collection library
- Easier for users
  - Better discoverability
  - Better error messages
- Easier for collection implementers
  - More regular
  - But still plenty of necessary complexity

IterableOnce – Not Quite a Collection

IterableOnce

- "Has an `iterator` method"
  - Similar to `java.lang.Iterable`
  - But supports single-traversal implementations
- Lightweight interface
  - No unnecessary methods
  - Can be implemented by non-collection types
- Many collection methods accept `IterableOnce` values



IterableOnce

```
class MyIterableOnce extends IterableOnce[Int] {
  private[this] val data = Array(1, 2, 3)

  def iterator = data.iterator
}
```

IterableOnce

```
class MyIterableOnce extends IterableOnce[Int] {
  private[this] val data = Array(1, 2, 3)

  def iterator = data.iterator
  override def knownSize = data.length
}


val b = mutable.ArrayBuffer.empty[Int]
b.addAll(new MyIterableOnce)
```

IterableOnce

```
class MyIterableOnce extends IterableOnce[Int] {
  private[this] val data = Array(1, 2, 3)

  def iterator = data.iterator
  override def knownSize = data.length
  override def stepper[S <: Stepper[_]]
    (implicit shape: StepperShape[Int, S]): S = data.stepper[S]
}
```

Previously in *scala-java8-compat*
Like Java's `Spliterator`

- Parallel collections implement parallelizable Steppers

IterableOnce

```
class MyIterableOnce extends IterableOnce[Int] {
  private[this] val data = Array(1, 2, 3)

  def iterator = data.iterator
  override def knownSize = data.length
  override def stepper[S <: Stepper[_]]
    (implicit shape: StepperShape[Int, S]): S = data.stepper[S]
}


val st: IntStepper = (new MyIterableOnce).stepper
val it: PrimitiveIterator.OfInt = st.javaIterator
while(it.hasNext)
  println(it.next())
```

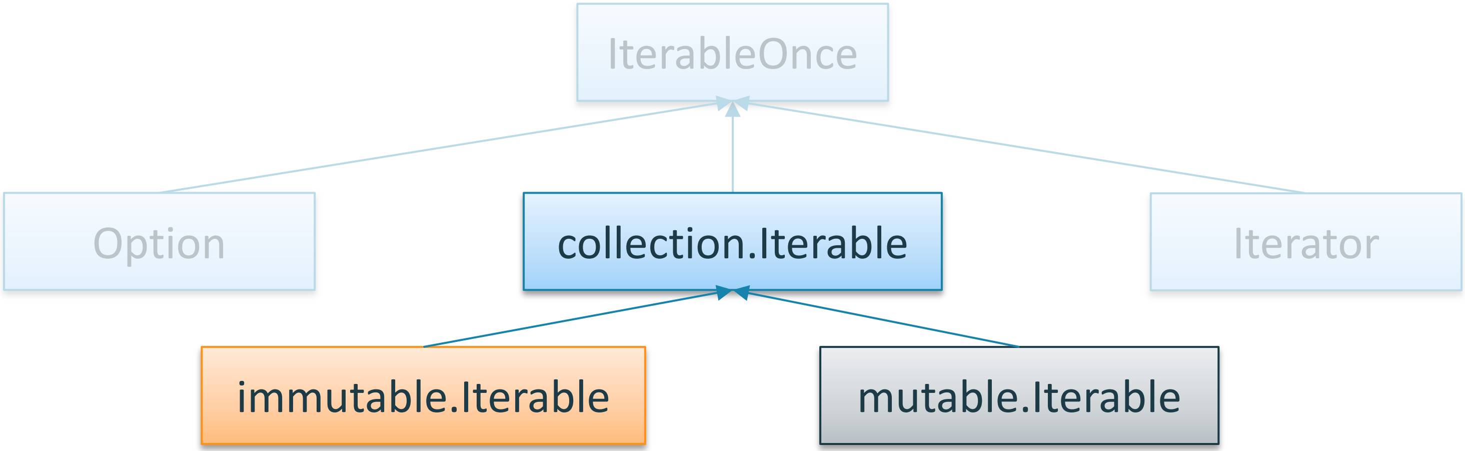
Implementing a Real Collection

Simplified immutable.ArraySeq

- Sequence of elements with arbitrary type
- Immutable
- Array-like performance characteristics

```
class MySeq[+A](data: Array[Any]) extends ??? {
  ...
}
```

Abstract Collection Types



IterableOnce

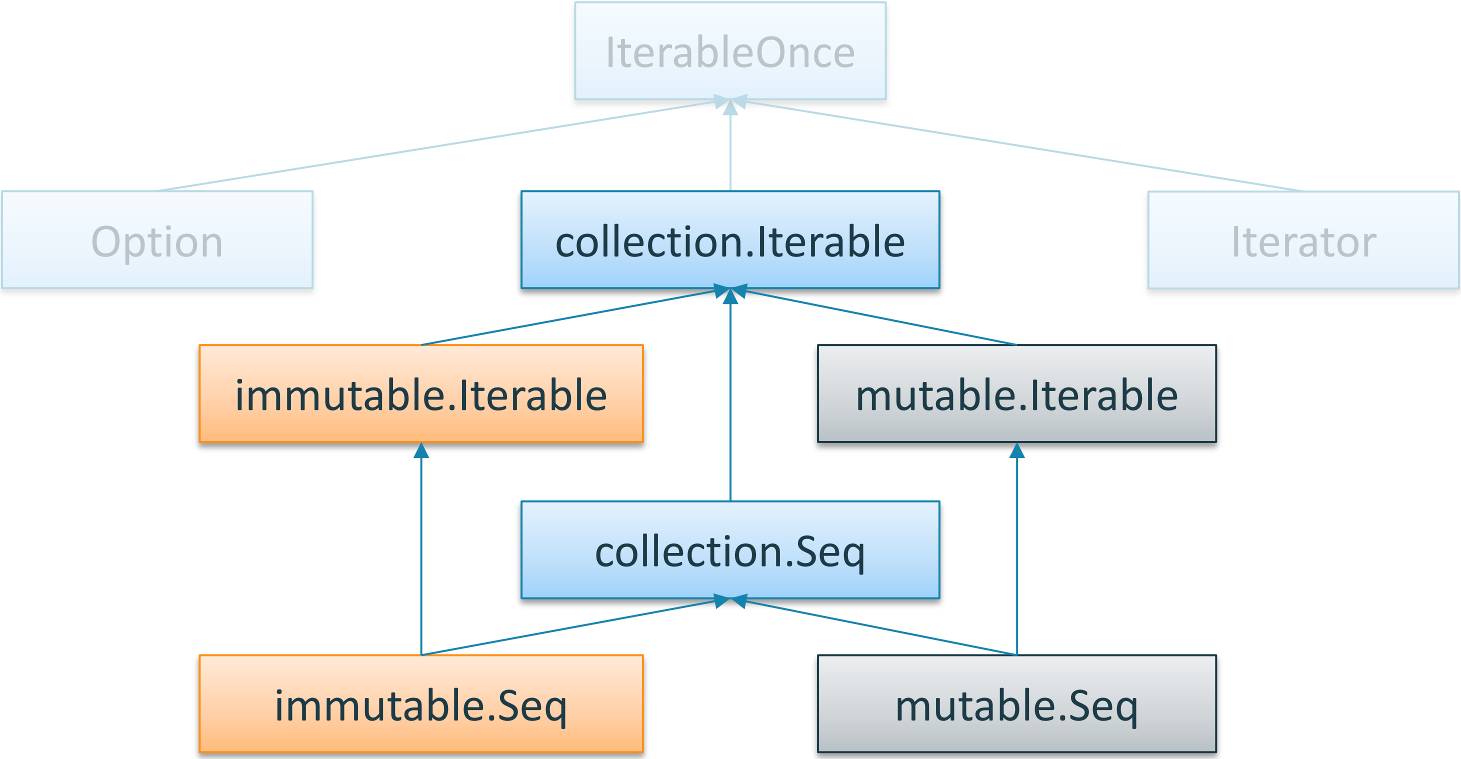
Option

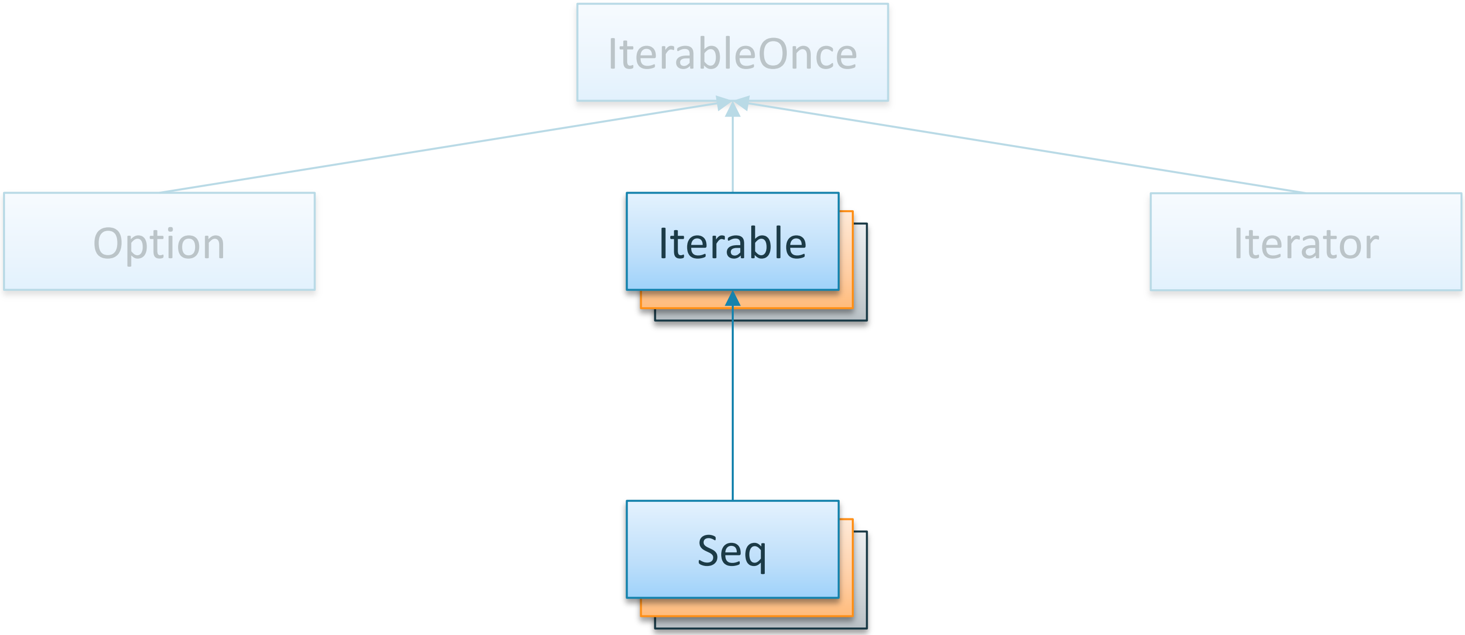
collection.Iterable

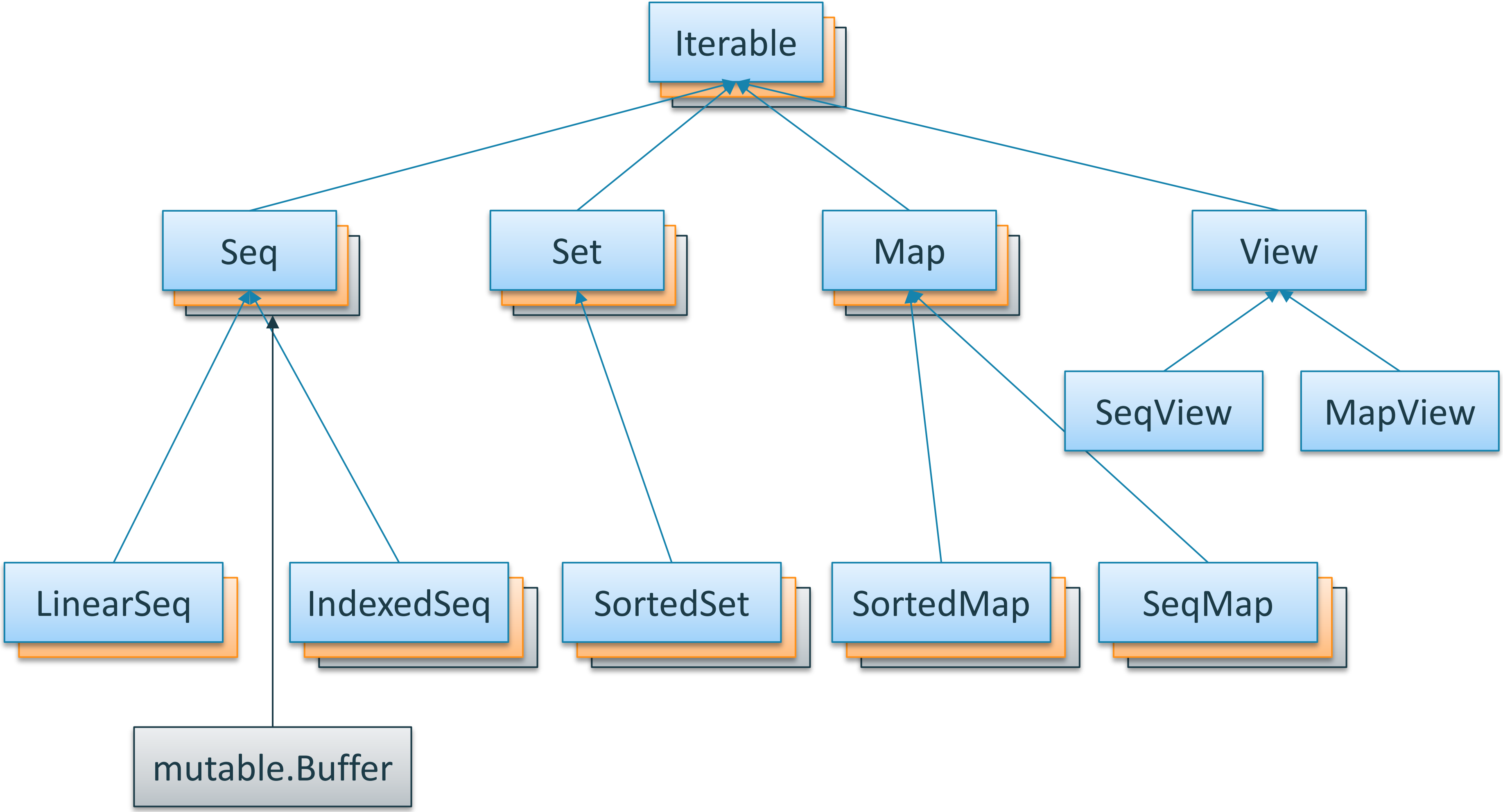
Iterator

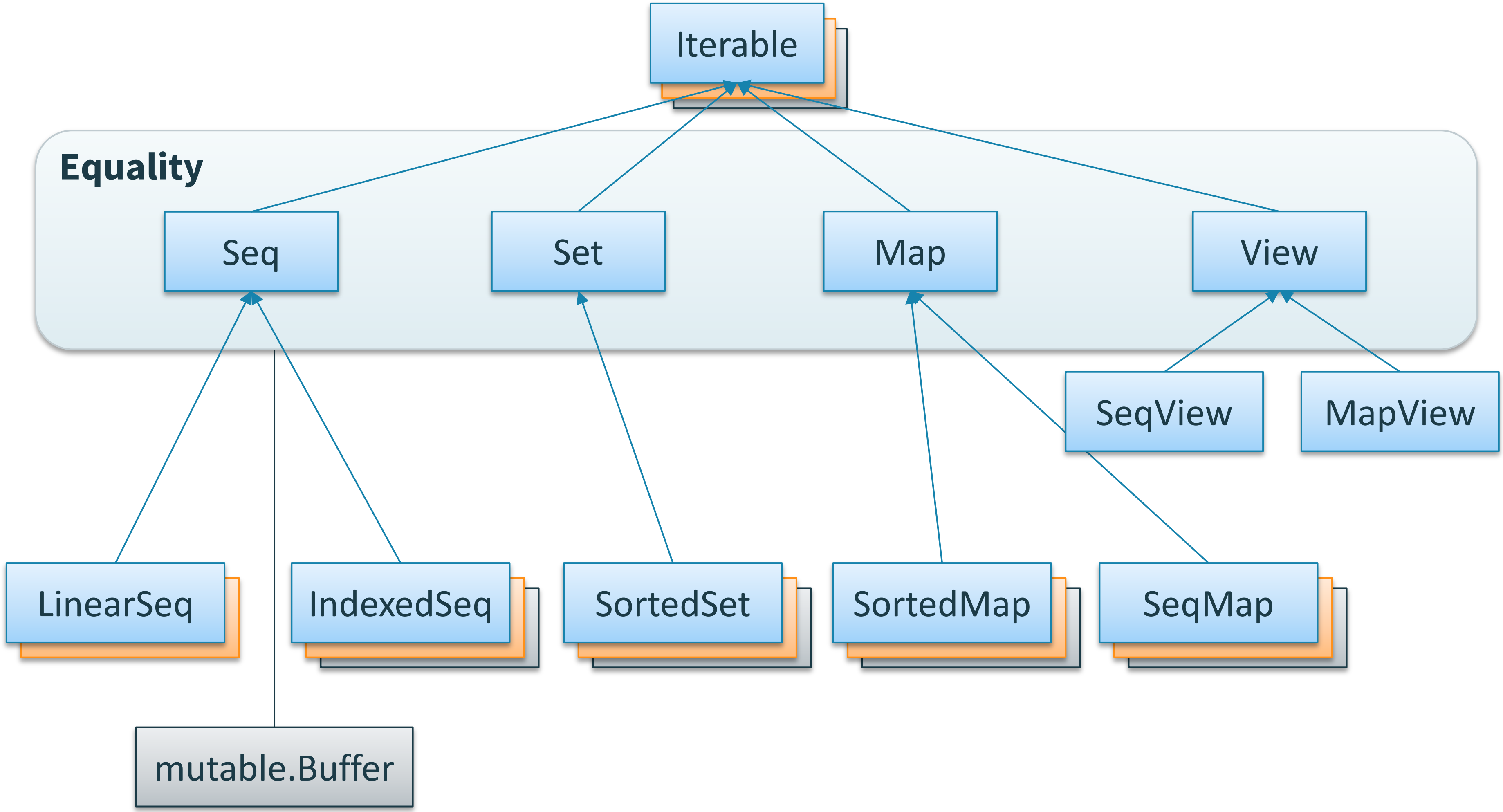
immutable.Iterable

mutable.Iterable


IterableOnce
Option
collection.Iterable
Iterator
immutable.Iterable
mutable.Iterable
collection.Seq
immutable.Seq
mutable.Seq


IterableOnce
Option
Iterable
Iterator
Seq

Iterable
Seq
Set
Map
View
SeqView
MapView
LinearSeq
IndexedSeq
SortedSet
SortedMap
SeqMap
mutable.Buffer


Iterable
Equality
Seq
Set
Map
View
SeqView
MapView
LinearSeq
IndexedSeq
SortedSet
SortedMap
SeqMap
mutable.Buffer

Implementing an immutable IndexedSeq

Simple IndexedSeq

```scala
class MySeq[+A](data: Array[Any]) extends IndexedSeq[A] {
  def length: Int = data.length
  def apply(i: Int): A = data(i).asInstanceOf[A]
  override def className = "MySeq"
}

scala> val s = new MySeq[String](Array("a", "b", "c"))
s: com.example.MySeq[String] = MySeq(a, b, c)

scala> s.foreach(println)
a
b
c

scala> println(s.indexWhere(_ > "b"))
2
```

Companion Object as Factory

```scala
object MySeq extends SeqFactory[MySeq] {

  private[this] val _empty = new MySeq(Array.empty)
  def empty[A]: MySeq[A] = _empty

  def newBuilder[A]: mutable.Builder[A, MySeq[A]] =
    Array.newBuilder[Any].mapResult(new MySeq(_))

  def from[A](source: IterableOnce[A]): MySeq[A] =
    new MySeq(Array.from(source))
}
```

Using the Factory

```
scala> MySeq.tabulate(5)(_ * 10)
res0: com.example.MySeq[Int] = MySeq(0, 10, 20, 30, 40)

scala> List("a", "b", "c").to(MySeq)
res1: com.example.MySeq[String] = MySeq(a, b, c)
```

Using the Factory

```
scala> MySeq.tabulate(5)(_ * 10)
res0: com.example.MySeq[Int] = MySeq(0, 10, 20, 30, 40)

scala> List("a", "b", "c").to(MySeq)
res1: com.example.MySeq[String] = MySeq(a, b, c)


scala> res0.filter(_ > 25)
res2: IndexedSeq[Int] = Vector(30, 40)

scala> res0.map(_.toString)
res3: IndexedSeq[String] = Vector(0, 10, 20, 30, 40)
```

Ops Traits

```
class MySeq[+A](data: Array[Any])
  extends IndexedSeq[A] {

  ...
}
```

extends IndexedSeqOps[A, IndexedSeq, IndexedSeq[A]]

```
  def map[B](f: A => B): CC[B] = ...
  def filter(pred: A => Boolean): C = ...
```

Ops Traits

```
class MySeq[+A](data: Array[Any])
  extends IndexedSeq[A] {

  ...
}
```

trait IndexedSeqOps[+A, +CC[_], +C]

extends IndexedSeqOps[A, IndexedSeq, IndexedSeq[A]]

```
  def map[B](f: A => B): CC[B] = ...
  def filter(pred: A => Boolean): C = ...
```

Simple IndexedSeq

```
class MySeq[+A](data: Array[Any])
  extends IndexedSeq[A]
  with IndexedSeqOps[A, MySeq, MySeq[A]]
  with IterableFactoryDefaults[A, MySeq] {

  override def iterableFactory: SeqFactory[MySeq] = MySeq
  ...
}
```

IndexedSeq Optimizations

```
class MySeq[+A] private (data: Array[Any])
  extends AbstractSeq[A]
  with IndexedSeq[A]
  with IndexedSeqOps[A, MySeq, MySeq[A]]
  with StrictOptimizedSeqOps[A, MySeq, MySeq[A]]
  with IterableFactoryDefaults[A, MySeq]
  with DefaultSerializable {

  override def iterableFactory: SeqFactory[MySeq] = MySeq
  def length: Int = data.length
  def apply(i: Int): A = data(i).asInstanceOf[A]
  override def className = "MySeq"
}
```

Overriding Methods

```
class MySeq[+A] private (data: Array[Any]) ... {
  ...

  override def map[B](f: A => B): MySeq[B] = {
    val len = length
    val a = new Array[Any](len)
    var i = 0
    while(i < len) {
      a(i) = f(data(i).asInstanceOf[A])
      i += 1
    }
    new MySeq[B](a)
  }
}
```

Collection Kinds

Kinds in Type Theory

- Kinds classify types (like types classify values)
- *Proper types* are of kind `*` ("type")
  - All *values* have a *proper type*
  - For example: `Int`, `String`, `List[Int]`
- Unary type constructors are of kind `* → *`
  - For example: `List`, `Set`
- Binary type constructors are of kind `* → * → *`
  - For example: `Map`, `Function1`

Extending Kinds for Scala[*]

- Proper types have lower and upper bounds: `*(T, U)`
  - Abbreviate `*(Nothing, T)` to `*(T)`
  - Abbreviate `*(Any)` to `*`
- Variance tracked by kind arrows

```
trait Map[K, +V]
```

- The kind of immutable.`Map` is $* \to * \overset{+}{\to} *$
- The kind of mutable.`AnyRefMap` is $*(\texttt{AnyRef}) \to * \to *$

```
trait AnyRefMap[K <: AnyRef, V]
```

[*] http://adriaanm.github.io/files/higher.pdf

Collection Kinds

- Need to track context bounds
  - Similar to type bounds
- But not *kinds* in the type theory sense anymore
  - A type with a missing context bound implicit cannot be instantiated
  - But it is still a valid type

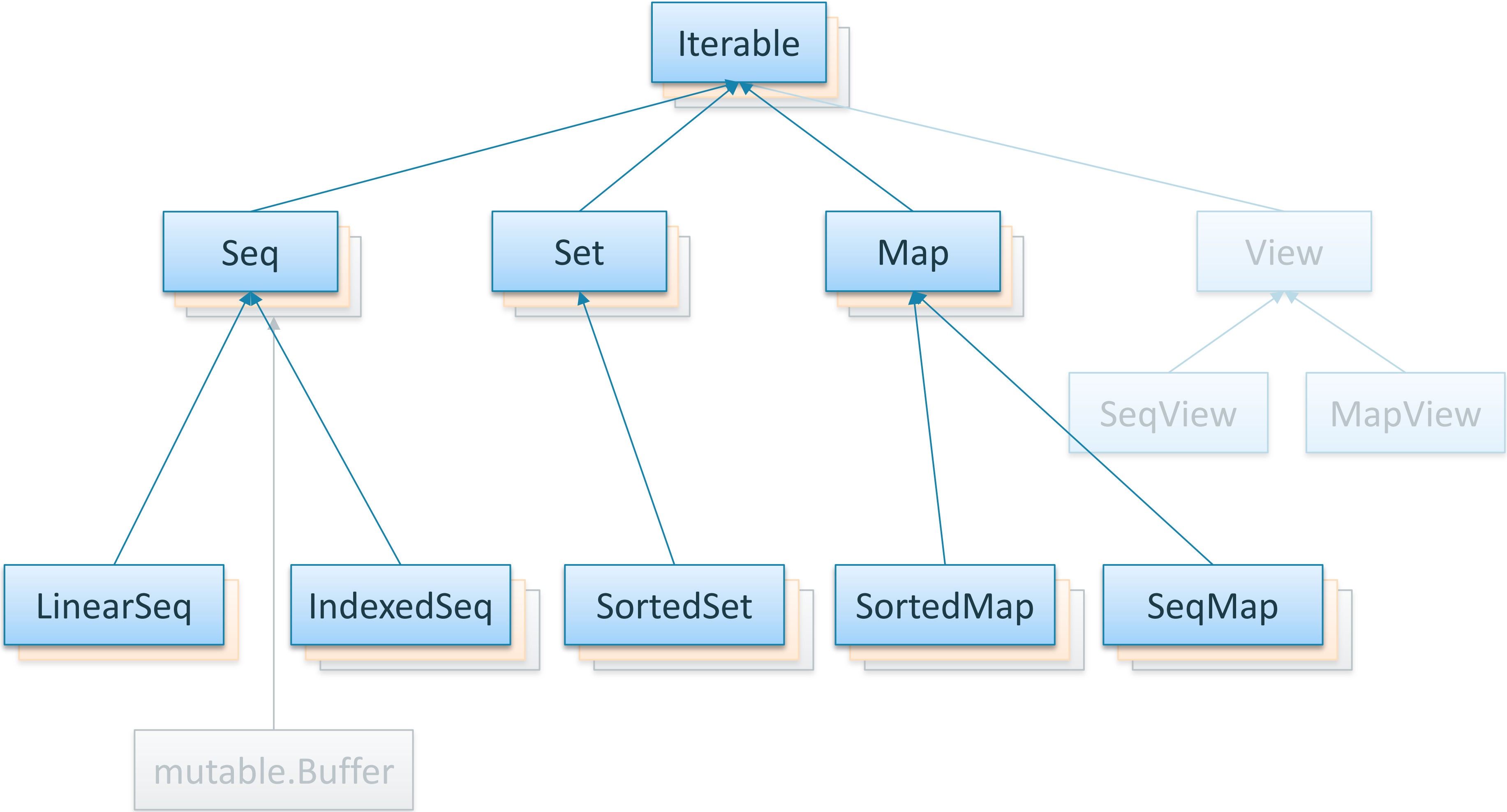
- Collection kinds capture what you need to instantiate types *and values*

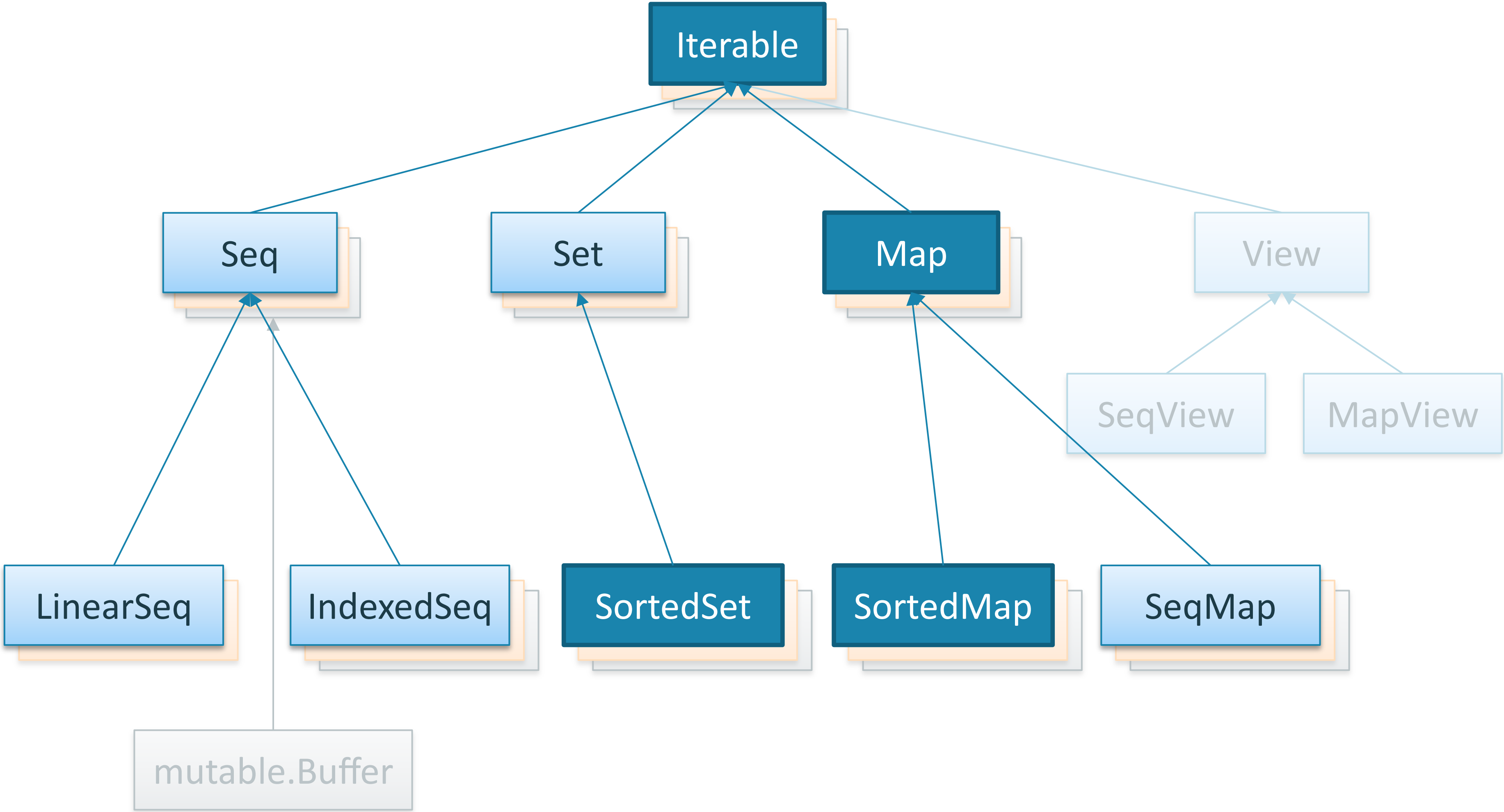
Standard Collection Kinds

	Collection Kind[*]	Collection	Factory
Iterable	* → *	✓	✓
Map	* → * → *	✓	✓
Evidence Iterable	(Ev @ (* → *)) → (* : Ev) → *		✓
Sorted Iterable	(* : Ordering) → *	✓	✓
Sorted Map	(* : Ordering) → * → *	✓	✓
Specific Iterable	*		✓

- Non-standard kinds can be implemented

[*] Improvised notation


Iterable
Seq
Set
Map
View
SeqView
MapView
LinearSeq
IndexedSeq
SortedSet
SortedMap
SeqMap
mutable.Buffer


Iterable
Seq
Set
Map
View
SeqView
MapView
LinearSeq
IndexedSeq
SortedSet
SortedMap
SeqMap
mutable.Buffer

Collection Kinds

```
trait   IterableOps [+A,      +CC[_],         +C     ]

trait   MapOps      [ K, +V,  +CC[_, _] <: …, +C     ]
extends IterableOps [(K,  V),  Iterable,       C     ]

trait   SortedSetOps[ A,      +CC[_] <: …,    +C <: …]
extends SetOps      [ A,       Set,            C     ]

trait   SortedMapOps[ K, +V,  +CC[_, _] <: …, +C <: …]
extends MapOps      [ K,  V,   Map,            C     ]
```

- Different `CC` type for each kind
- Collections must support all kinds of their parent types

Implementing a Map

Factory

```
object MyMap extends MapFactory[MyMap] {

  private[this] val _empty = new MyMap(new JHashMap[Any, Any])
  def empty[K, V]: MyMap[K,V] = _empty.asInstanceOf[MyMap[K, V]]

  def from[K, V](it: IterableOnce[(K, V)]): MyMap[K,V] =
    newBuilder[K, V].addAll(it).result()

  def newBuilder[K, V]: mutable.Builder[(K, V), MyMap[K,V]] =
    new mutable.Builder[(K, V), MyMap[K, V]] {
      private[this] val m = new JHashMap[K, V]
      def clear() = m.clear()
      def result() = new MyMap[K, V](m)
      def addOne(elem: (K, V)) = { m.put(elem._1, elem._2); this }
    }
}
```

Outline

```
class MyMap[K, +V] private (m: JHashMap[K, _ <: V])
  extends AbstractMap[K, V]
  with StrictOptimizedMapOps[K, V, MyMap, MyMap[K, V]]
  with MapFactoryDefaults[K, V, MyMap, immutable.Iterable] {

  override def mapFactory = MyMap
  //override def iterableFactory = immutable.Iterable

  ...
}
```

Implementation

```
class MyMap[K, +V] private (m: JHashMap[K, _ <: V]) ... {
  ...
  def iterator: Iterator[(K, V)] =
    m.entrySet().iterator().asScala.map(e => (e.getKey, e.getValue))
  def get(key: K): Option[V] = Option(m.get(key)).orElse {
    if(m.containsKey(key)) Some(null.asInstanceOf[V]) else None
  }
  def removed(key: K): MyMap[K,V] = if(!contains(key)) this else {
    val m2 = m.clone().asInstanceOf[JHashMap[K, V]]
    m2.remove(key)
    new MyMap(m2)
  }
  def updated[V1 >: V](key: K, value: V1): MyMap[K,V1] = {
    val m2 = m.clone().asInstanceOf[JHashMap[K, V1]]
    m2.put(key, value)
    new MyMap(m2)
  }
}
```

Overriding

```
class MyMap[K, +V] private (m: JHashMap[K, _ <: V]) ... {
  ...

  override def map[K2, V2](f: ((K, V)) => (K2, V2)):
    MyMap[K2, V2] = ...

  override def map[B](f: ((K, V)) => B):
    immutable.Iterable[B] = super.map(f)
}
```

- Methods that return a `CC` are overloaded
- Typical cases: `map`, `flatMap`, `collect`, `concat`

Non-Standard Collection Kinds

In the Standard Library

- AnyRefMap
- IntMap
- LongMap
- BitSet
- CollisionProofHashMap

immutable.BitSet

```scala
sealed abstract class BitSet
  extends AbstractSet[Int]
  with SortedSet[Int]
  with StrictOptimizedSortedSetOps[Int, SortedSet, BitSet]
  with collection.BitSet
  with collection.BitSetOps[BitSet]
  with Serializable {

  override protected def fromSpecific(coll: IterableOnce[Int]): BitSet =
    bitSetFactory.fromSpecific(coll)
  override protected def newSpecificBuilder: Builder[Int, BitSet] =
    bitSetFactory.newBuilder
  override def empty: BitSet = bitSetFactory.empty

  def bitSetFactory = BitSet
  ...
}
```

No …FactoryDefaults

mutable.CollisionProofHashMap

```
final class CollisionProofHashMap[K, V](...)(implicit ordering: Ordering[K])
  extends AbstractMap[K, V]
  with MapOps[K, V, Map, CollisionProofHashMap[K, V]]
  with StrictOptimizedIterableOps[(K, V), Iterable,
                                  CollisionProofHashMap[K, V]]
  with StrictOptimizedMapOps[K, V, Map, CollisionProofHashMap[K, V]] {

  private[this] final def sortedMapFactory:
    SortedMapFactory[CollisionProofHashMap] = CollisionProofHashMap  ...
}
```

- Reuse SortedMapFactory
- But not SortedMap

immutable.IntMap

```
sealed abstract class IntMap[+T]
  extends AbstractMap[Int, T]
  with StrictOptimizedMapOps[Int, T, Map, IntMap[T]]
  with Serializable {

  protected def intMapFrom[V2](coll: IterableOnce[(Int, V2)]):
    IntMap[V2] = ...

  def map[V2](f: ((Int, T)) => (Int, V2)): IntMap[V2] =
    intMapFrom(new View.Map(toIterable, f))  ...
}
```

- Typical overloads: `map`, `flatMap`, `collect`, `concat`
- Good baseline implementation: wrap a `View` with the appropriate `from` method

IntMap Factory

```
object IntMap {
  def empty[T] : IntMap[T] = ...
  def singleton[T](key: Int, value: T): IntMap[T] = ...
  def apply[T](elems: (Int, T)*): IntMap[T] = ...
  def from[V](coll: IterableOnce[(Int, V)]): IntMap[V] = ...
  def newBuilder[V]: Builder[(Int, V), IntMap[V]] = ...

  ...
}
```

- No standard factory type
- Methods up to the collection implementor

BuildFrom & Factory

- `Factory` enables static type-driven collection building
- `BuildFrom` enables dynamic type-driven collection building
  - based on an existing object of the same type
- Implementations are generally the same for *concrete* collection types

	Implicit Conversion of Companion Object	Implicit Instance for Type

`Factory`	**`List(1 -> "a")`**
	`.to(IntMap)`
		`implicitly[`
		`Factory[Int, List[Int]]`
		`]`

`BuildFrom`	`Future.sequence(xs)`
	`(List)`
		`val xs: Iterable[Future[Int]]`
		`Future.sequence(xs)`

IntMap Factory

```
object IntMap { ...
  implicit def toFactory[V](dummy: IntMap.type): Factory[(Int, V), IntMap[V]] =
    ToFactory.asInstanceOf[Factory[(Int, V), IntMap[V]]]

  private[this] object ToFactory
    extends Factory[(Int, AnyRef), IntMap[AnyRef]] with Serializable {
    def fromSpecific(it: IterableOnce[(Int, AnyRef)]): IntMap[AnyRef] =
      IntMap.from[AnyRef](it)
    def newBuilder: Builder[(Int, AnyRef), IntMap[AnyRef]] =
      IntMap.newBuilder[AnyRef]
  }

  implicit def iterableFactory[V]: Factory[(Int, V), IntMap[V]] =
    toFactory(this)
}
```

- Implicit Factory and implicit conversion to Factory

IntMap Factory

```
object IntMap { ...
  implicit def toBuildFrom[V](factory: IntMap.type):
    BuildFrom[Any, (Int, V), IntMap[V]] =
    ToBuildFrom.asInstanceOf[BuildFrom[Any, (Int, V), IntMap[V]]]

  private[this] object ToBuildFrom
    extends BuildFrom[Any, (Int, AnyRef), IntMap[AnyRef]] {
    def fromSpecific(from: Any)(it: IterableOnce[(Int, AnyRef)]) =
      IntMap.from(it)
    def newBuilder(from: Any) = IntMap.newBuilder[AnyRef]
  }

  implicit def buildFromIntMap[V]:
    BuildFrom[IntMap[_], (Int, V), IntMap[V]] = toBuildFrom(this)
}
```

- The same for BuildFrom

Summary

Summary

- Integration of non-collection types via `IterableOnce`
- Unboxed and parallel iteration with `Stepper`
- Use of collection factories
- No special treatment of Iterable-kinded collections anymore
    - Abstractions for some standard kinds is provided
    - Non-standard kinds can be added
- Factory and `BuildFrom` replace `CanBuildFrom`

Links

- The Architecture of Scala 2.13's Collections
  https://docs.scala-lang.org/overviews/core/architecture-of-scala-213-collections.html
  - and other collection docs on docs.scala-lang.org
- Demo project and slides:
  https://github.com/szeiger/implementing-scala-collections
- 🐦 @StefanZeiger