Imperative Programming 2:
Scala's collection library

Hongseok Yang
University of Oxford

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

scala> val b = Map(2->4,3->9)
b: scala.collection.immutable.Map[Int,Int] = Map(2->4, 3->9)

scala> a map (_ + 1)
res0: scala.collection.immutable.Set[Int] = Set(2, 3)

scala> b map (x => (x._2->x._1))
res1: scala.collection.immutable.Map[Int,Int] = Map(4->2, 9->3)
```

Scala has a rich set of collection classes.

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

scala> val b = Map(2->4,3->9)
b: scala.collection.immutable.Map[Int,Int] = Map(2->4, 3->9)

scala> a map (_ + 1)
res0: scala.collection.immutable.Set[Int] = Set(2, 3)

scala> b map (x => (x._2->x._1))
res1: scala.collection.immutable.Map[Int,Int] = Map(4->2, 9->3)
```

Scala has a rich set of collection classes.

Usually, these classes support powerful operations, such as those for lists in Haskell.

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

scala> val b = Map(2->4,3->9)
b: scala.collection.immutable.Map[Int,Int] = Map(2->4, 3->9)

scala> a map (_ + 1)
res0: scala.collection.immutable.Set[Int] = Set(2, 3)

scala> b map (x => (x._2->x._1))
res1: scala.collection.immutable.Map[Int,Int] = Map(4->2, 9->3)
```

Scala has a rich set of collection classes.

Usually, these classes support powerful operations, such as those for lists in Haskell.

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

scala> val b = Map(2->4,3->9)
b: scala.collection.immutable.Map[Int,Int] = Map(2->4, 3->9)

scala> a map (_ + 1)
res0: scala.collection.immutable.Set[Int] = Set(2, 3)

scala> b map (x => (x._2->x._1))
res1: scala.collection.immutable.Map[Int,Int] = Map(4->2, 9->3)
```

Scala has a rich set of collection classes.

Usually, these classes support powerful operations, such as those for lists in Haskell.

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

scala> val b = Map(2->4,3->9)
b: scala.collection.immutable.Map[Int,Int] = Map(2->4, 3->9)

scala> a map (_ + 1)
res0: scala.collection.immutable.Set[Int] = Set(2, 3)

scala> b map (x => (x._2->x._1))
res1: scala.collection.immutable.Map[Int,Int] = Map(4->2, 9->3)

scala> b map (x => x._2)
```

[Q] What would happen?

Scala has a rich set of collection classes.

Usually, these classes support powerful operations, such as those for lists in Haskell.

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

scala> val b = Map(2->4,3->9)
b: scala.collection.immutable.Map[Int,Int] = Map(2->4, 3->9)

scala> a map (_ + 1)
res0: scala.collection.immutable.Set[Int] = Set(2, 3)

scala> b map (x => (x._2->x._1))
res1: scala.collection.immutable.Map[Int,Int] = Map(4->2, 9->3)

scala> b map (x => x._2)
res2: scala.collection.immutable.Iterable[Int] = List(4, 9)
```

[Q] What would happen?

[A] Change in type.

Scala has a rich set of collection classes.

Usually, these classes support powerful operations, such as those for lists in Haskell.

But one implementation of map.

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

scala> val b = Map(2->4,3->9)
b: scala.collection.immutable.Map[Int,Int] = Map(2->4, 3->9)

scala> a map (_ + 1)
res0: scala.collection.immutable.Set[Int] = Set(2, 3)

scala> b map (x => (x._2->x._1))
res1: scala.collection.immutable.Map[Int,Int] = Map(4->2, 9->3)

scala> b map (x => x._2)
res2: scala.collection.immutable.Iterable[Int] = List(4, 9)
```

[Q] What would happen?

[A] Change in type.

Learning outcome

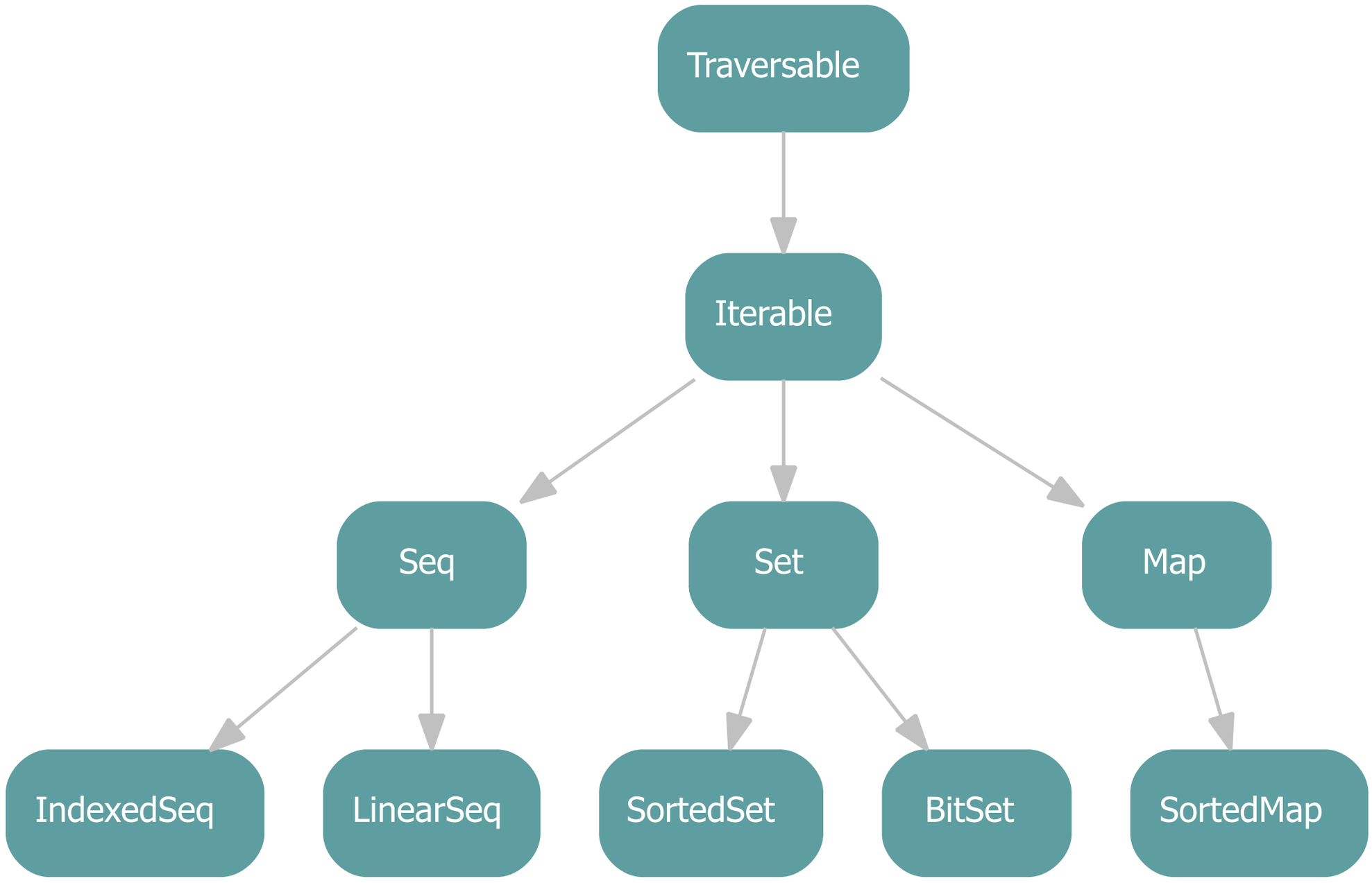
- Can explain a big picture about Scala collections.
- Can describe the design of Scala collections, and use it for solving other problems.

Overview of the collection library

Big picture I

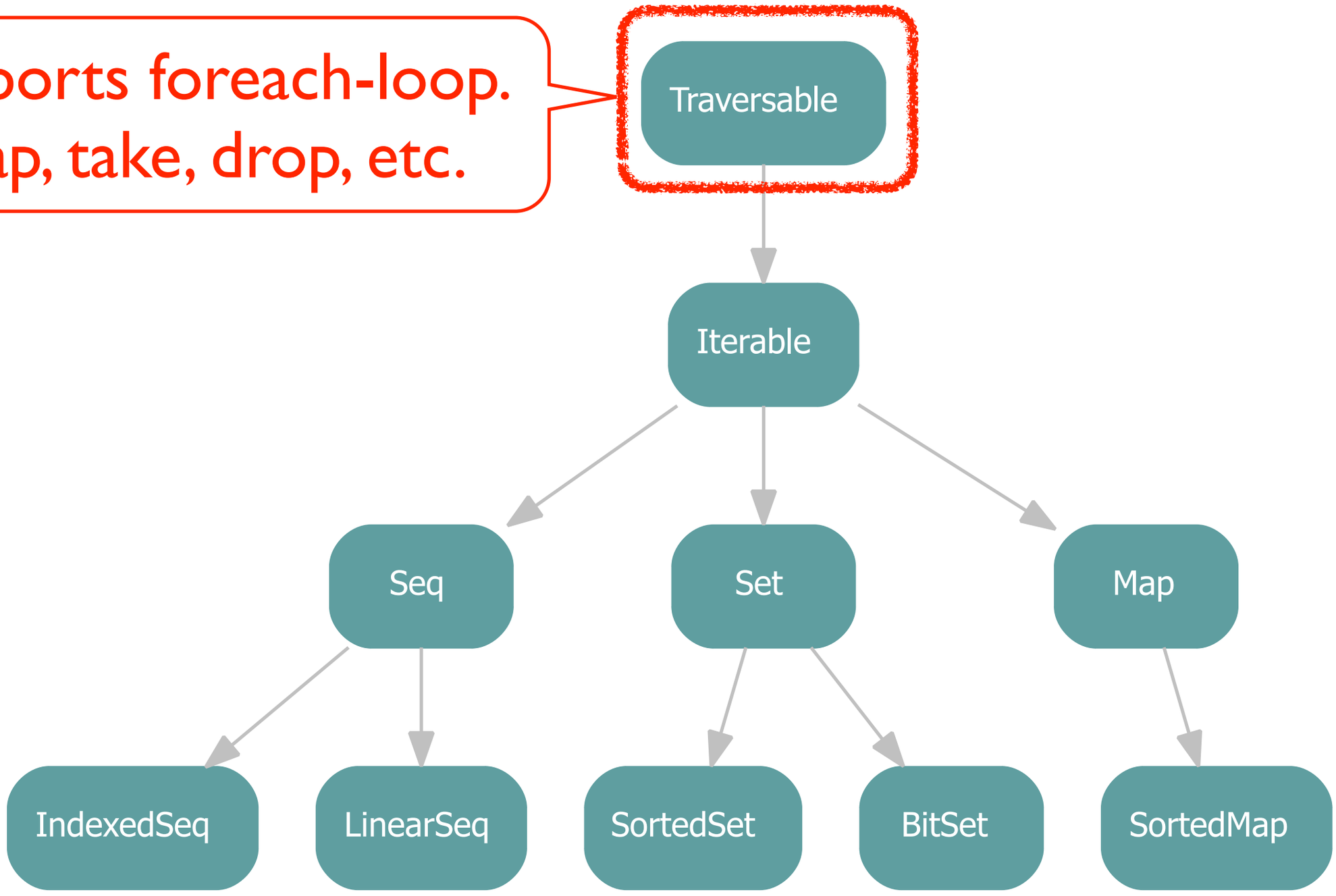
- Each collection typically comes with three versions: mutable, immutable, general.

Big picture 2



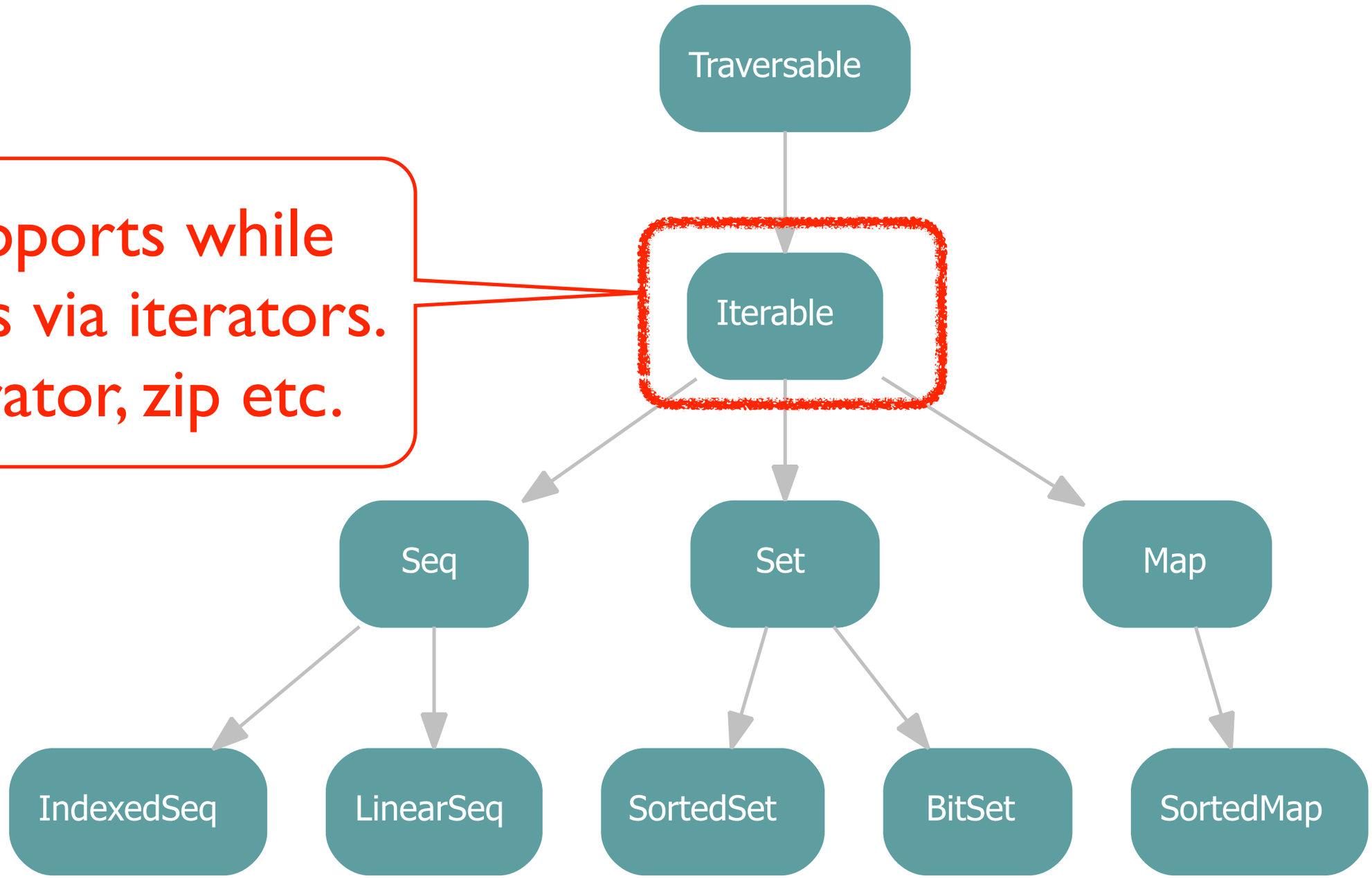
Big picture 2

Supports foreach-loop.
map, take, drop, etc.



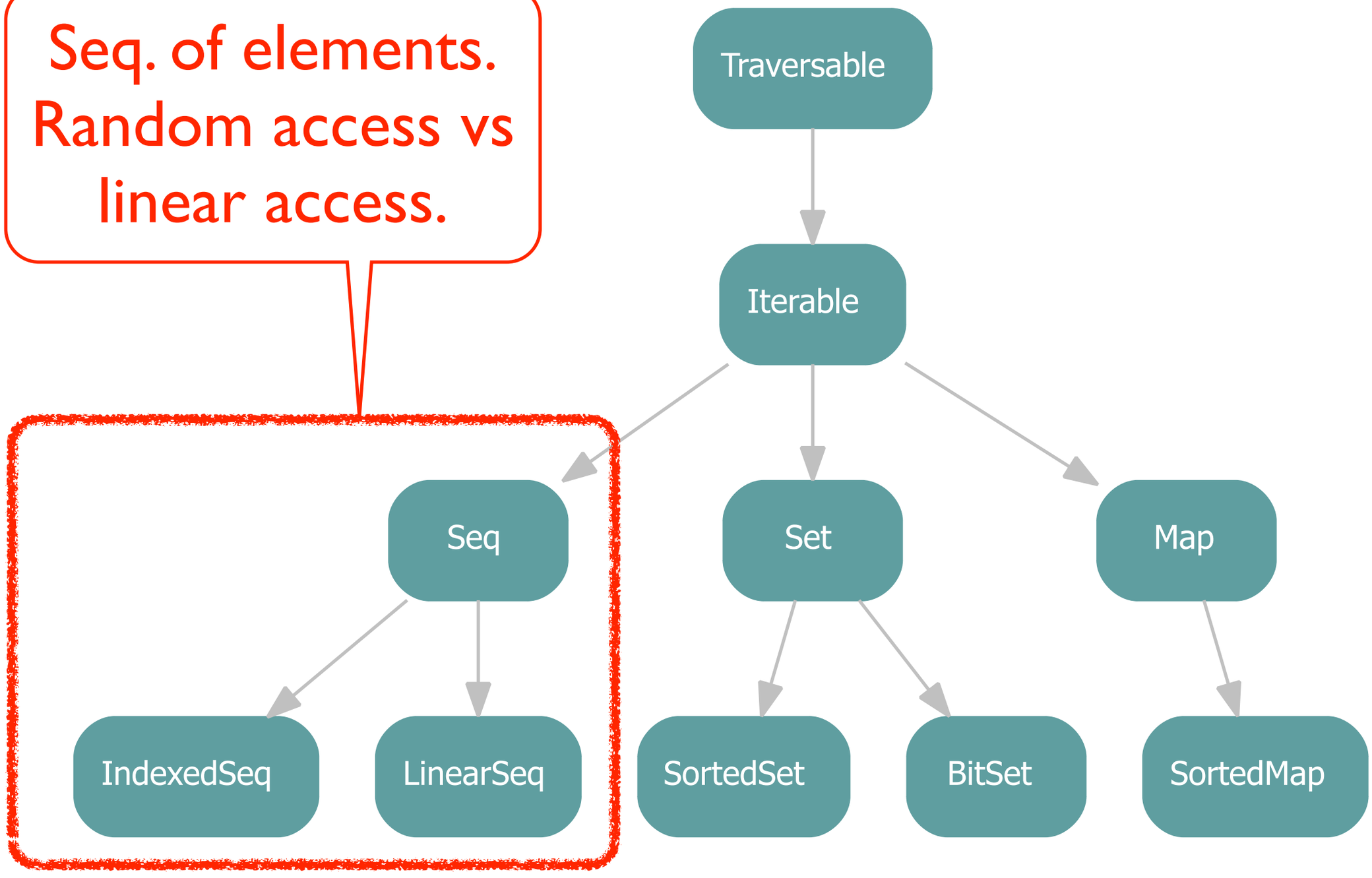
Big picture 2

Supports while loops via iterators. iterator, zip etc.

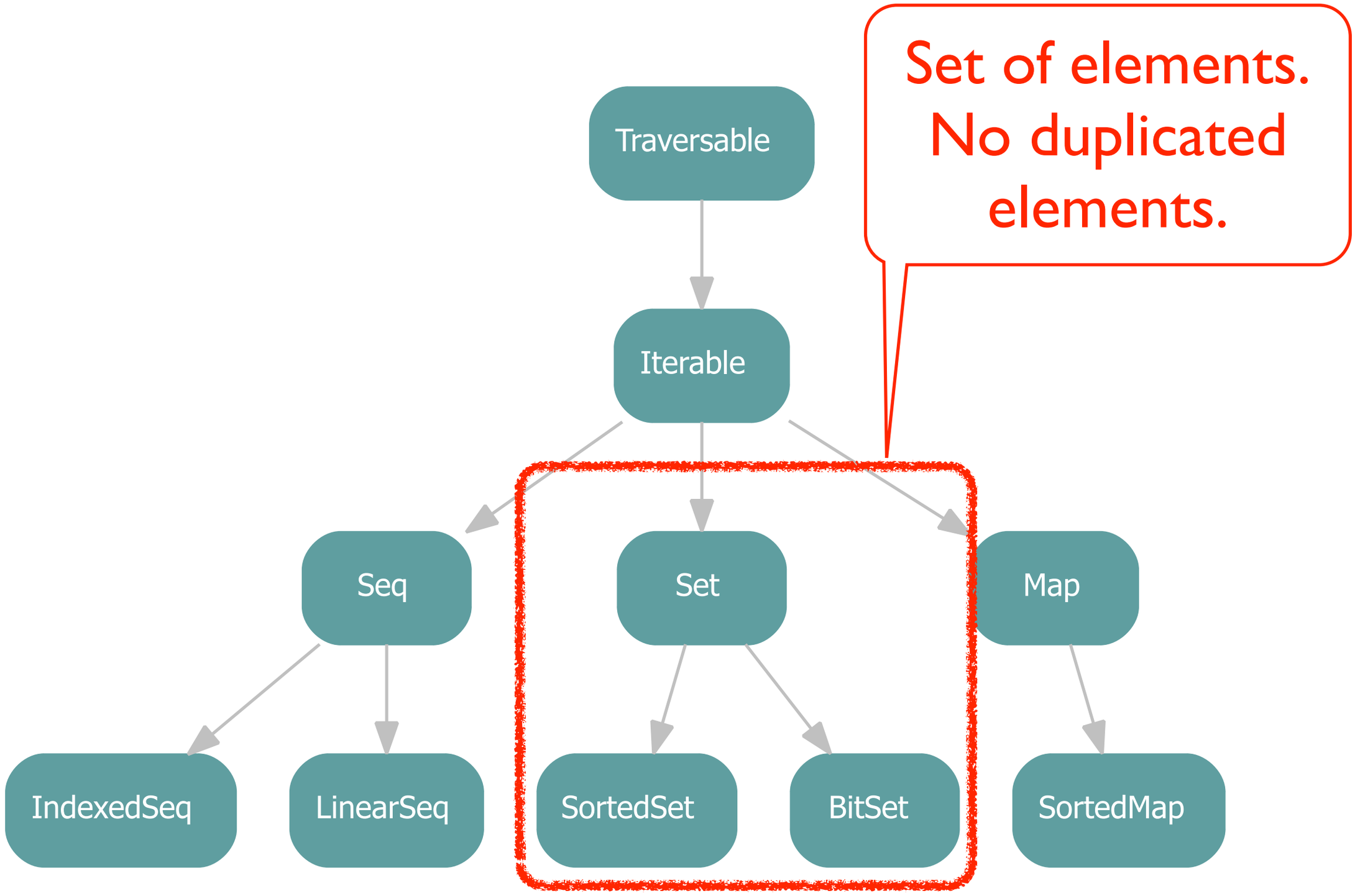


Big picture 2

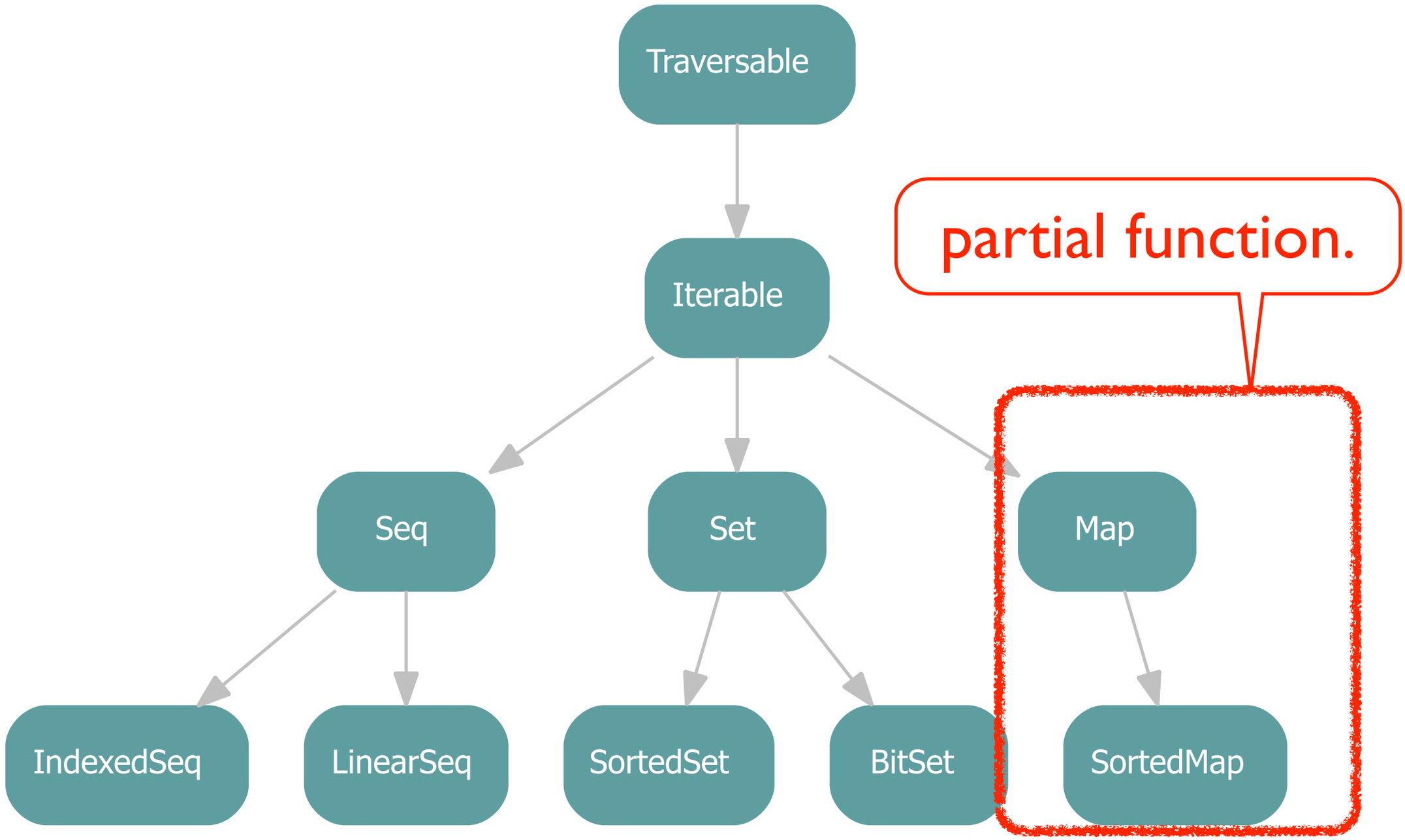
Seq. of elements.
Random access vs
linear access.



Big picture 2



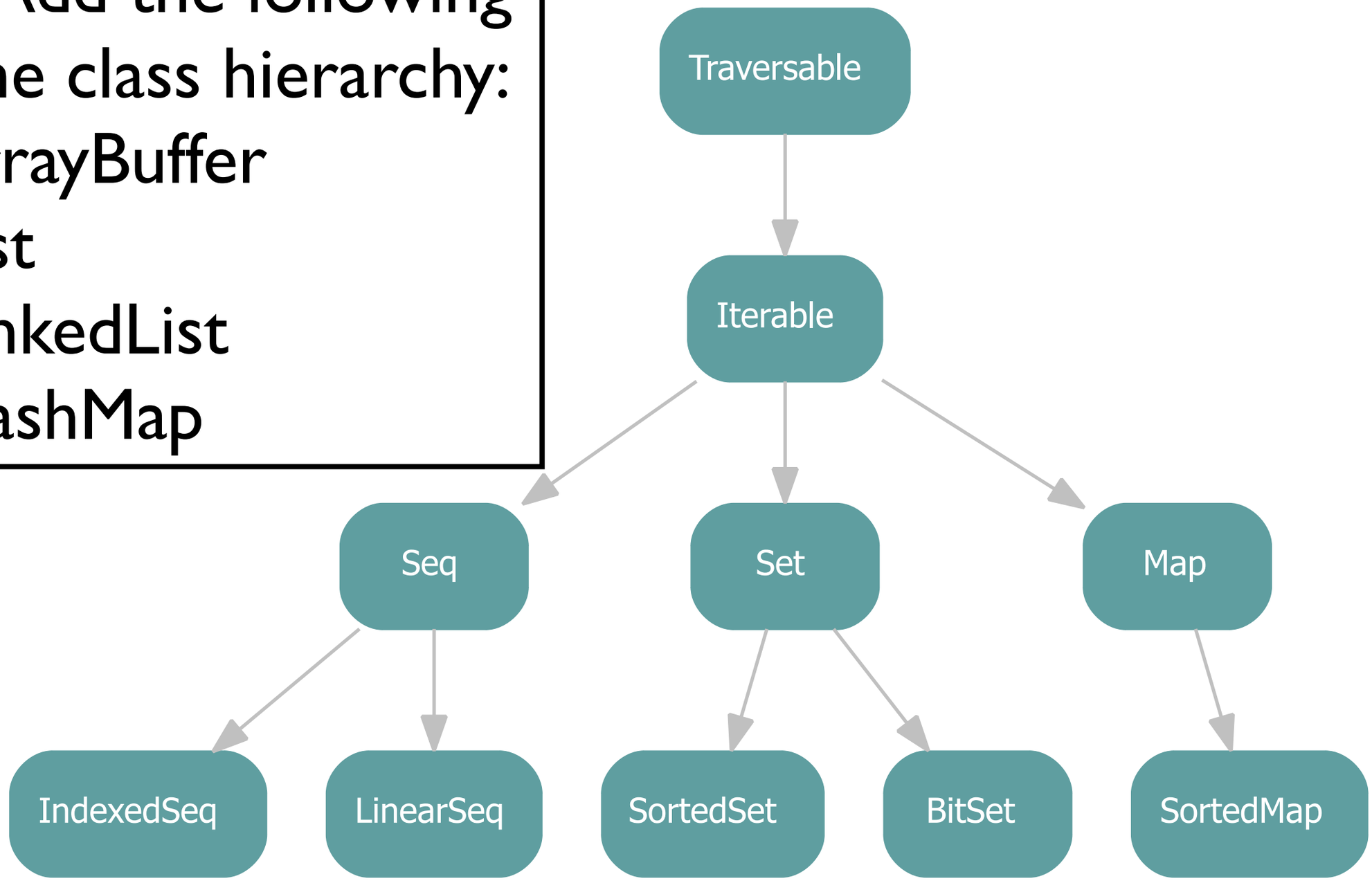
Big picture 2



Big picture 2

[Q] Add the following to the class hierarchy:

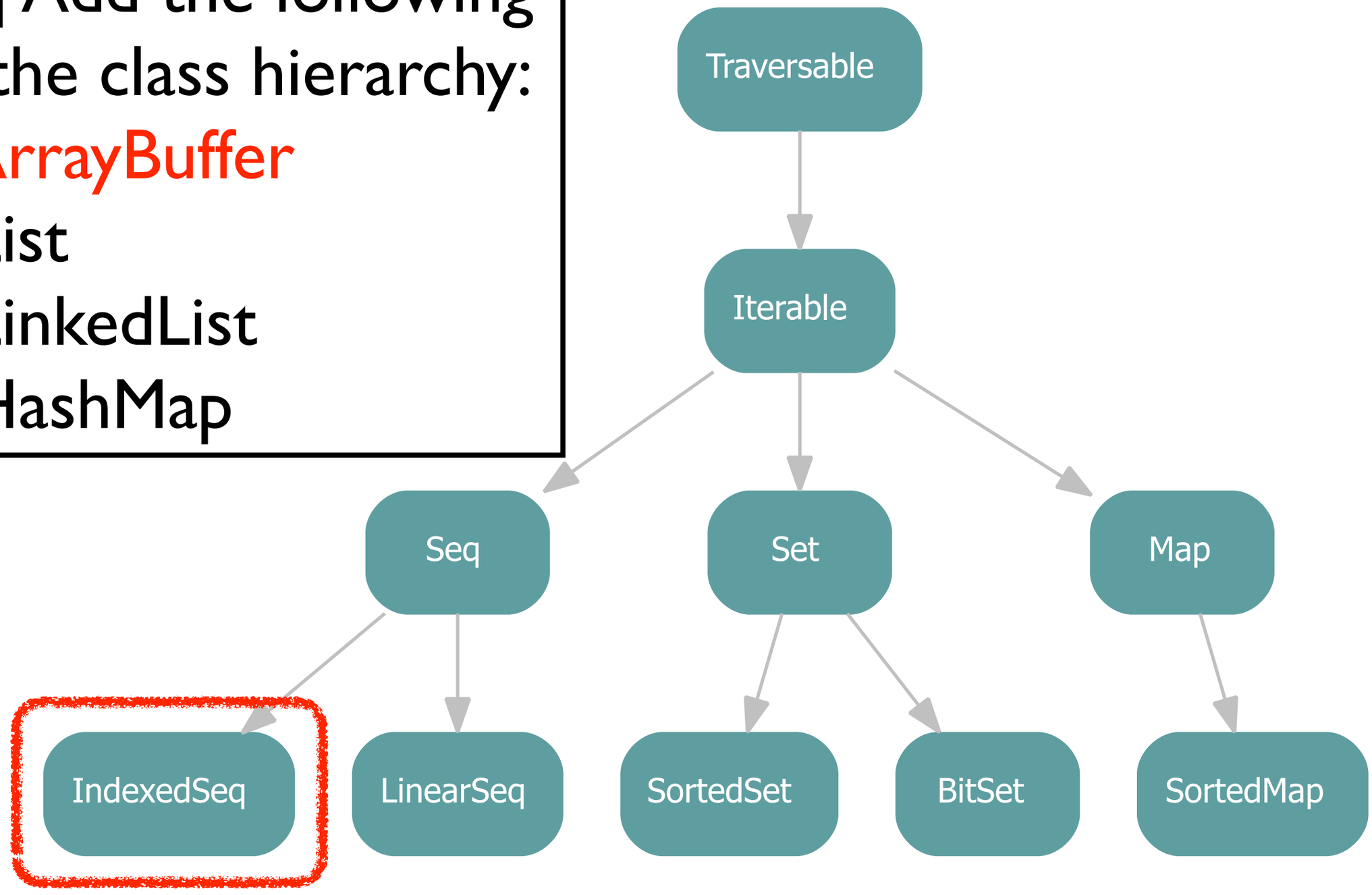
1. ArrayBuffer
2. List
3. LinkedList
4. HashMap



Big picture 2

[Q] Add the following to the class hierarchy:

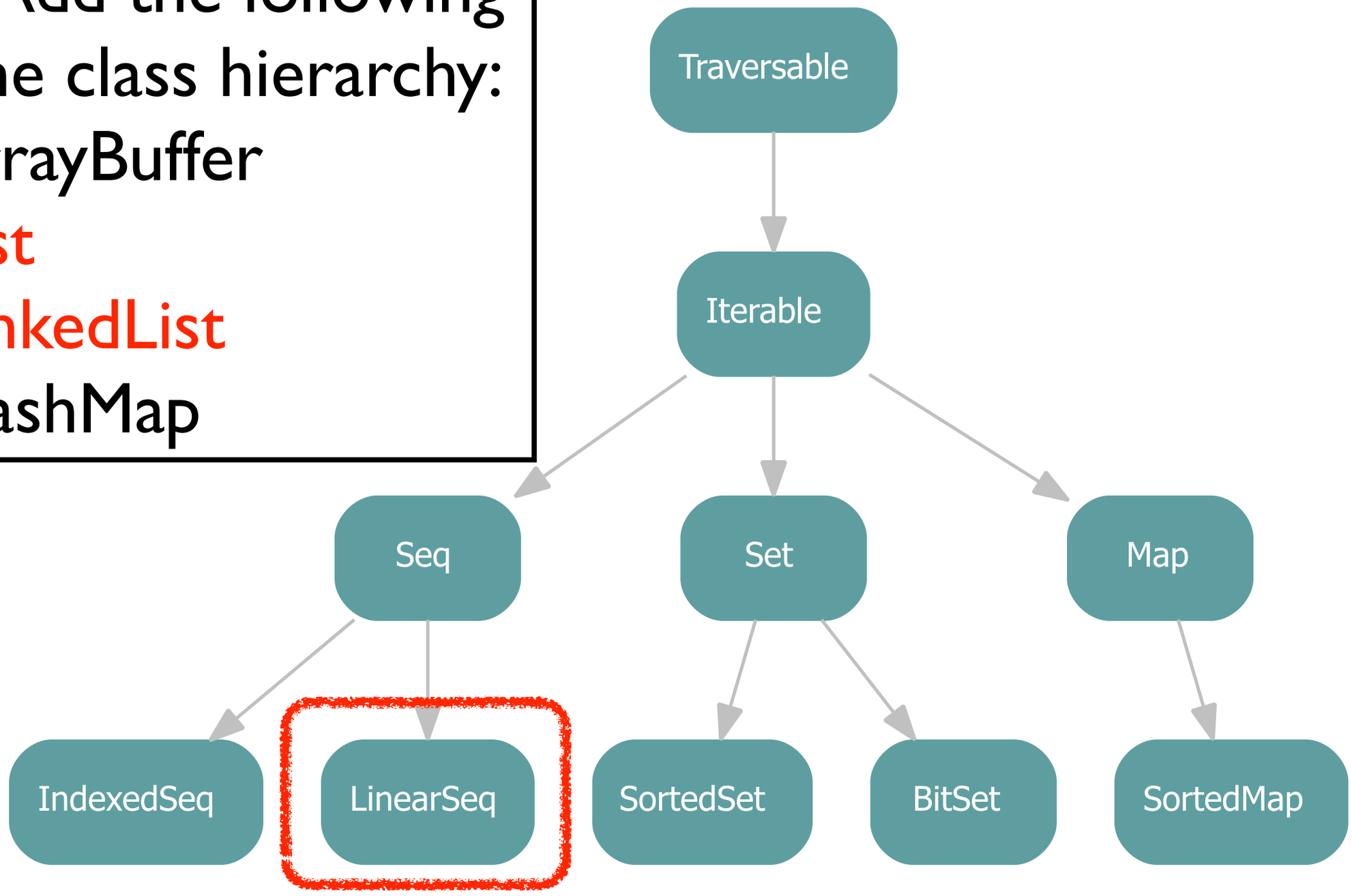
1. ArrayBuffer
2. List
3. LinkedList
4. HashMap



Big picture 2

[Q] Add the following to the class hierarchy:

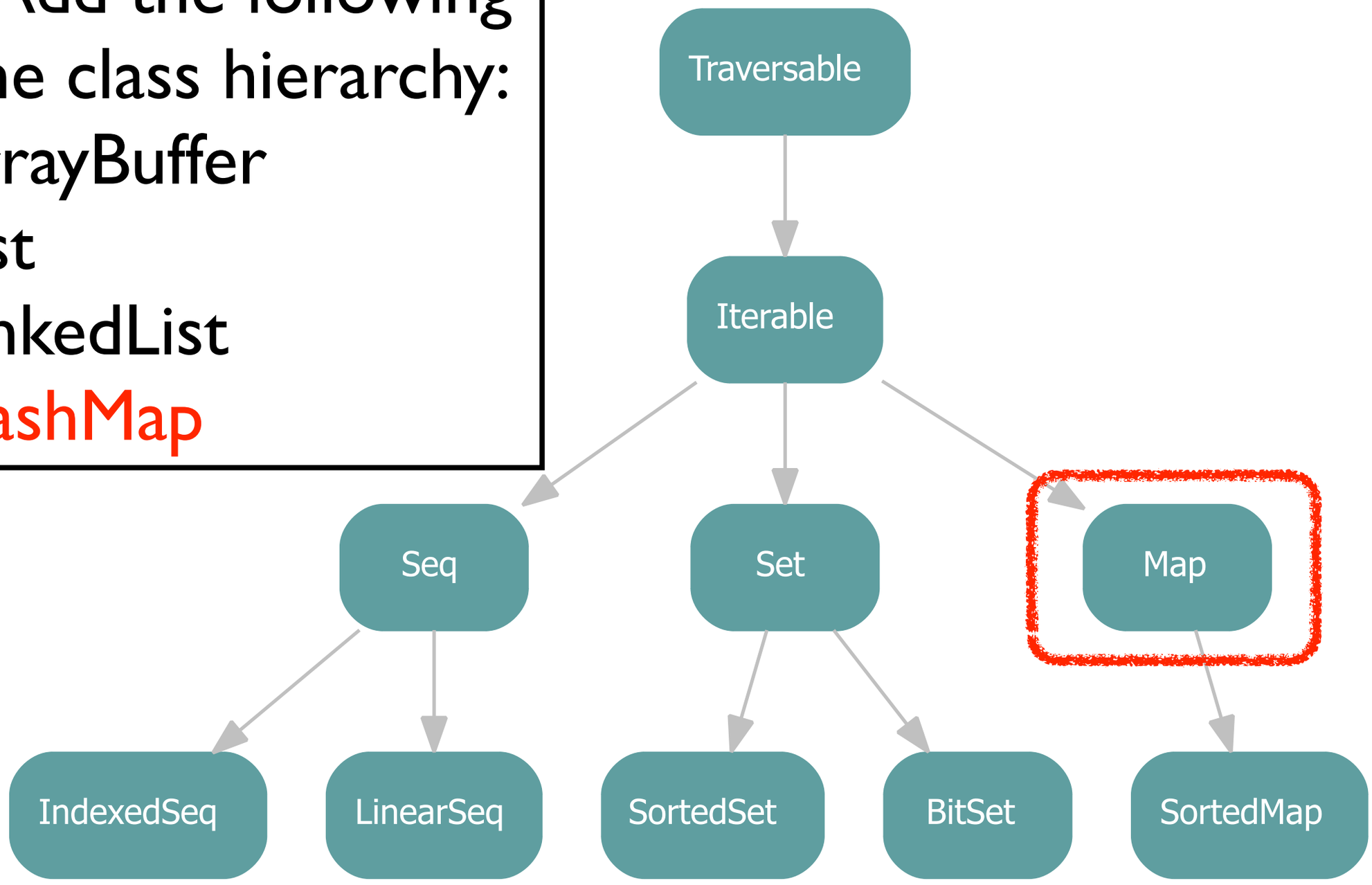
1. ArrayBuffer
2. List
3. LinkedList
4. HashMap



Big picture 2

[Q] Add the following to the class hierarchy:

1. ArrayBuffer
2. List
3. LinkedList
4. HashMap

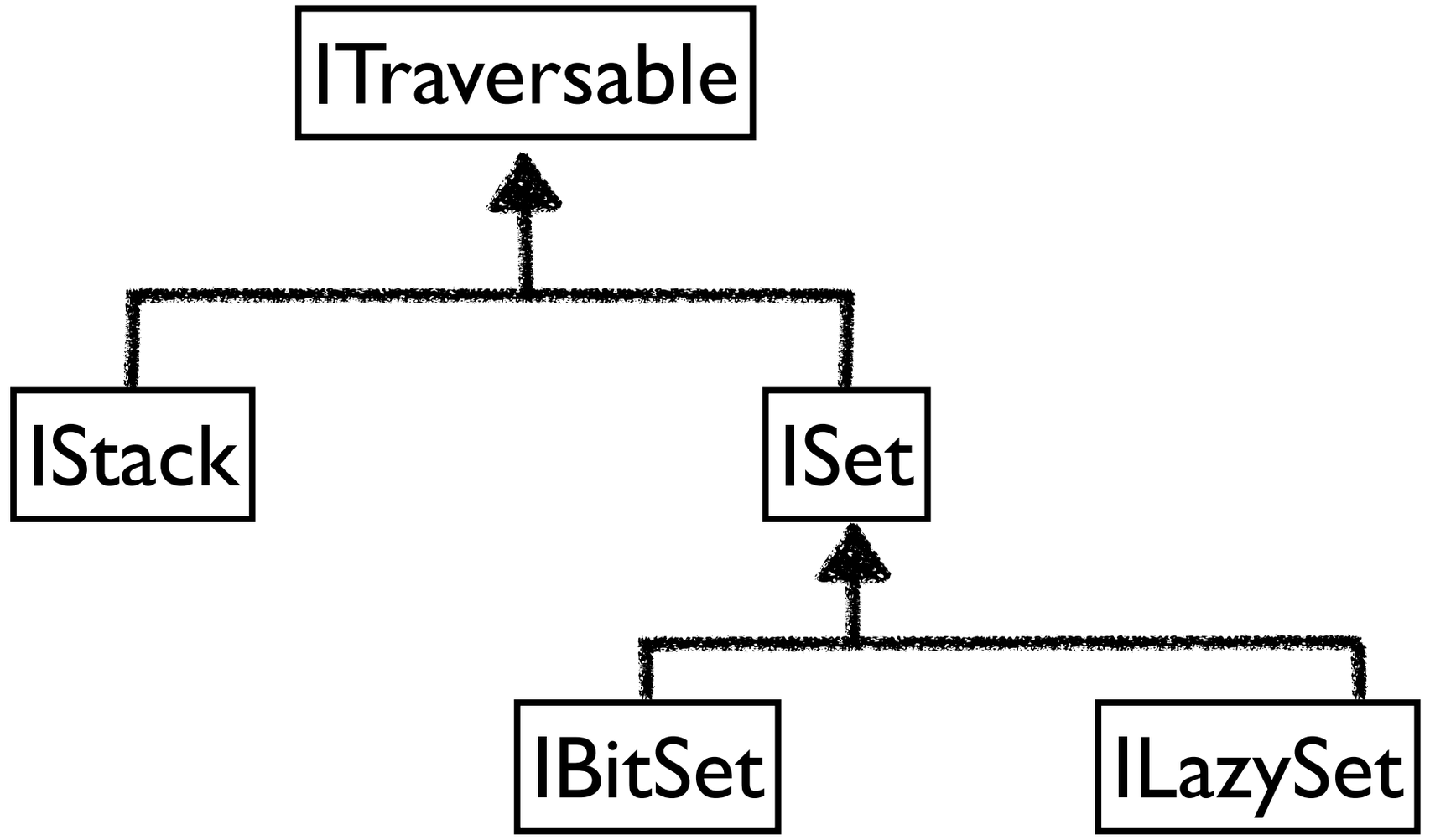


Using collection classes

- Decide a semantics of your collection.
- Decide whether you want to use a mutable or immutable version.
- Look at: http://www.scala-lang.org/api/current/index.html#package

Design of the collection library

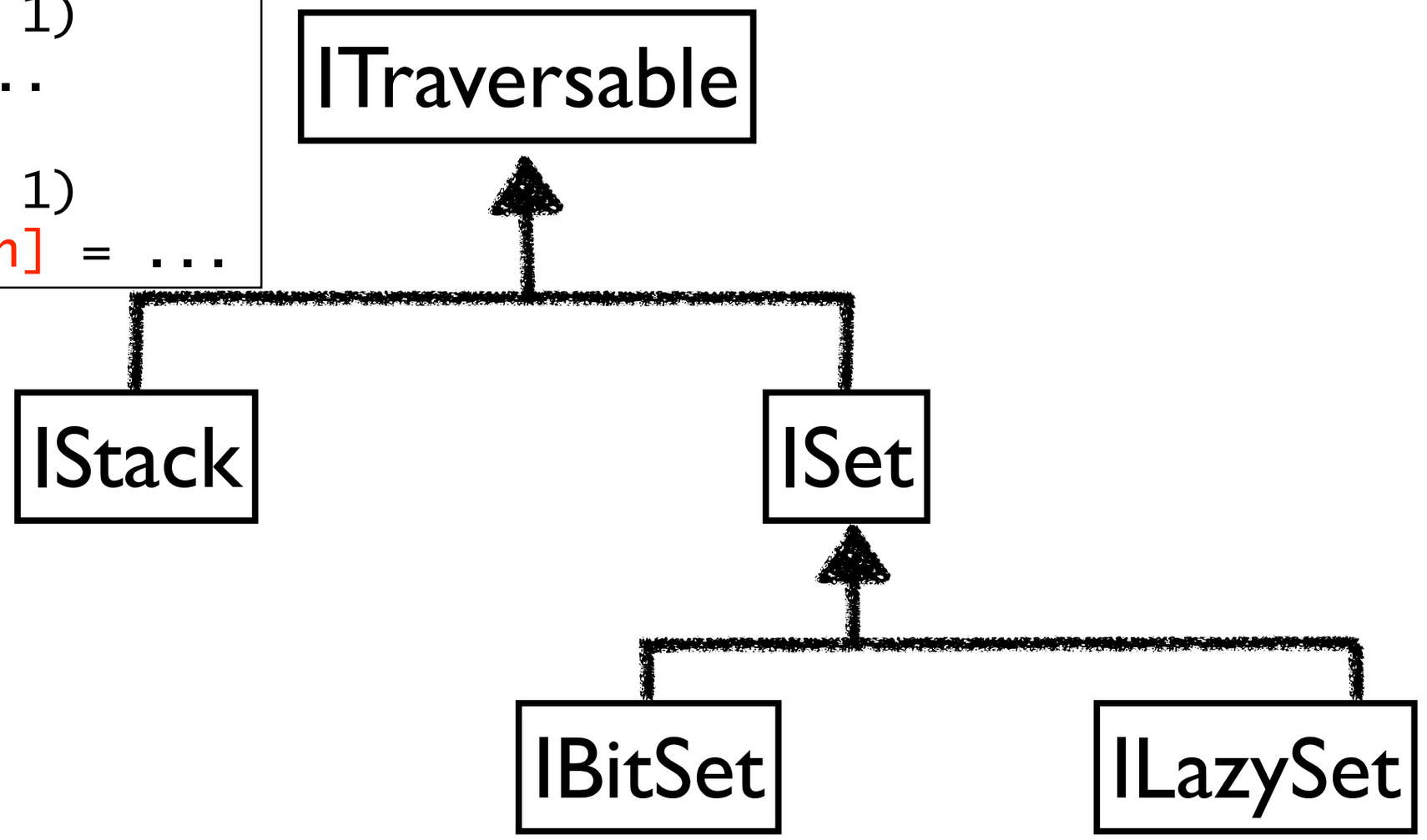
IP2 collection library



- Immutable stack, mutable sets.
- Support exists, filter, map, etc.
- Minimal code duplication. But type-accurate.

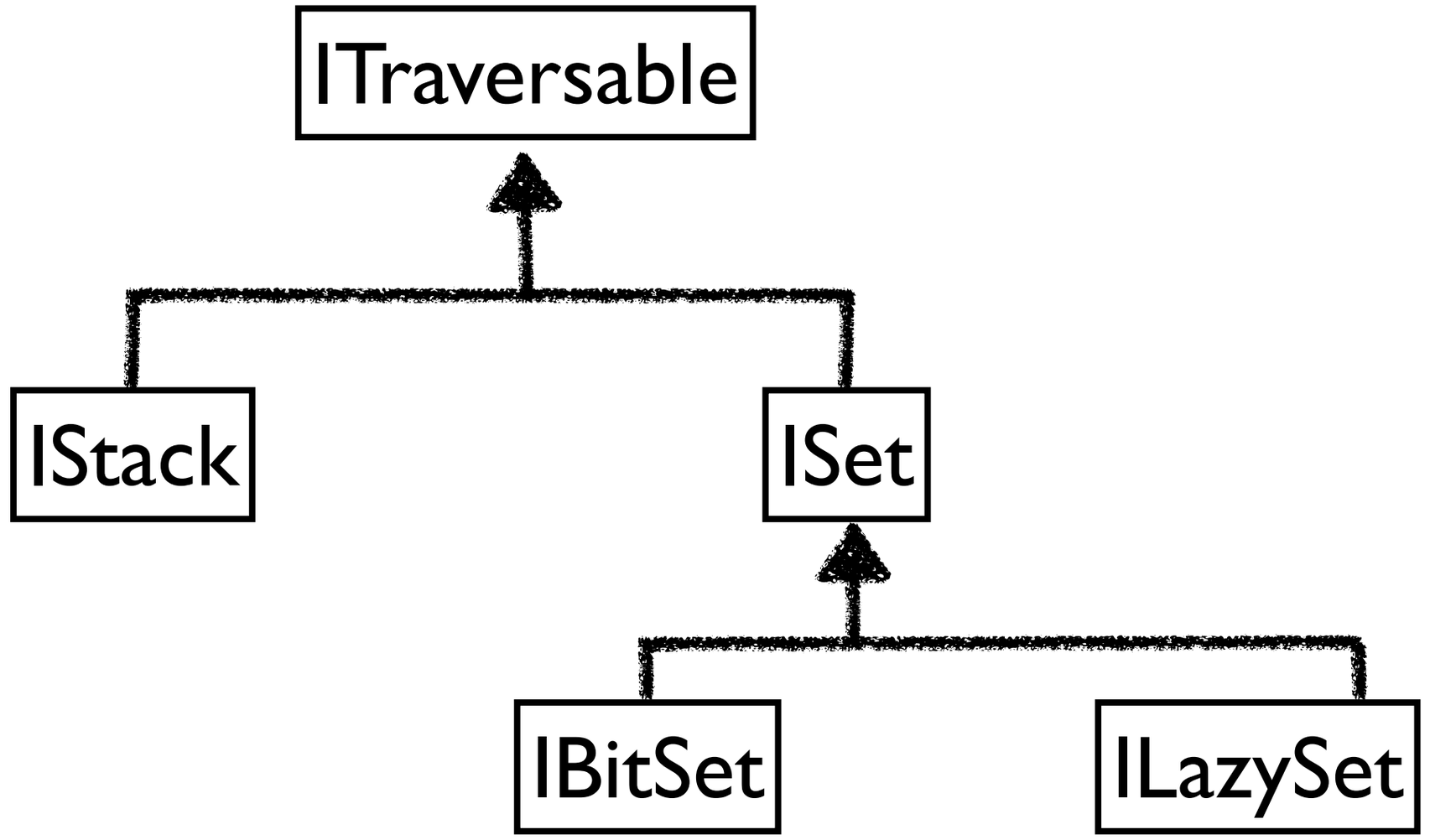
lection library

```
scala> val x = new BitSet
x: IBitSet = ...

...

scala> x map (_ + 1)
res0: IBitSet = ...

scala> x map (_ > 1)
res1: ISet[Boolean] = ...
```



- Immutable stack, mutable sets.
- Support exists, filter, map, etc.
- Minimal code duplication. But type-accurate.

Our approach



- Put common code in ITraversable.
- Specialise this code in subclasses.

Version 1.0

- ITraversable[A] has the following methods:

```
exists(p: A=>Boolean): Boolean
forall(p: A=>Boolean): Boolean
last: A
```

- Usual stack or set operations in IStack[A], ISet[A] and IBitSet.

Idea: Abstract for loop

- Add an abstract method to ITraversable[T]:

```
foreach[U](f: T=>U): Unit
```

- It models the for loop, and applies `f` to every element in a collection.

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit

  def forall(p: A => Boolean): Boolean = ...
  def exists(p: A => Boolean): Boolean = ...
  def last: A = ...
}
class IStack[A](val cont: List[A]) extends ITraversable[A] ...
class ISet[A] extends ITraversable[A] ...
class IBitSet extends ISet[Int] ...
```

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit

  def forall(p: A => Boolean): Boolean = {
    var result = true
    foreach {x => if (!(p(x))) result=false }
    result
  }
  def exists(p: A => Boolean): Boolean = ...
  def last: A = ...
}
class IStack[A](val cont: List[A]) extends ITraversable[A] ...
class ISet[A] extends ITraversable[A] ...
class IBitSet extends ISet[Int] ...
```

Define `exists` and `last`.

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit

  def forall(p: A => Boolean): Boolean = {
    var result = true
    foreach {x => if (!(p(x))) result=false }
    result
  }
  def exists(p: A => Boolean): Boolean = ...

  def last: A = ...

}
class IStack[A](val cont: List[A]) extends ITraversable[A] ...
class ISet[A] extends ITraversable[A] ...
class IBitSet extends ISet[Int] ...
```

Define `exists` and `last`.

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit

  def forall(p: A => Boolean): Boolean = {
    var result = true
    foreach {x => if (!(p(x))) result=false }
    result
  }
  def exists(p: A => Boolean): Boolean = {
    var result = false
    foreach {x => if (p(x)) result=true }
    result
  }
  def last: A = ...




}
class IStack[A](val cont: List[A]) extends ITraversable[A] ...
class ISet[A] extends ITraversable[A] ...
class IBitSet extends ISet[Int] ...
```

Define `exists` and `last`.

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit

  def forall(p: A => Boolean): Boolean = {
    var result = true
    foreach {x => if (!(p(x))) result=false }
    result
  }
  def exists(p: A => Boolean): Boolean = {
    var result = false
    foreach {x => if (p(x)) result=true }
    result
  }
  def last: A = {
    var lst: Option[A] = None
    foreach {x => lst=Some(x)}
    lst match {
      case None => throw new NoSuchElementException
      case Some(x) => x }
  }
}
class IStack[A](val cont: List[A]) extends ITraversable[A] ...
class ISet[A] extends ITraversable[A] ...
class IBitSet extends ISet[Int] ...
```

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit
  ...
}
class IStack[A](val cont: List[A]) extends ITraversable[A] = {
  def push(x: A): IStack[A] = ...
  def pop: (A, IStack[A]) = ...
  def foreach[U](f: A => U) ...



}
class ISet[A] extends ITraversable[A] = ...
class IBitSet extends ISet[Int] = ...
```

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit
  ...
}
class IStack[A](val cont: List[A]) extends ITraversable[A] = {
  def push(x: A): IStack[A] = new IStack(x::cont)
  def pop: (A, IStack[A]) = (cont.head, new IStack(cont.tail))
  def foreach[U](f: A => U) ...



}
class ISet[A] extends ITraversable[A] = ...
class IBitSet extends ISet[Int] = ...
```

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit
  ...
}
class IStack[A](val cont: List[A]) extends ITraversable[A] = {
  def push(x: A): IStack[A] = new IStack(x::cont)
  def pop: (A, IStack[A]) = (cont.head, new IStack(cont.tail))
  def foreach[U](f: A => U) {
    var l = cont
    while(!l.isEmpty) { f(l.head); l = l.tail }
  }
}
class ISet[A] extends ITraversable[A] = ...
class IBitSet extends ISet[Int] = ...
```

Implement foreach in ISet and IBitSet

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit
  ...
}
class IStack[A](val cont: List[A]) extends ITraversable[A] = { .. }
class ISet[A] extends ITraversable[A] = {
  var cont: List[A] = Nil
  ...
  def foreach[U](f: A => U) ...



}
class IBitSet extends ISet[Int] = {
  val bcont: Array[Boolean] = new Array(100)
  ...
  override def foreach[U](f: Int => U) ...



}
```

Implement foreach in ISet and IBitSet

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit
  ...
}
class IStack[A](val cont: List[A]) extends ITraversable[A] = { .. }
class ISet[A] extends ITraversable[A] = {
  var cont: List[A] = Nil
  ...
  def foreach[U](f: A => U) {
    var l = cont
    while(!l.isEmpty) { f(l.head); l = l.tail }
  }
}
class IBitSet extends ISet[Int] = {
  val bcont: Array[Boolean] = new Array(100)
  ...
  override def foreach[U](f: Int => U) ...









}
```

Implement foreach in ISet and IBitSet

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit
  ...
}
class IStack[A](val cont: List[A]) extends ITraversable[A] = { .. }
class ISet[A] extends ITraversable[A] = {
  var cont: List[A] = Nil
  ...
  def foreach[U](f: A => U) {
    var l = cont
    while(!l.isEmpty) { f(l.head); l = l.tail }
  }
}
class IBitSet extends ISet[Int] = {
  val bcont: Array[Boolean] = new Array(100)
  ...
  override def foreach[U](f: Int => U) {
    var i = 0
    while (i < bcont.length) {
      if (bcont(i)) f(i)
      i += 1
    }
  }
}
```

Version 2.0

- Add the following methods to Traversable:

```
filter(p: A=>Boolean): ColTy[A]

partition(p: A=>Boolean): (ColTy[A], ColTy[A])

dropWhile(p: A=>Boolean): ColTy[A]
```

- Here `ColTy[A]` is the type of the receiver object.

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit
  ..
}
class IStack[A](val cont: List[A]) extends ITraversable[A] ..
class ISet[A] extends ITraversable[A] ..
class IBitSet extends ISet[Int] ..
```

No code duplication. But types are not as accurate as we want.

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit
  ..
  def filter(p:A=>Boolean):ITraversable[A] ..
  def partition(p:A=>Boolean):(ITraversable[A],ITraversable[A]) ..
  def dropWhile(p:A=>Boolean):ITraversable[A] ..
}
class IStack[A](val cont: List[A]) extends ITraversable[A] ..
class ISet[A] extends ITraversable[A] ..
class IBitSet extends ISet[Int] ..
```

Accurate types, but code duplication.

```
trait ITraversable[A] {
  def foreach[U](f: A => U): Unit
  ..
}
class IStack[A](val cont: List[A]) extends ITraversable[A] {
  ..
  def filter(p:A=>Boolean):IStack[A] ..
  def partition(p:A=>Boolean):(IStack[A],IStack[A]) ..
  def dropWhile(p:A=>Boolean):IStack[A] ..
}
class ISet[A] extends ITraversable[A] {
  ..
  def filter(p:A=>Boolean):ISet[A] ..
  def partition(p:A=>Boolean):(ISet[A],ISet[A]) ..
  def dropWhile(p:A=>Boolean):ISet[A] ..
}
class IBitSet extends ISet[Int] {
  ..
  def filter(p:A=>Boolean):IBitSet ..
  def partition(p:A=>Boolean):(IBitSet,IBitSet) ..
  def dropWhile(p:A=>Boolean):IBitSet ..
}
```

A builder object

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}

val b1 = new Builder[Int, IStack[Int]] { ... }
b1 += 1; b1 += 2; b2 += 3
val stack: IStack[Int] = b1.result()

val b2 = new Builder[Int, ISet[Int]] { ... }
b2 += 4; b2 += 5; b2 += 6
val set: ISet[Int] = b2.result()
```

- A builder knows how to construct a collection.
- The `To` parameter remembers the type of the coll.

Use an abstract method for creating a builder

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}

trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
}
class ISet[A] extends ITraversable[A, ISet[A]]
```

Use an abstract method for creating a builder

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}

trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  def filter(p: A => Boolean): Repr = {
    val b = newBuilder
    foreach {x => if (p(x)) b += x }
    b.result()
  }
  ...
}
class ISet[A] extends ITraversable[A, ISet[A]]
```

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}
trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
  def filter(p: A => Boolean): Repr = {
    val b = newBuilder
    foreach {x => if (p(x)) b += x }
    b.result()
  }
  def partition(p: A => Boolean): (Repr, Repr) = ...



  def dropWhile(p: A => Boolean): Repr = ...



}
```

Implement `partition` and `dropWhile`.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}
trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
  def filter(p: A => Boolean): Repr = {
    val b = newBuilder
    foreach {x => if (p(x)) b += x }
    b.result()
  }
  def partition(p: A => Boolean): (Repr, Repr) = {
    val l = newBuilder; val r = newBuilder
    foreach {x => if (p(x)) { l += x } else { r += x }}
    (l.result(), r.result())
  }
  def dropWhile(p: A => Boolean): Repr = ...



}
```

Implement `partition` and `dropWhile`.

Implement `partition` and `dropWhile`.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}
trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
  def filter(p: A => Boolean): Repr = {
    val b = newBuilder
    foreach {x => if (p(x)) b += x }
    b.result()
  }
  def partition(p: A => Boolean): (Repr, Repr) = {
    val l = newBuilder; val r = newBuilder
    foreach {x => if (p(x)) { l += x } else { r += x }}
    (l.result(), r.result())
  }
  def dropWhile(p: A => Boolean): Repr = {
    val b = newBuilder; var keep = false
    foreach {x => if (!keep && !p(x)) keep=true;
                  if (keep) b += x }
    b.result()
  }
}
```

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}
trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
}
class IStack[A](val cont: List[A])
        extends ITraversable[A,IStack[A]] {
  ...
  def newBuilder:Builder[A,IStack[A]] = new Builder[A,IStack[A]] {



  }
}
```

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}
trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
}
class IStack[A](val cont: List[A])
        extends ITraversable[A,IStack[A]] {
  ...
  def newBuilder:Builder[A,IStack[A]] = new Builder[A,IStack[A]] {

    def +=(elem: A): Builder[A,IStack[A]] =
    def result(): IStack[A] =
  }
}
```

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}
trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
}
class IStack[A](val cont: List[A])
        extends ITraversable[A,IStack[A]] {
  ...
  def newBuilder:Builder[A,IStack[A]] = new Builder[A,IStack[A]] {
    var l: List[A] = Nil
    def +=(elem: A): Builder[A,IStack[A]] = { l ::= elem; this }
    def result(): IStack[A] = new IStack(l.reverse)
  }
}
```

Implement newBuilder.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}
trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
}
class ISet[A] extends ITraversable[A, ISet[A]] {
  var cont: List[A] = Nil
  ...
  def newBuilder: Builder[A,ISet[A]] = ...
}
class IBitSet extends ISet[Int] with ITraversable[Int, IBitSet] {
  val bcont: Array[Boolean] = new Array(100)
  ...
  override def newBuilder: Builder[Int,IBitSet] = ...
}
```

Implement newBuilder.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}
trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
}
class ISet[A] extends ITraversable[A, ISet[A]] {
  var cont: List[A] = Nil
  ...
  def newBuilder: Builder[A,ISet[A]] = new Builder[A, ISet[A]] {
    var s: ISet[A] = new ISet
    def +=(elem: A): Builder[A, ISet[A]] = { s.add(elem); this }
    def result(): ISet[A] = { s.cont = s.cont.reverse; s }
  }
}
class IBitSet extends ISet[Int] with ITraversable[Int, IBitSet] {
  val bcont: Array[Boolean] = new Array(100)
  ...
  override def newBuilder: Builder[Int,IBitSet] = ...
}
```

Implement newBuilder.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}
trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
}
class ISet[A] extends ITraversable[A, ISet[A]] {
  var cont: List[A] = Nil
  ...
  def newBuilder: Builder[A,ISet[A]] = { ... }
}
class IBitSet extends ISet[Int] with ITraversable[Int, IBitSet] {
  val bcont: Array[Boolean] = new Array(100)
  ...
  override def newBuilder: Builder[Int,IBitSet] =
    new Builder[Int, IBitSet] {
      var s: IBitSet = new IBitSet
      def +=(elem: Int):Builder[Int,IBitSet] = { s.add(elem);this }
      def result(): IBitSet = s
    }
}
```

Vertion 3.0

- Add the following method to Traversable:

```
map[B](f: A=>B): ColTy[B]
```

- Here `ColTy[B]` is the type of the receiver object.

Challenge 1

`map[B](f: A=>B): ColTy[B]`

- A builder knows how to construct `ColTy[A]` but not `ColTy[B]`.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A, To]
  def result(): To
}
trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]
  ...
  def map[B](f: A=>B): ???
}
```

Challenge 2

```
map[B](f: A=>B): ColTy[B]
```

- `ColTy` can changes depending on `B`.

```
val b1 = new IBitSet; b1 add 1; b1 add 2
val b2: IBitSet = b1 map (_ + 1)
val b3: ISet[List[Int]] = b1 map (List(_))
```

Use implicit parameter

```
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}

trait ITraversable[A, +Repr] {

  def map[B, That](f: A => B)(
    implicit bf: CanBuildFrom[Repr, B, That]): That = ...
  ...
}
```

- Implicit parameter bf that can create a builder.

Use implicit parameter

```
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}

trait ITraversable[A, +Repr] {
  def newBuilder: Builder[A, Repr]
  def map[B, That](f: A => B)(
    implicit bf: CanBuildFrom[Repr, B, That]): That = ...
  ...
}
```

- Implicit parameter bf that can create a builder.
- Similar to newBuilder, but more general.

Use implicit parameter

```
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}

trait ITraversable[A, +Repr] {
  def newBuilder: Builder[A, Repr]
  def map[B, That](f: A => B)(
    implicit bf: CanBuildFrom[Repr, B, That]): That = ...
  ...
}
...
val b1 = new IBitSet; b1 add 1; b1 add 2
val b2: IBitSet = b1.map(_ + 1)
val b3: ISet[List[Int]] = b1.map(List(_))
```

- Implicit parameter bf that can create a builder.
- Similar to newBuilder, but more general.

Implicit values:

```
def bf2[B]: CanBuildFrom[ISet[_],B,ISet[B]]
val bf3: CanBuildFrom[IBitSet,Int,IBitSet]
```

```
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}

trait ITraversable[A, +Repr] {
  def newBuilder: Builder[A, Repr]
  def map[B, That](f: A => B)(
    implicit bf: CanBuildFrom[Repr, B, That]): That = ...
  ...
}
...
val b1 = new IBitSet; b1 add 1; b1 add 2
val b2: IBitSet = b1.map(_ + 1)
val b3: ISet[List[Int]] = b1.map(List(_))
```

- Implicit parameter bf that can create a builder.
- Similar to newBuilder, but more general.

Implicit values:

```
def bf2[B]: CanBuildFrom[ISet[_],B,ISet[B]]
val bf3: CanBuildFrom[IBitSet,Int,IBitSet]
```

```
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}

trait ITraversable[A, +Repr] {
  def newBuilder: Builder[A, Repr]
  def map[B, That](f: A => B)(
    implicit bf: CanBuildFrom[Repr, B, That]): That = ...
  ...
}
...
val b1 = new IBitSet; b1 add 1; b1 add 2
val b2: IBitSet = b1.map(_ + 1)(bf3)
val b3: ISet[List[Int]] = b1.map(List(_))
```

- Implicit parameter bf that can create a builder.
- Similar to newBuilder, but more general.

Implicit values:

```
def bf2[B]: CanBuildFrom[ISet[_],B,ISet[B]]
val bf3: CanBuildFrom[IBitSet,Int,IBitSet]
```

```
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}

trait ITraversable[A, +Repr] {
  def newBuilder: Builder[A, Repr]
  def map[B, That](f: A => B)(
    implicit bf: CanBuildFrom[Repr, B, That]): That = ...
  ...
}
...
val b1 = new IBitSet; b1 add 1; b1 add 2
val b2: IBitSet = b1.map(_ + 1)(bf3)
val b3: ISet[List[Int]] = b1.map(List(_))(bf2[List[Int]])
```

- Implicit parameter bf that can create a builder.
- Similar to newBuilder, but more general.

[Q] Define map.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}

trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}

trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]

  def map[B, That](f: A => B)(
        implicit bf: CanBuildFrom[Repr, B, That]): That = ...



  ...
}
```

[Q] Define map.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}

trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}

trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]

  def map[B, That](f: A => B)(
        implicit bf: CanBuildFrom[Repr, B, That]): That = {
    val b = bf(this)
    foreach {x => b += f(x)}
    b.result()
  }
  ...
}
```

[Q] Define map.

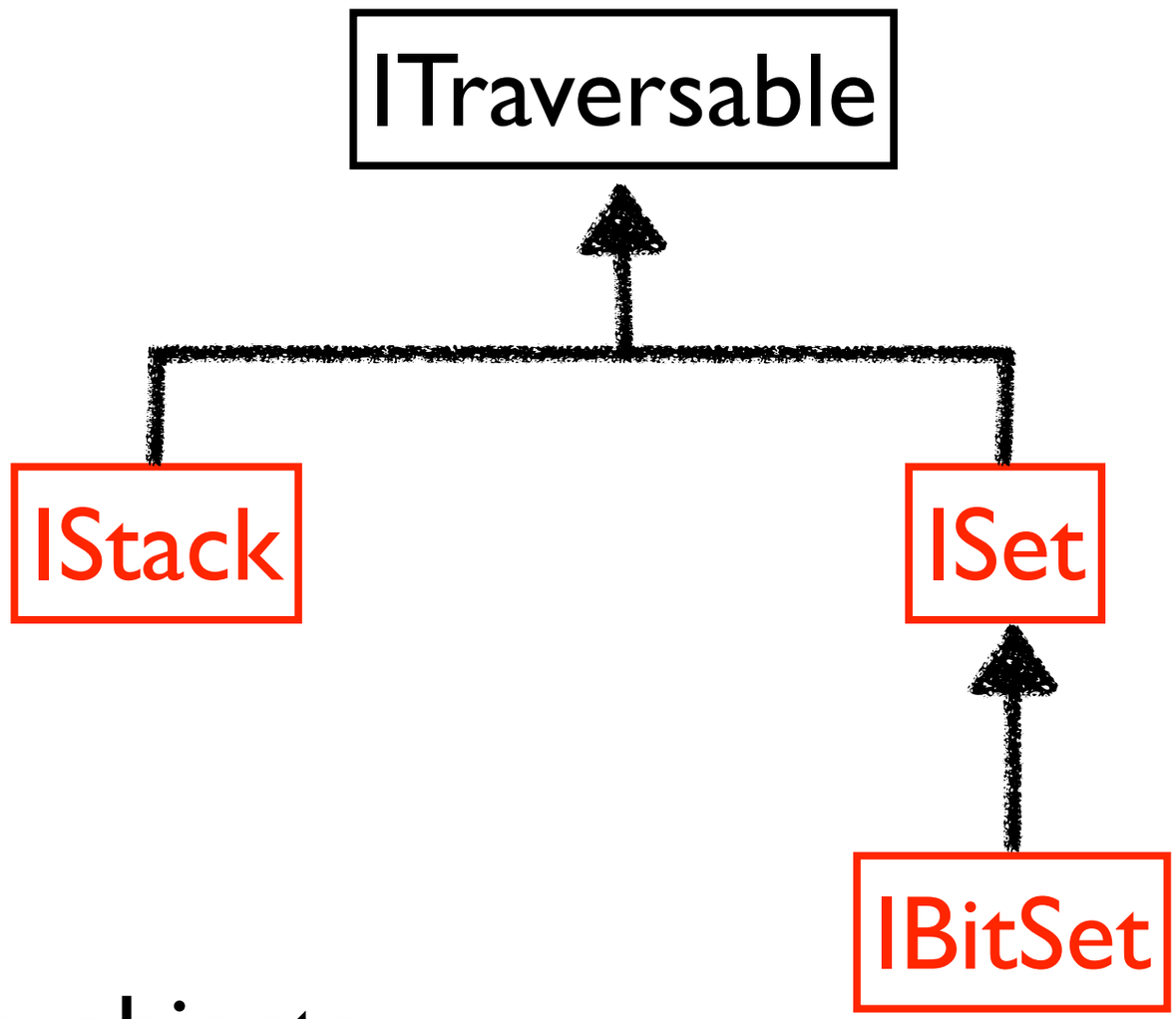
```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}

trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}

trait ITraversable[A, +Repr] {
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]

  def map[B, That](f: A => B)(
        implicit bf: CanBuildFrom[Repr, B, That]): That = {
    val b = bf(this)
    foreach {x => b += f(x)}
    b.result()
  }
  ...
}
```

Almost but doesn't comple.
The type of `this` is not `Repr`.

[Q] Define map.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}

trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}

trait ITraversable[A, +Repr] { this : Repr =>
  def foreach[U](f: A => U): Unit
  def newBuilder: Builder[A, Repr]

  def map[B, That](f: A => B)(
        implicit bf: CanBuildFrom[Repr, B, That]): That = {
    val b = bf(this)
    foreach {x => b += f(x)}
    b.result()
  }
  ...
}
```

Self type. It says that every object of `ITraversable[A,Repr]` has a subtype of `Repr`.



1. Create companion objects.
2. Add implicit values.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}
class IStack[A](val cont: List[A]) extends ...

object IStack {
  implicit def bf1[B] =




}
```

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}
class IStack[A](val cont: List[A]) extends ...

object IStack {
  implicit def bf1[B] = new CanBuildFrom[IStack[_],B,IStack[B]] {
  }
}
```

IStack[_] means IStack[X] for some X

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}
class IStack[A](val cont: List[A]) extends ...

object IStack {
  implicit def bf1[B] = new CanBuildFrom[IStack[_],B,IStack[B]] {
    def apply(from: IStack[_]): Builder[B, IStack[B]] =




    }
  }
}
```

IStack[_] means IStack[X] for some X

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}
class IStack[A](val cont: List[A]) extends ...

object IStack {
  implicit def bf1[B] = new CanBuildFrom[IStack[_],B,IStack[B]] {
    def apply(from: IStack[_]): Builder[B, IStack[B]] =
      new Builder[B, IStack[B]] {



      }
  }
}
```

IStack[_] means IStack[X] for some X

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}
class IStack[A](val cont: List[A]) extends ...

object IStack {
  implicit def bf1[B] = new CanBuildFrom[IStack[_],B,IStack[B]] {
    def apply(from: IStack[_]): Builder[B, IStack[B]] =
      new Builder[B, IStack[B]] {

        def +=(x: B): Builder[B,IStack[B]] =
        def result(): IStack[B] =
      }
  }
}
```

IStack[_] means IStack[X] for some X

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}
class IStack[A](val cont: List[A]) extends ...

object IStack {
  implicit def bf1[B] = new CanBuildFrom[IStack[_],B,IStack[B]] {
    def apply(from: IStack[_]): Builder[B, IStack[B]] =
      new Builder[B, IStack[B]] {
        var l: List[B] = Nil
        def +=(x: B): Builder[B,IStack[B]] = { l ::= x; this }
        def result(): IStack[B] = new IStack(l.reverse)
      }
  }
}
```

IStack[_] means IStack[X] for some X

[Q] Define bf2 and bf3.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}
object IStack {
  implicit def bf1[B] = new CanBuildFrom[IStack[_],B,IStack[B]] {
    def apply(from: IStack[_]): Builder[B, IStack[B]] =
      new Builder[B, IStack[B]] {
        var l: List[B] = Nil
        def +=(x: B): Builder[B,IStack[B]] = { l ::= x; this }
        def result(): IStack[B] = new IStack(l.reverse)
      }
  }
}

class ISet[A] extends ITraversable[A, ISet[A]]
{ var cont: List[A] = Nil; ... }
object ISet { implicit def bf2[B] = ... }

class IBitSet extends ISet[Int] with ITraversable[Int, IBitSet]
{ val bcont: Array[Boolean] = new Array(100); ... }
object IBitSet { implicit val bf3 = ... }
```

[Q] Define bf2 and bf3.

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}
```

```
object ISet {
  implicit def bf2[B] = new CanBuildFrom[ISet[_],B,ISet[B]] {
    def apply(from: ISet[_]): Builder[B, ISet[B]] =
      new Builder[B, ISet[B]] {
        var s: ISet[B] = new ISet
        def +=(x: B): Builder[B, ISet[B]] = { s.add(x); this }
        def result(): ISet[B] = { s.cont = s.cont.reverse; s }
      }
  }
}
```

```
class ISet[A] extends ITraversable[A, ISet[A]]
{ var cont: List[A] = Nil; ... }
object ISet { implicit def bf2[B] = ... }

class IBitSet extends ISet[Int] with ITraversable[Int, IBitSet]
{ val bcont: Array[Boolean] = new Array(100); ... }
object IBitSet { implicit val bf3 = ... }
```

```
trait Builder[-A, +To] {
  def +=(elem: A): Builder[A,To]
  def result(): To
}
trait CanBuildFrom[-This, -B, +That] {
  def apply(from: This): Builder[B, That]
}
```

[Q] Define bf2 and bf3.

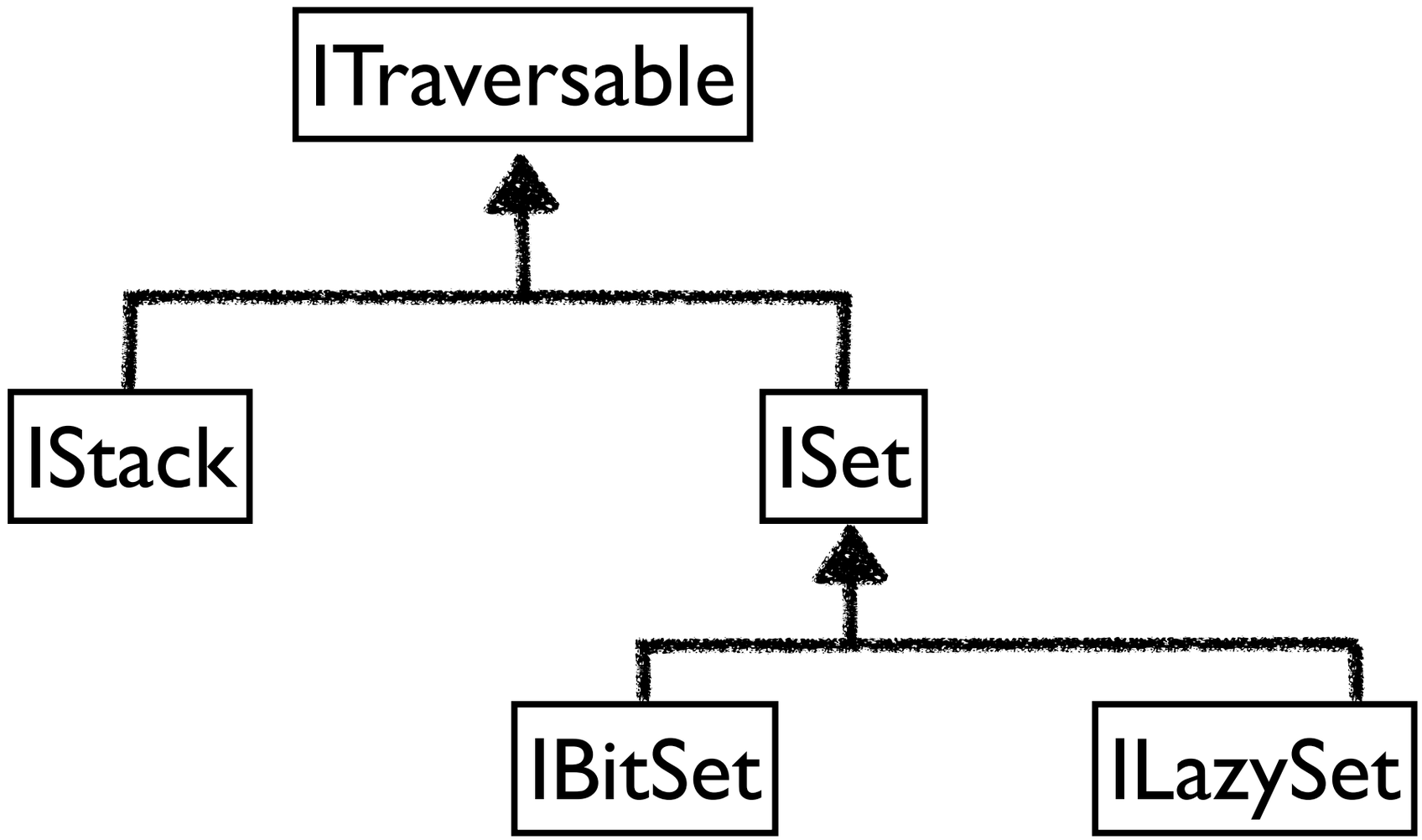
```
object ISet {
  implicit def bf2[B] = new CanBuildFrom[ISet[A], B, ISet[B]] {
    def apply(from: ISet[A]): Builder[B, ISet[B]] =
      new Builder[B, ISet[B]] {
        var s: ISet[B] = new ISet
        def +=(x:B): Builder[B,ISet[B]] = { s.add(x); this }
        def result(): ISet[B] = s
      }
  }
}
```

```
object IBitSet {
  implicit val bf3 = new CanBuildFrom[IBitSet, Int, IBitSet] {
    def apply(from: IBitSet): Builder[Int, IBitSet] =
      new Builder[Int, IBitSet] {
        var s: IBitSet = new IBitSet
        def +=(x:Int): Builder[Int,IBitSet] = { s.add(x); this }
        def result(): IBitSet = s
      }
  }
}
```

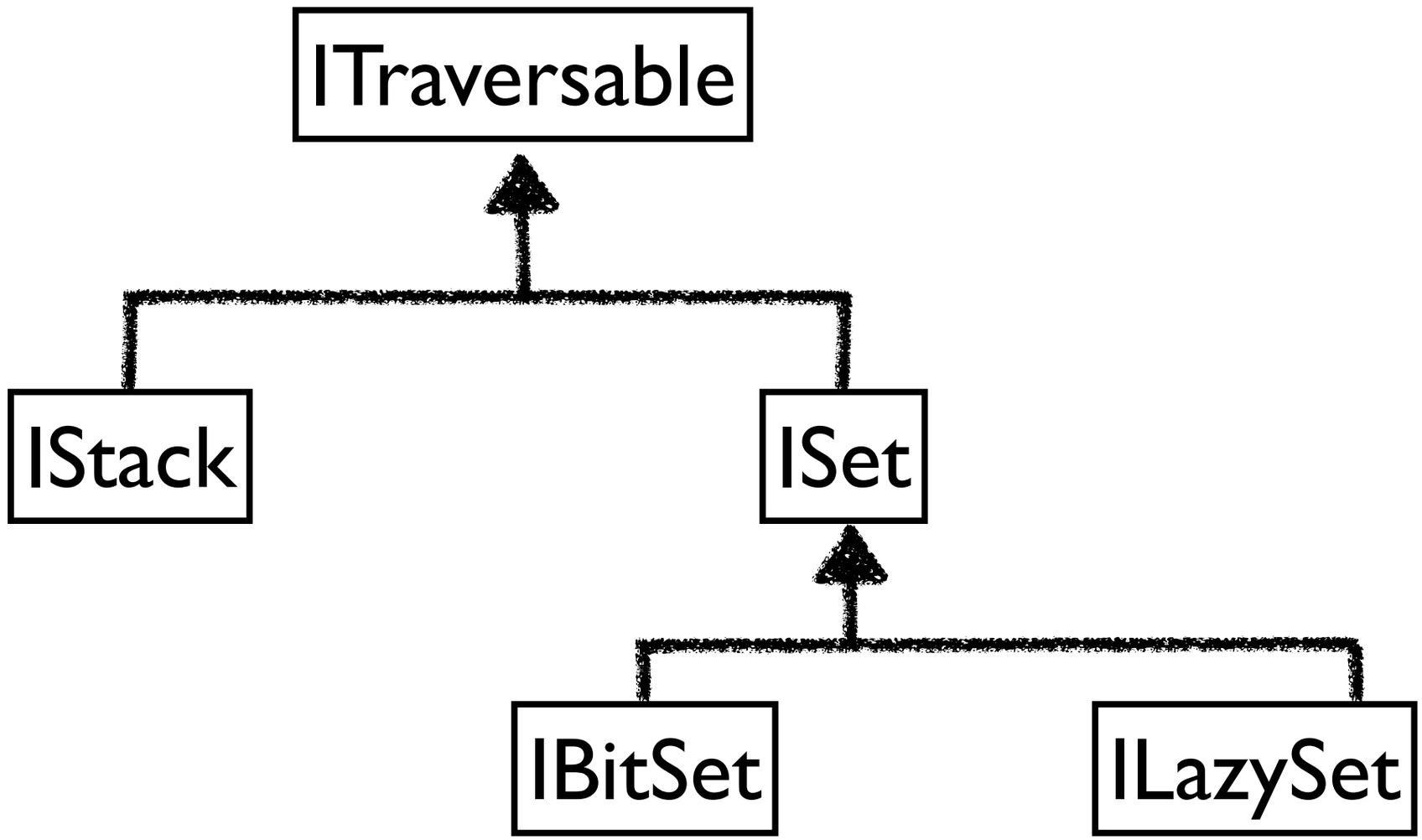
```
class ISet[A] extends ITraversable[A, ISet[A]]
{ var cont: List[A] = Nil; ... }
object ISet { implicit def bf2[B] = ... }

class IBitSet extends ISet[Int] with ITraversable[Int, IBitSet]
{ val bcont: Array[Boolean] = new Array(100); ... }
object IBitSet { implicit val bf3 = ... }
```

Our library doesn't handle dynamic type well.

Our library doesn't handle dynamic type well.

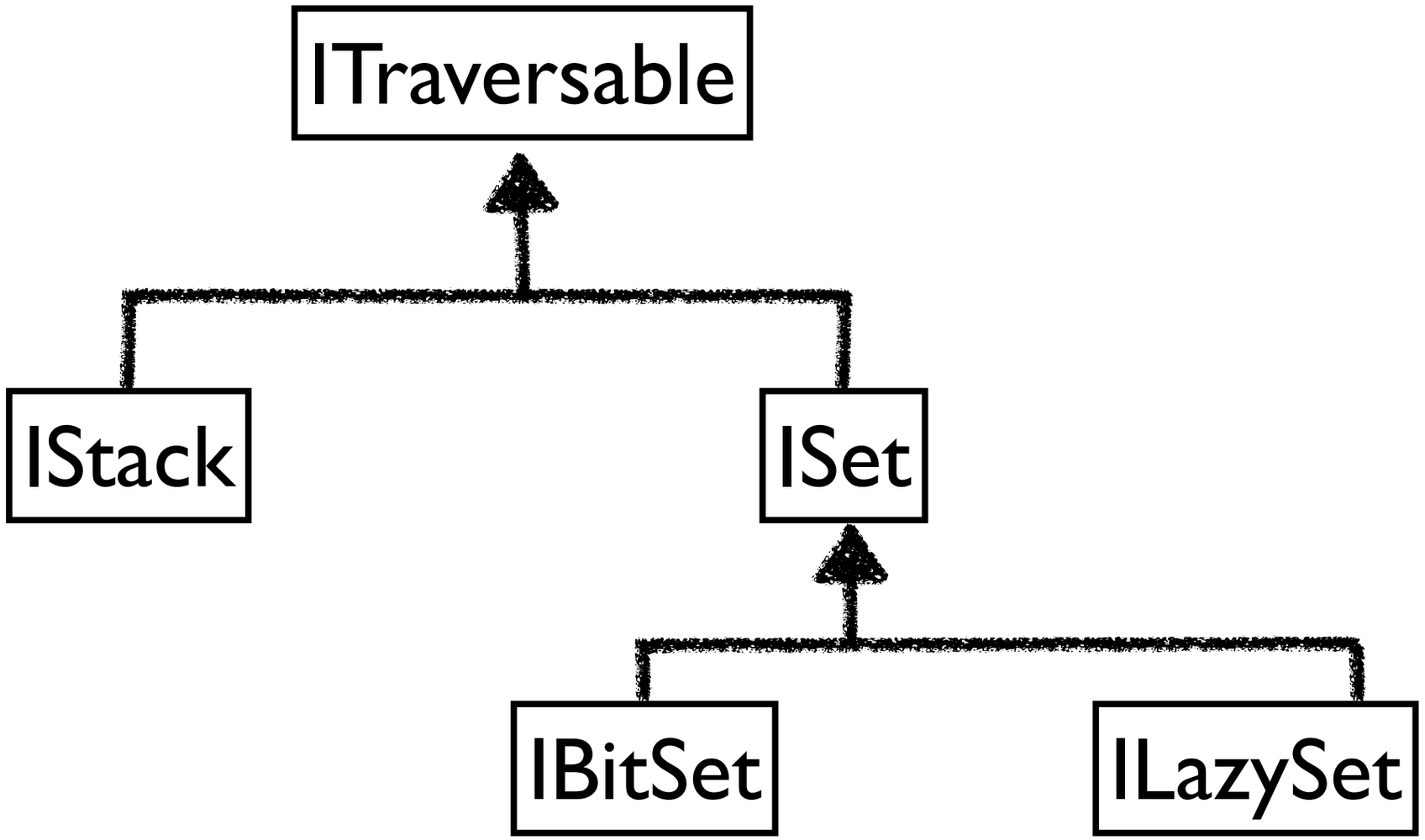


Our library doesn't handle dynamic type well.



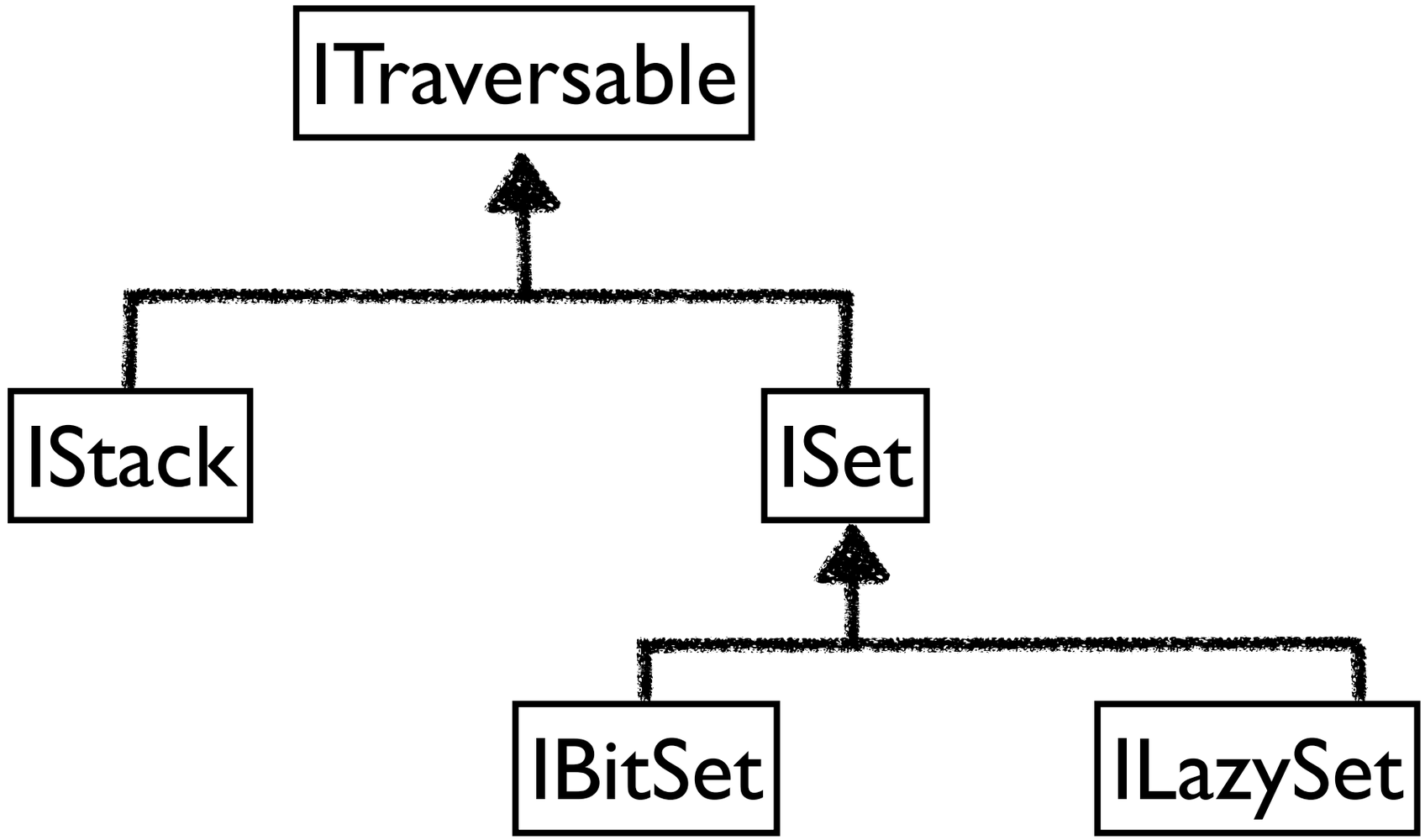

```
scala> val x: ISet[Int] = new ILazySet[Int]; x.add(1)
x: ISet[Int] = ...
```

Our library doesn't handle dynamic type well.



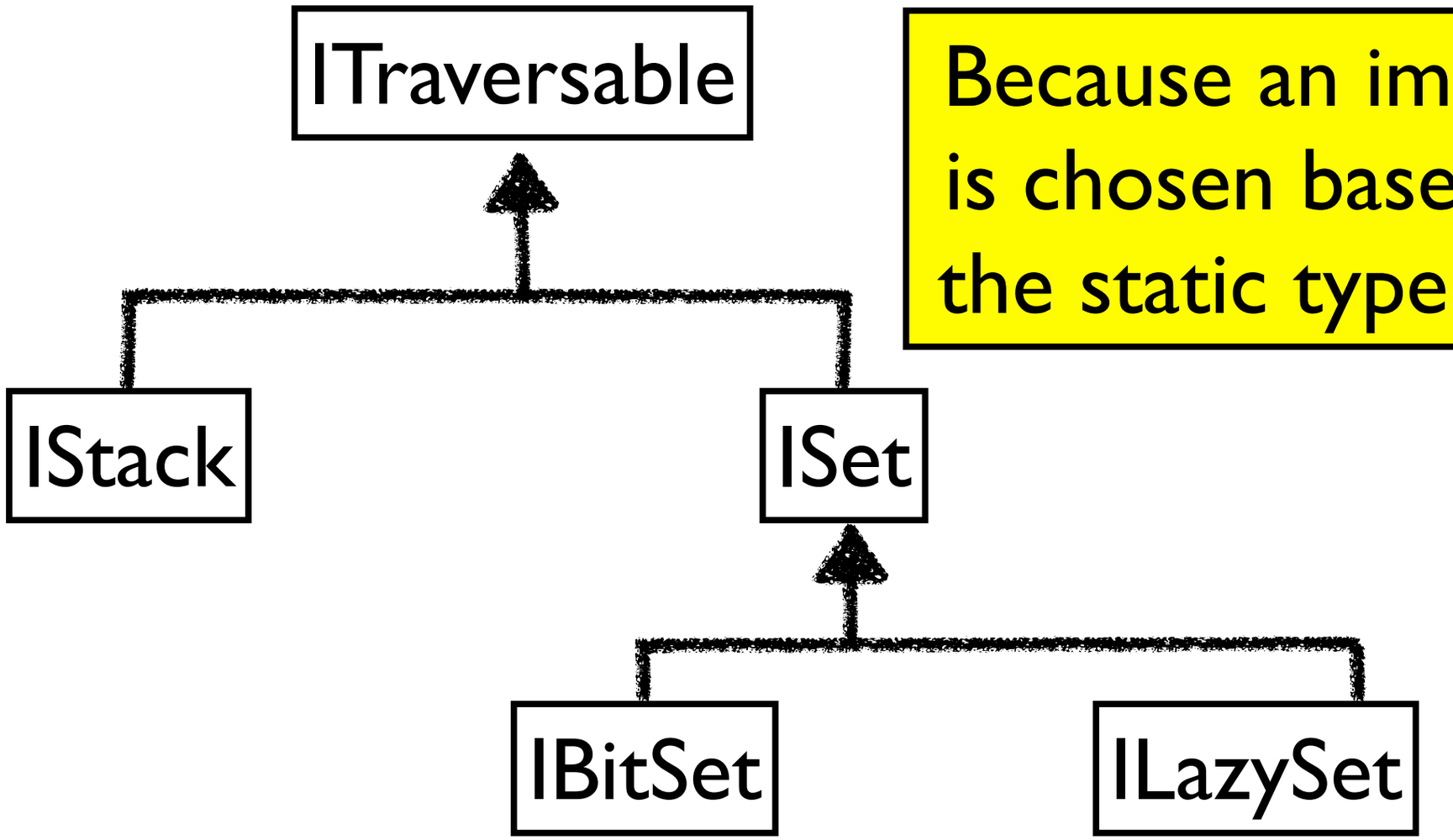

```
scala> val x: ISet[Int] = new ILazySet[Int]; x.add(1)
x: ISet[Int] = ...

scala> println(x.isInstanceOf[ILazySet[_]])
true
```

Our library doesn't handle dynamic type well.




```
scala> val x: ISet[Int] = new ILazySet[Int]; x.add(1)
x: ISet[Int] = ...

scala> println(x.isInstanceOf[ILazySet[_]])
true

scala> println((x map (_+1)).isInstanceOf[ILazySet[_]])
false
```

Our library doesn't handle dynamic type well.



Because an implicit is chosen based on the static type of x.

```
scala> val x: ISet[Int] = new ILazySet[Int]; x.add(1)
x: ISet[Int] = ...

scala> println(x.isInstanceOf[ILazySet[_]])
true

scala> println((x map (_+1)).isInstanceOf[ILazySet[_]])
false
```

```
scala> val x: ISet[Int] = new ILazySet[Int]; x.add(1)
x: ISet[Int] = ...

scala> println(x.isInstanceOf[ILazySet[_]])
true

scala> println((x map (_+1)).isInstanceOf[ILazySet[_]])
false
```

- Dynamic type of `x: ILazySet[Int]`
- Static type of `x: ISet[Int]`

```
scala> val x: ISet[Int] = new ILazySet[Int]; x.add(1)
x: ISet[Int] = ...

scala> println(x.isInstanceOf[ILazySet[_]])
true

scala> println((x map (_+1)).isInstanceOf[ILazySet[_]])
false
```

- Dynamic type of `x: ILazySet[Int]`
- Static type of `x: ISet[Int]`

```
trait Traversable[A, +Repr] {
  ...
  def map[B, That](f: A => B)(
       implicit bf: CanBuildFrom[Repr, B, That]
     ): That = {
    val b = bf(this)
    foreach {x => b += f(x)}
    b.result()
  }
}
```

```
scala> val x: ISet[Int] = new ILazySet[Int]; x.add(1)
x: ISet[Int] = ...

scala> println(x.isInstanceOf[ILazySet[_]])
true

scala> println((x map (_+1)).isInstanceOf[ILazySet[_]])
false
```

- Dynamic type of `x: ILazySet[Int]`
- Static type of `x: ISet[Int]`

```
trait Traversable[Int, ISet[Int]] {
  ...
  def map[Int, That](f: Int => Int)(
       implicit bf: CanBuildFrom[ISet[Int], Int, That]
      ): That = {
    val b = bf(this)
    foreach {x => b += f(x)}
    b.result()
  }
}
```

Implicit values:
bf2[B]: CanBuildFrom[ISet[_],B,ISet[B]]
bf4[B]: CanBuildFrom[ILazySet[_],B,ILazySet[B]]

```
true

scala> println((x map (_+1)).isInstanceOf[ILazySet[_]])
false
```

- Dynamic type of x: `ILazySet[Int]`
- Static type of x: `ISet[Int]`

```
trait Traversable[Int, ISet[Int]] {
  ...
  def map[Int, That](f: Int => Int)(
       implicit bf: CanBuildFrom[ISet[Int], Int, That]
     ): That = {
    val b = bf(this)
    foreach {x => b += f(x)}
    b.result()
  }
}
```

Version 4.0

```
scala> val x: ISet[Int] = new ILazySet[Int]; x.add(1)
x: ISet[Int] = ...

scala> println(x.isInstanceOf[ILazySet[_]])
true

scala> println((x map (_+1)).isInstanceOf[ILazySet[_]])
false
true
```

- Correct dynamic type.
- It means that the library should use a correct builder.

Idea: Double dispatch

```
object ISet {
  implicit def bf2[B] = new CanBuildFrom[ISet[_],B,ISet[B]] {
    def apply(from: ISet[_]) = new Builder[B, ISet[B]] {
      var s: ISet[B] = new ISet
      def +=(x: B) = { s.add(x); this }
      def result() = { s.cont = s.cont.reverse; s }
    }
  }
}
```

Idea: Double dispatch

```
object ISet {
  implicit def bf2[B] = new CanBuildFrom[ISet[_], B, ISet[B]] {
    def apply(from: ISet[_]) = from.genBuilder[B]
  }
}
```

Idea: Double dispatch

```
object ISet {
  implicit def bf2[B] = new CanBuildFrom[ISet[_], B, ISet[B]] {
    def apply(from: ISet[_]) = from.genBuilder[B]
  }
}

class ISet[A] extends ITraversable[A, ISet[A]] {
  ...
  def genBuilder[B]: Builder[B,ISet[B]] =
    new Builder[B,ISet[B]] { ... }
}
```

Idea: Double dispatch

```
object ISet {
  implicit def bf2[B] = new CanBuildFrom[ISet[_], B, ISet[B]] {
    def apply(from: ISet[_]) = from.genBuilder[B]
  }
}

class ISet[A] extends ITraversable[A, ISet[A]] {
  ...
  def genBuilder[B]: Builder[B,ISet[B]] =
    new Builder[B,ISet[B]] { ... }
}

class ILazySet[A] extends ITraversable[A, ILazySet[A]] {
  ...
  def genBuilder[B]: Builder[B,ILazySet[B]] =
    new Builder[B,ILazySet[B]] { ... }
}
```

```
def map...(f...)(implicit bf ...) ... {
  val b = bf(this)
  foreach {x => b += f(x)}
  b.result()
}
```

- bf is bf2
- dyn. type of this: ILazySet[Int]

```
object ISet {
  implicit def bf2[B] = new CanBuildFrom[ISet[_], B, ISet[B]] {
    def apply(from: ISet[_]) = from.genBuilder[B]
  }
}

class ISet[A] extends ITraversable[A, ISet[A]] {
  ...
  def genBuilder[B]: Builder[B,ISet[B]] =
    new Builder[B,ISet[B]] { ... }
}

class ILazySet[A] extends ITraversable[A, ILazySet[A]] {
  ...
  def genBuilder[B]: Builder[B,ILazySet[B]] =
    new Builder[B,ILazySet[B]] { ... }
}
```

```
def map...(f...)(implicit bf ...) ... {
  val b = bf(this)
  foreach {x => b += f(x)}
  b.result()
}
```

- bf is bf2
- dyn. type of this: ILazySet[Int]

```
object ISet {
  implicit def bf2[B] = new CanBuildFrom[ISet[_], B, ISet[B]] {
    def apply(from: ISet[_]) = from.genBuilder[B]
  }
}

class ISet[A] extends ITraversable[A, ISet[A]] {
  ...
  def genBuilder[B]: Builder[B,ISet[B]] =
    new Builder[B,ISet[B]] { ... }
}

class ILazySet[A] extends ITraversable[A, ILazySet[A]] {
  ...
  def genBuilder[B]: Builder[B,ILazySet[B]] =
    new Builder[B,ILazySet[B]] { ... }
}
```

```
object ISet {
  implicit def bf2[B] = new CanBuildFrom[ISet[_], B, ISet[B]] {
    def apply(from: ISet[_]) = from.genBuilder[B]
  }
}

class ISet[A] extends ITraversable[A, ISet[A]] {
  ...
  def genBuilder[B]: Builder[B,ISet[B]] =
    new Builder[B,ISet[B]] { ... }
}

object ILazySet {
  implicit def bf3[B] =
    new CanBuildFrom[ILazySet[_],B,ILazySet[B]] {
      def apply(from: ILazySet[_]) = from.genBuilder[B]
    }
}

class ILazySet[A] extends ITraversable[A, ILazySet[A]] {
  ...
  def genBuilder[B]: Builder[B,ILazySet[B]] =
    new Builder[B,ILazySet[B]] { ... }
}
```

Further information

- Source code of this collection library is in the course web page.
  - coll1.scala : without dynamic dispatch
  - coll2.scala : with dynamic dispatch
- Try to navigate the source code of Scala collections.

Scala Standard Library API (Scaladoc) 2.10.1 - Iterabl...API (Scaladoc) 2.10.1 - scala.collection.IterableLike

www.scala-lang.org/api/current/index.html#scala.collection.IterableLike

Reader

Apple iCloud Facebook Twitter Wikipedia Yahoo! News Popular

#ABCDEFGHIJKLMNOP

display packages only

scala hide focus

- Any
- AnyRef
- AnyVal
- App
- Application
- Array
- Boolean
- Byte
- Char
- Cloneable
- Console
- DelayedInit
- deprecated
- deprecatedName
- Double
- Dynamic
- Enumeration
- Equals

scala.collection

IterableLike

```
trait IterableLike[+A, +Repr] extends Equals with
TraversableLike[A, Repr] with GenIterableLike[A, Repr]
```

A template trait for iterable collections of type `Iterable[A]`.

This is a base trait for all Scala collections that define an `iterator` method to step through one-by-one the collection's elements. Implementations of this trait need to provide a concrete method with signature:

```
def iterator: Iterator[A]
```

They also need to provide a method `newBuilder` which creates a builder for collections of the same kind.

This trait implements `Iterable`'s `foreach` method by stepping through all elements using `iterator`. Subclasses should re-implement `foreach` with something more efficient, if possible.

This trait adds methods `iterator`, `sameElements`, `takeRight`, `dropRight` to the methods inherited from trait `Traversable`.

Note: This trait replaces every method that uses `break` in `TraversableLike` by an iterator version.

Self Type	IterableLike[A, Repr]
Source	IterableLike.scala

Click here

```
39   *    Note: This trait replaces every method that uses `break` in
40   *    `TraversableLike` by an iterator version.
41   *
42   *  @author Martin Odersky
43   *  @version 2.8
44   *  @since   2.8
45   *  @tparam A    the element type of the collection
46   *  @tparam Repr the type of the actual collection containing the elements.
47   *
48   *  @define Coll Iterable
49   *  @define coll iterable collection
50   */
51 trait IterableLike[+A, +Repr] extends Any with Equals with TraversableLike[A, Repr] with GenIterable
52 self =>
53
54   override protected[this] def thisCollection: Iterable[A] = this.asInstanceOf[Iterable[A]]
55   override protected[this] def toCollection(repr: Repr): Iterable[A] = repr.asInstanceOf[Iterable[A]]
56
57   /** Creates a new iterator over all elements contained in this iterable object.
58    *
59    *  @return the new iterator
60    */
61   def iterator: Iterator[A]
62
```

Summary

- Overall structure and the design of Scala's collection library.
- Minimal code duplication while keeping maximum info about types.
- Read Chapter 25. A good paper to read:
  - "Fighting bit rot with types" by Odersky & Moors.
  - http://lampwww.epfl.ch/~odersky/papers/fsttcs09.html