コレクション詳説

1. 参考文献

- Scala スケーラブルプログラミング 第 4 版
  - 2021 年
  - https://book.impress.co.jp/books/1119101190
  - 参考書として使用

1. 参考文献

- https://docs.scala-lang.org/ja/overviews/collections/introduction.html
  - 2025-01-05 参照
  - コレクションに関しての日本語の文献
  - Scala スケーラブルプログラミングと構成や文がほとんど同じだが、こちらの方が古い
- https://docs.scala-lang.org/overviews/collections-2.13/introduction.html
  - 2025-01-05 参照
  - コレクションに関する文献
  - Scala スケーラブルプログラミングはこれを訳したものと考えてよい
  - Scala3 のコードが掲載されているが、部分的に関数スタイルのコードが撤廃されている

2. コードとスライド

一部のコレクションの動作確認を行ったコード、本スライドは下記のページに公開している。

- https://github.com/blank71/scala-learn

Outline

3. コレクション詳細

Scala のコレクションの特徴

- 使いやすさ
- 簡潔性
- 安全性
- 高速
- 統一性

4. 安全性

Scala のコレクションは静的に型づけされ関数型の性質を持つため、問題をコンパイル時に静的に検出することができる。

1. コレクション演算自体は多用されているため、よくテストされている
2. コレクション演算を使うと、関数のパラメーターと結果値という形で入出力が明確になる
3. これら明示的な入出力は静的型チェックの対象となる

5. 高速

- コレクション演算は、調整、最適化されてライブラリーにまとめられている
  - そのため、コレクションは効率的に使えることが多い
- マルチコアでの並列実行に対応している
  - 並列コレクションは逐次コレクションと同じ演算をサポートしている
  - 新しい演算を覚えたり、コードを書き換える必要はない
  - `par` メソッドで簡単に並列コレクションに変換できる。

5. 高速


```
val a = (1 to 1000000).toArray
a.fold(0)((x, y) ⇒ x + y)
```

```
val a = (1 to 1000000).toArray.par
a.fold(0)((x, y) ⇒ x + y)
```

```
$ scala-cli run . --server=false --main-class Par
Time: 131 ms
Time: 76 ms
```

6. ミュータブルなコレクションとイミュータブルなコレクション

- ミュータブルな(可変)コレクション
  - 上書きする形で更新
  - 副作用という形でコレクションの要素を追加、変更、削除
- イミュータブルな(不変)コレクション
  - 変更できない
  - 追加、変更、削除の際は新しいコレクションを返す

6. ミュータブルなコレクションとイミュータブルなコレクション

- scala.collection.immutable
  - 誰に対しても不変
- scala.collection.mutable
  - コレクションを直接書き換える事が可能
  - コードベースのどこで変更されうるのか把握しておく必要がある
- scala.collection
  - immutable と mutabale のどちらも可能
  - scala.collection.IndexedSeq[T] は下記のスーパートレイト
    - scala.collection.immutable.IndexedSeq[T]
    - scala.collection.mutable.IndexedSeq[T]

6. ミュータブルなコレクションとイミュータブルなコレクション

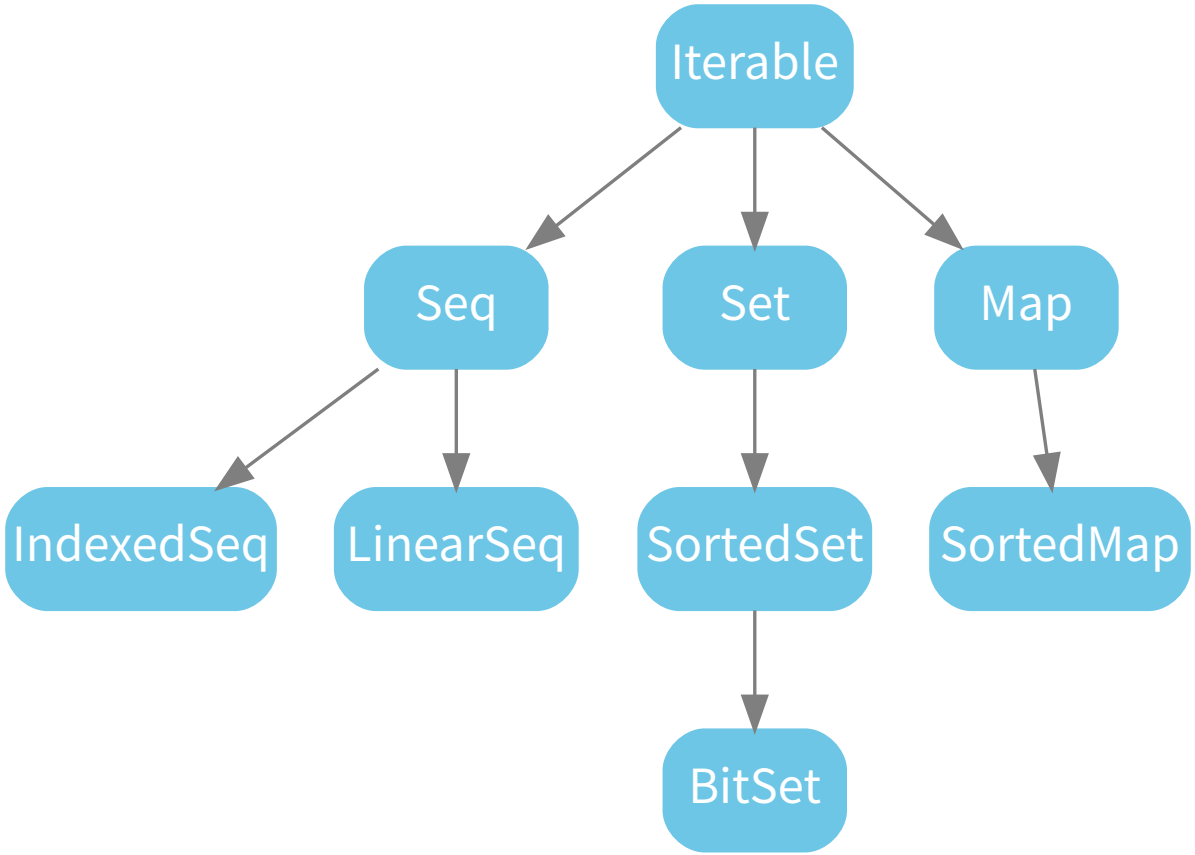
`scala.collection` の基底 (root) コレクションは不変コレクションと同じインターフェイスを定義し、`scala.collection.mutable` の可変コレクションは副作用を伴う変更演算を可変インターフェイスに加える。

基底コレクションと不変コレクションの違いは、不変コレクションのクライアントは他の誰もコレクションを変更しないという保証があるのに対し、基底コレクションのクライアントは自分ではコレクションを変更しないという約束しかできない。

たとえ静的な型がコレクションを変更するような演算を提供していなくても、実行時の型は他のクライアントが手を加えることができる可変コレクションである可能性がある。

7. コレクションの一貫性

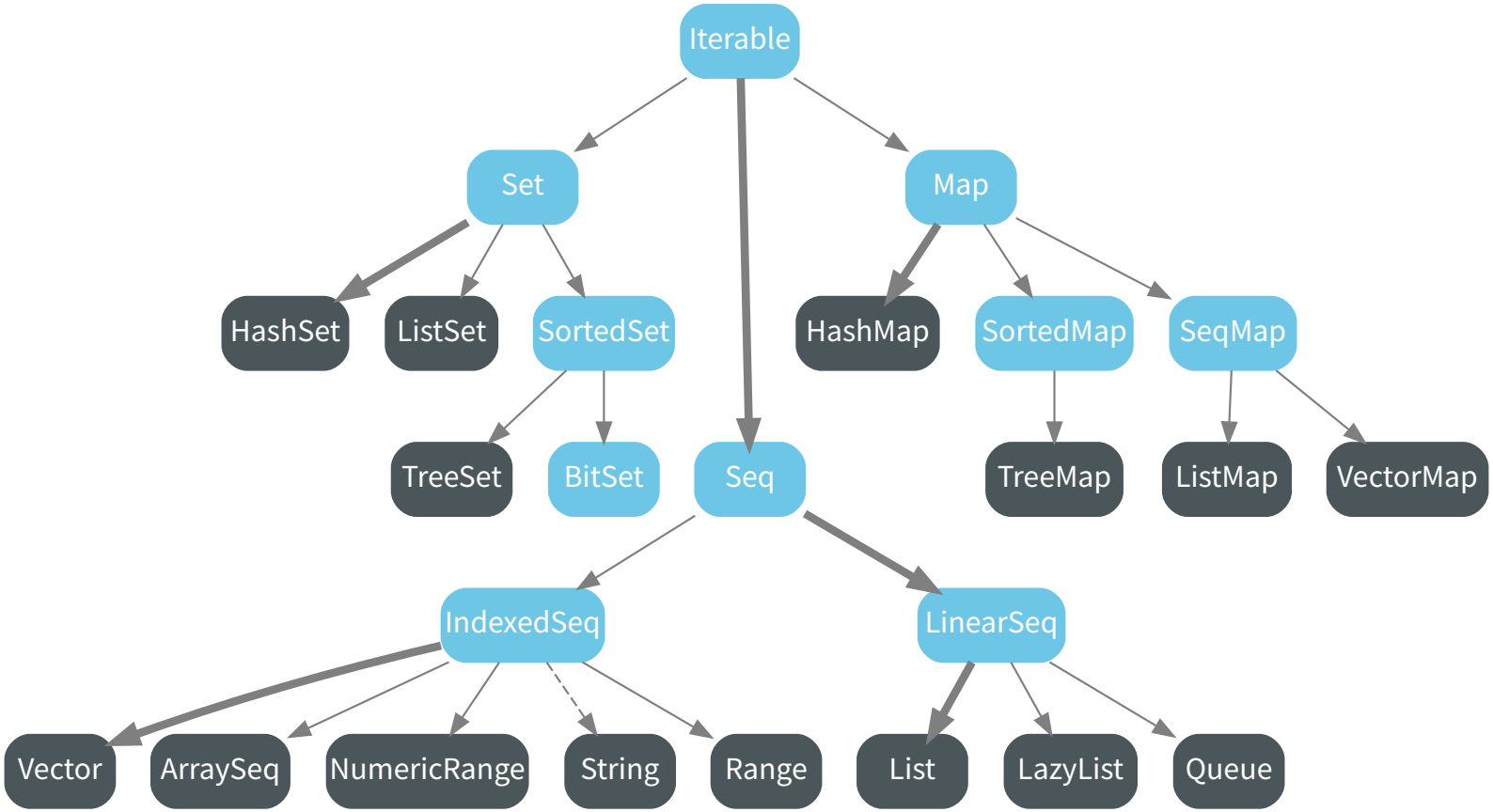
`scala.collection` の抽象クラスまたはトレイトの関係



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

7. コレクションの一貫性

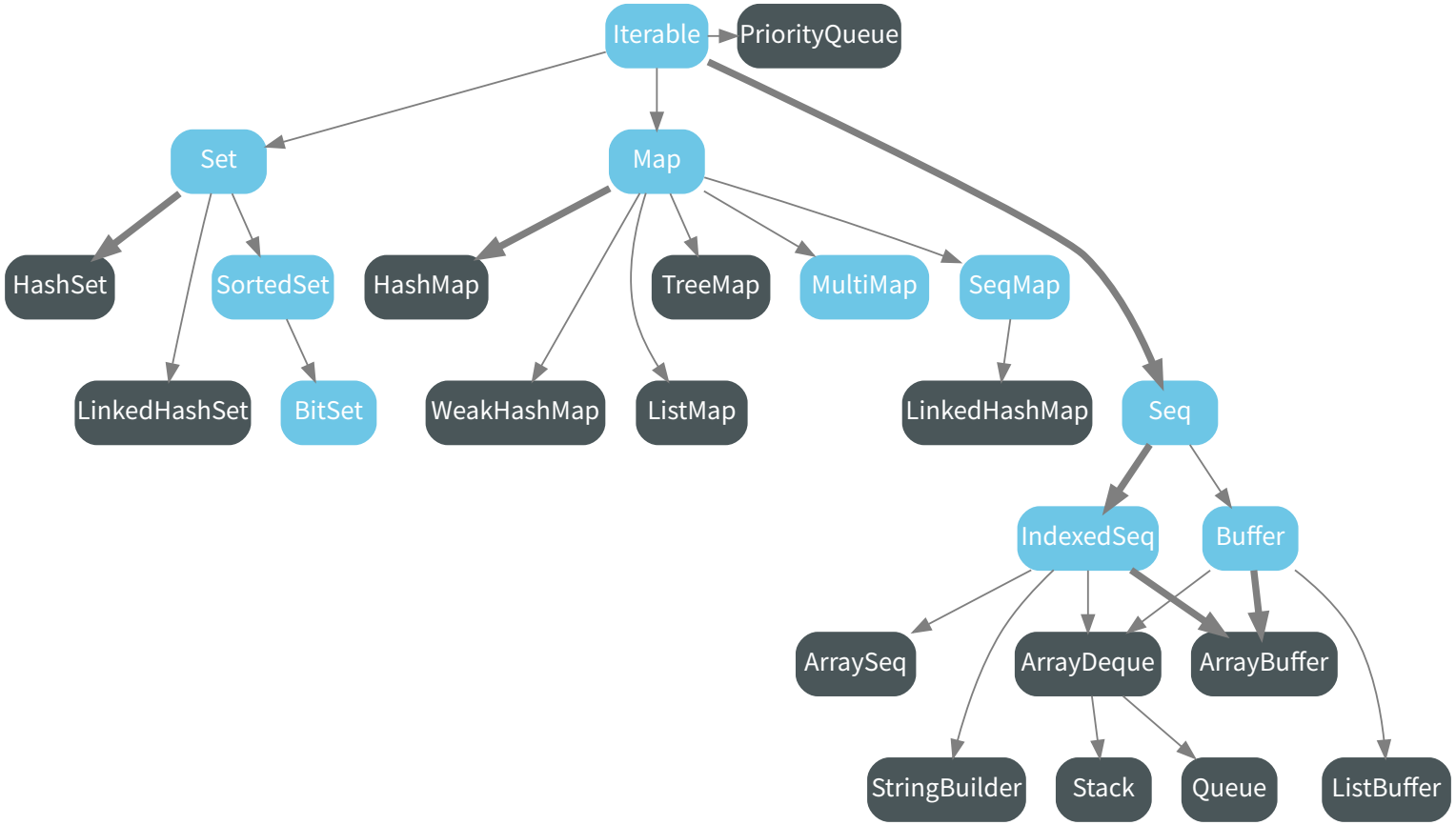
`scala.collection.immutable`



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

7. コレクションの一貫性

scala.collection.mutable



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

7. コレクションの一貫性

あらゆるコレクションはクラス名を書いたあとに要素を書くという統一した構文で作成することができる。

```
import scala.collection.immutable._
import scala.collection.mutable._
Iterable("x", "y", "z")
Map("x" → 24, "y" → 25, "z" → 26)
Set(1, 2, 3)
SortedSet("hello", "world")
Buffer(1, 2, 3)
IndexedSeq(1.0, 2.0)
LinearSeq(1, 2, 3)
// 特定のコレクションでも勿論使用できる
List(1, 2, 3)
HashMap("x" → 24, "y" → 25, "z" → 26)
```

7. コレクションの一貫性

- すべてのコレクションは Iterable が提供する API をサポートする
- それらのメソッドはどれも基底クラスの Iterable ではなく自分のクラスのインスタンスを返す
- List の map メソッドの戻り値の型は List
- Set の map メソッドの戻り値の型は Set

```
scala> List(1, 2, 3) map (x ⇒ x + 1)
val res0: List[Int] = List(2, 3, 4)

scala> Set(1, 2, 3) map (x ⇒ x + 1)
val res1: Set[Int] = Set(2, 3, 4)
```

8. Iterable トレイト

- コレクションの最上位は Iterable[A] トレイトである
- A はコレクションの要素型
- このトレイトのすべてのメソッドは抽象メソッド iterator を使って定義されている
- iterator はコレクションの要素を 1 つずつ返す

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

- Iterable を実装するコレクションクラスが実装しなければならないのは、この iterator メソッドだけ
- それ以外は Iterable から継承できる

8. Iterable トレイト

Iterable では多くの具象メソッドも定義されている。これらのメソッドはこれから説明するものに分類することができる。

8. Iterable トレイト

- Iteration(列挙): foreach, grouped, sliding
  - イテレーターが定義した順序でコレクションの要素を反復的に返す

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val git = xs grouped 3
val git: Iterator[List[Int]] = <iterator>

scala> git.next
val res0: List[Int] = List(1, 2, 3)

scala> git.next
val res1: List[Int] = List(4, 5)

scala> git.next
java.util.NoSuchElementException: next on empty iterator
  at scala.collection.Iterator$$anon$19.next(Iterator.scala:973)
  at scala.collection.Iterator$$anon$19.next(Iterator.scala:971)
  at scala.collection.Iterator$GroupedIterator.next(Iterator.scala:269)
```

```
  at scala.collection.Iterator$GroupedIterator.next(Iterator.scala:156)
  at scala.collection.Iterator$$anon$9.next(Iterator.scala:584)
  ... 32 elided

scala> val sit = xs sliding 3
val sit: Iterator[List[Int]] = <iterator>

scala> sit.next
val res0: List[Int] = List(1, 2, 3)

scala> sit.next
val res1: List[Int] = List(2, 3, 4)

scala> sit.next
val res2: List[Int] = List(3, 4, 5)

scala> xs foreach (x => println(s"squared: $x -> ${x * x}"))
squared: 1 -> 1
squared: 2 -> 4
squared: 3 -> 9
```

```
squared: 4 -> 16
squared: 5 -> 25
```

8. Iterable トレイト

- Addition(追加): concat, ++
  - コレクションの後ろにコレクション、またはイテレーションが生成 (yield) するすべての要素を追加

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val ys = xs.concat(List(4, 5, 6))
val ys: List[Int] = List(1, 2, 3, 4, 5, 4, 5, 6)

scala> val ys = xs concat List(4, 5, 6)
val ys: List[Int] = List(1, 2, 3, 4, 5, 4, 5, 6)

scala> val ys = xs ++ List(4, 5, 6)
val ys: List[Int] = List(1, 2, 3, 4, 5, 4, 5, 6)
```

8. Iterable トレイト

- Map(写像): map, flatMap, collect
  - コレクションの要素に何らかの関数を適用して新しいコレクションを作成する

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val xss = xs map (x ⇒ x * x)
val xss: List[Int] = List(1, 4, 9, 16, 25)

scala> val xss = xs flatMap (x ⇒ List(x * x))
val xss: List[Int] = List(1, 4, 9, 16, 25)

scala> val xss = xs map (x ⇒ List(x * x))
val xss: List[List[Int]] = List(List(1), List(4), List(9), List(16), List(25))

scala> val xss = xs collect ({ case x if (x * x > 9) ⇒ x })
val xss: List[Int] = List(4, 5)
```

8. Iterable トレイト

- Conversions(変換): to, toList, toVector, toMap, toSet, toIndexedSeq, toBuffer, toArray
  - 具体的な Iterable コレクションを返す
  - ミュータブルコレクションが与えられた場合であっても新しいミュータブルコレクションを返す
  - 要求された型がコレクションの型と一致したとき、新しいコレクションが生成される

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val xss = xs.toArray
val xss: Array[Int] = Array(1, 2, 3, 4, 5)
```

8. Iterable トレイト

- Copy(コピー): copyToArray
  - コレクションの要素を配列にコピーする

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> var xss = new Array[Int](6)
var xss: Array[Int] = Array(0, 0, 0, 0, 0, 0)

scala> xs.copyToArray(xss)
val res15: Int = 5

scala> xss
val res16: Array[Int] = Array(1, 2, 3, 4, 5, 0)

scala> var xss = new Array[Int](6)
var xss: Array[Int] = Array(0, 0, 0, 0, 0, 0)

scala> xs.copyToArray(xss, 2)
val res17: Int = 4
```

```
18
19 scala> xss
20 val res18: Array[Int] = Array(0, 0, 1, 2, 3, 4)
```

8. Iterable トレイト

- サイズ情報: isEmpty, nonEmpty, size, knownSize, sizeIs
  - List などではコレクションの要素数を数えるためにトラバース(要素を順に辿っていくこと)が必要になる
  - LazyList などでは要素数が無限になることがある

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val s = xs.size
val s: Int = 5
```

8. Iterable トレイト

- 要素取得: head, last, headOption, lastOption, find
  - コレクションの先頭、末尾の要素、または条件を満たす最初の要素を選び出す
  - すべてのコレクションに「先頭」または「末尾」の意味が定義されているとは限らない
  - HashSet は実行ごとに変化するハッシュキーに従って要素を格納するため「先頭」要素が実行ごとに異なる可能性がある
  - 代わりに LinkedHashSet という順序付きのバージョンが存在する

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val h = xs.head
val h: Int = 1

scala> val f = xs find (x ⇒ x > 3)
val f: Option[Int] = Some(4)

```

8. Iterable トレイト

```
scala> val f = xs find (x ⇒ x > 10)
val f: Option[Int] = None
```

8. Iterable トレイト

- 部分コレクション取得: tail, init, slice, take, drop, takeWhile, dropWhile, filter, filterNot, withFilter
  - インデックスの範囲や述語によって識別される部分コレクションを返す

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val t = xs.tail
val t: List[Int] = List(2, 3, 4, 5)

scala> val xss = xs take 2
val xss: List[Int] = List(1, 2)

scala> val xss = xs drop 2
val xss: List[Int] = List(3, 4, 5)

scala> val xss = xs filter (x ⇒ x > 3)
val xss: List[Int] = List(4, 5)
```

8. Iterable トレイト

- 部分コレクションへの分割: splitAt, span, partition, partitionMap, groupBy, groupMap, groupMapReduce
  - レシーバーのコレクションの要素を複数の部分コレクションに分割する

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val xa = xs span (x ⇒ x < 3)
val xa: (List[Int], List[Int]) = (List(1, 2),List(3, 4, 5))

scala> val xb = xs span (x ⇒ x > 3)
val xb: (List[Int], List[Int]) = (List(),List(1, 2, 3, 4, 5))

scala> val xc = xs partition (x ⇒ x > 3)
val xc: (List[Int], List[Int]) = (List(4, 5),List(1, 2, 3))
```

8. Iterable トレイト

- 要素テスト: exists, forall, count
  - 指定された述語でコレクションの要素をテストする

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val xa = xs forall (x ⇒ x > 0)
val xa: Boolean = true

scala> val xb = xs forall (x ⇒ x > 1)
val xb: Boolean = false

scala> val xc = xs exists (x ⇒ x > 3)
val xc: Boolean = true

scala> val xd = xs exists (x ⇒ x > 6)
val xd: Boolean = false

scala> val xe = xs count (x ⇒ x > 1)
val xe: Int = 4
```

8. Iterable トレイト

- 畳み込み: foldLeft, foldRight, reduceLeft, reduceRight
  - 連続する要素に二項演算を適用する
  - fold は foldLeft の糖衣構文

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val x =  xs.fold (0) ((x,y) ⇒ x + y)
val x: Int = 15
```

8. Iterable トレイト

- 特定の要素型を対象とする畳み込み: sum, product, min, max

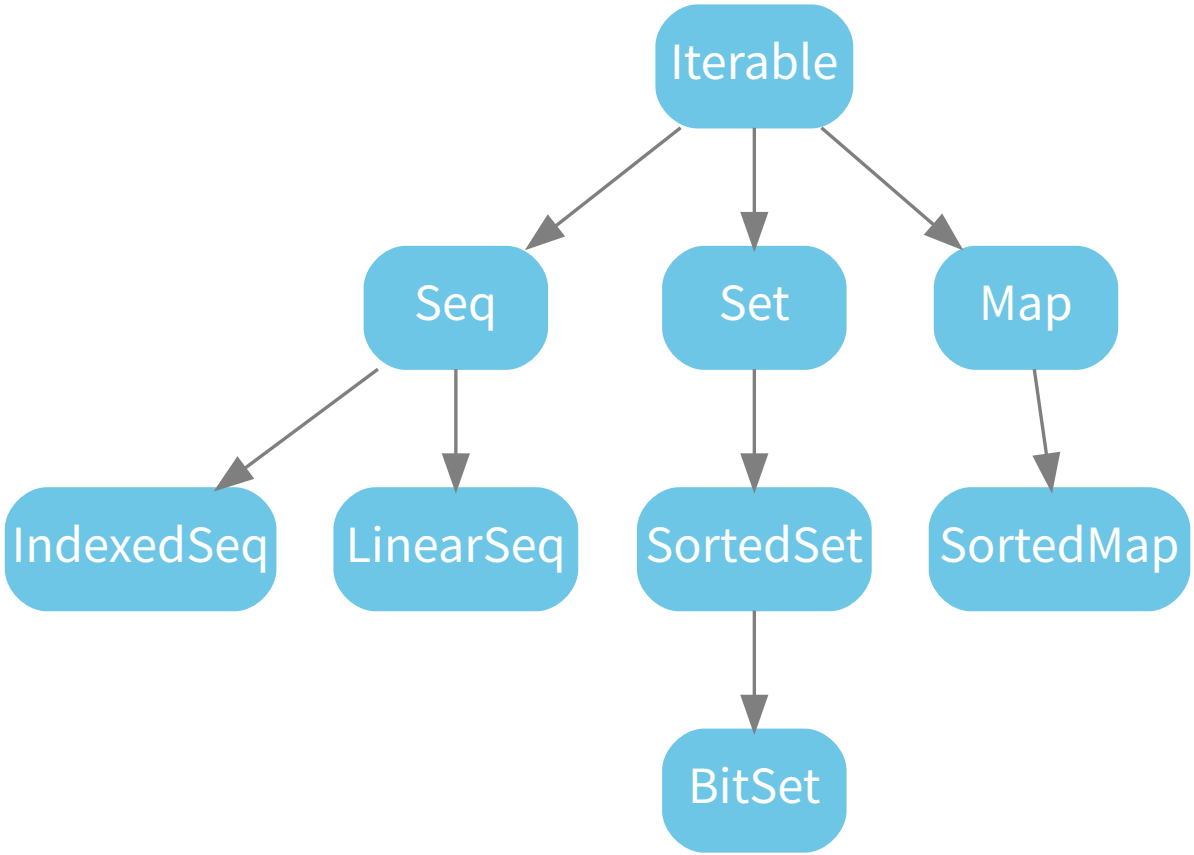
```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val xa = xs.product
val xa: Int = 120

scala> val xb = xs.fold (1) ((x, y) ⇒ x * y)
val xb: Int = 120
```

8. Iterable トレイト

- 文字列作成: mkString, addString
  - コレクションを文字列に変換する
- ビュー
  - 遅延評価されるコレクションのこと
  - 後で説明

9. Seq トレイト

Seq トレイトはシーケンスを表す。シーケンスは長さがあり、要素 0 から始まるインデックスーを持っているイテラブルである。



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

9. Seq トレイト

- インデックス参照、長さ: apply, isDefinedAt, length, indices, lengthCompare
  - Seq[T] 型のシーケンスは Int パラメーター(インデックス)をとり T 型の要素を返す部分関数
    - Seq[T] は PartialFunction[Int, T] を拡張している
    - シークエンスの要素には 0 から length - 1 までのインデックスがつけられる
    - シーケンスの length メソッドは コレクション全般の size メソッドの別名
    - lengthCompare メソッドを使うと無限シーケンスになっていても 2 つのシーケンスの長さを比較することができる
    - lengthIs メソッドは sizeIs メソッドの別名

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val xa = xs(2)
val xa: Int = 3

```

```
scala> val xb = xs apply 2
val xb: Int = 3

scala> val xc = xs.length
val xc: Int = 5

scala> val xd = xs lengthCompare List(1,2,3)
val xd: Int = 1

scala> val xe = xs lengthCompare List(1,2,3,4,5)
val xe: Int = 0

scala> val xf = xs lengthCompare List(1,2,3,4,5,6)
val xf: Int = -1

scala> val xg = xs.indices
val xg: Range = Range 0 until 5

scala> val xss = List(1, 3, 4)
val xss: List[Int] = List(1, 3, 4)

```

```
scala> val xh = xss.indices
val xh: Range = Range 0 until 3
```

9. Seq トレイト

- インデックスの検索: indexOf, lastIndexOf, indexOfSlice, lastIndexOfSlice, indexWhere, lastIndexWhere, segmentLength
  - 値または述語によって識別される要素のインデックスを返す

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val xa = xs indexOf 3
val xa: Int = 2

scala> val xb = xs indexOfSlice List(3, 4)
val xb: Int = 2

scala> val xc = xs indexWhere (x ⇒ x ⩾ 3)
val xc: Int = 2
```

9. Seq トレイト

- 追加: prepended(+:), prependedAll(++:), appended(:+), appendedAll(:++), padTo
  - シーケンスの先頭または末尾に要素を追加して得られる新しいシーケンスを返す

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val xa = -1 +: 0  +: xs
val xa: List[Int] = List(-1, 0, 1, 2, 3, 4, 5)

scala> val xb = List(-1, 0) ++: xs
val xb: List[Int] = List(-1, 0, 1, 2, 3, 4, 5)

scala> val xc = xs :+ 6 :+ 7
val xc: List[Int] = List(1, 2, 3, 4, 5, 6, 7)

scala> val xd = xs :++ List(6, 7)
val xd: List[Int] = List(1, 2, 3, 4, 5, 6, 7)

scala> val xe = xs.padTo(8, 2)
val xe: List[Int] = List(1, 2, 3, 4, 5, 2, 2, 2)
```

9. Seq トレイト

- 更新: updated, patch
  - 元のシーケンスに含まれる一部の要素を置換して得られる新しいシーケンスを返す

```
scala> val xs = List(1, 2, 3, 4, 5)
val xs: List[Int] = List(1, 2, 3, 4, 5)

scala> val xa = xs.patch(2, List(11, 12, 13), 0)
val xa: List[Int] = List(1, 2, 11, 12, 13, 3, 4, 5)

scala> val xb = xs.patch(2, List(11, 12, 13), 1)
val xb: List[Int] = List(1, 2, 11, 12, 13, 4, 5)

scala> val xc = xs.updated(2, 11)
val xc: List[Int] = List(1, 2, 11, 4, 5)
```

9. Seq トレイト

- ソート: sorted, sortWith, sortBy
  - さまざまな基準に基づいてシーケンスの要素を並べ替える

```
scala> val xa = List(-4, -2, -1, 2, 5, 11).sorted
val xa: List[Int] = List(-4, -2, -1, 2, 5, 11)

scala> val xb = List(-4, -2, -1, 2, 5, 11) sortWith ((x, y) =>
     |   if (x * x == y * y) then x < y else x * x < y * y
     | )
val xb: List[Int] = List(-1, -2, 2, -4, 5, 11)

scala> val xc = List(-4, 2, -1, -2, 5, 11) sortBy (x ⇒ x * x)
val xc: List[Int] = List(-1, 2, -2, -4, 5, 11)
```

9. Seq トレイト

- 反転: reverse, reverseIterator
  - シーケンスの要素を末尾から先頭に向かって逆順に並べたシーケンスか、そのような要素を生成するイテレーターを返す

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

9. Seq トレイト

- 比較: startsWith, endsWith, contains, containsSlice, corresponds, search
  - ２つのシーケンスの関係を調べたり、要素の有無を調べたりする

```
scala> val a = List(1, 2, 3) sameElements List(1, 2, 3)
val a: Boolean = true

scala> val b = List(1, 2, 3) sameElements List(1, 2, 4)
val b: Boolean = false

scala> val c = (0 to 10).toList startsWith (0 to 3).toList
val c: Boolean = true

scala> val d = (0 to 10).toList startsWith (0 to 11).toList
val d: Boolean = false
```

```
scala> import scala.collection.Searching._

scala> def found(r: scala.collection.Searching.SearchResult): (Int, Boolean) = {
     |   r match {
     |     case Found(index) ⇒
     |       println(s"index: $index")
     |       (index, true)
     |     case InsertionPoint(index) ⇒
     |       println(s"Not found")
     |       (index, false)
     |   }
     | }
def found(r: scala.collection.Searching.SearchResult): (Int, Boolean)

scala> val e = found ((0 to 10).toList search 3)
index: 3
val e: (Int, Boolean) = (3,true)

scala> val f = found ((0 to 10).toList search 11)
Not found
```

9. Seq トレイト

```
21 val f: (Int, Boolean) = (11,false)
```

```
1 scala> val g = (0 to 10).toList.corresponds ((0 to 10).toList) ((x,y) ⇒ x + y ⩾ 0)
2 val g: Boolean = true
3
4 scala> val h = (0 to 10).toList.corresponds ((0 to 11).toList) ((x,y) ⇒ x + y ⩾ 0)
5 val h: Boolean = false
6
7 scala> val i = (0 to 10).toList.corresponds ((0 to 10).toList) ((x,y) ⇒ x + y ⩾ 1)
8 val i: Boolean = false
```

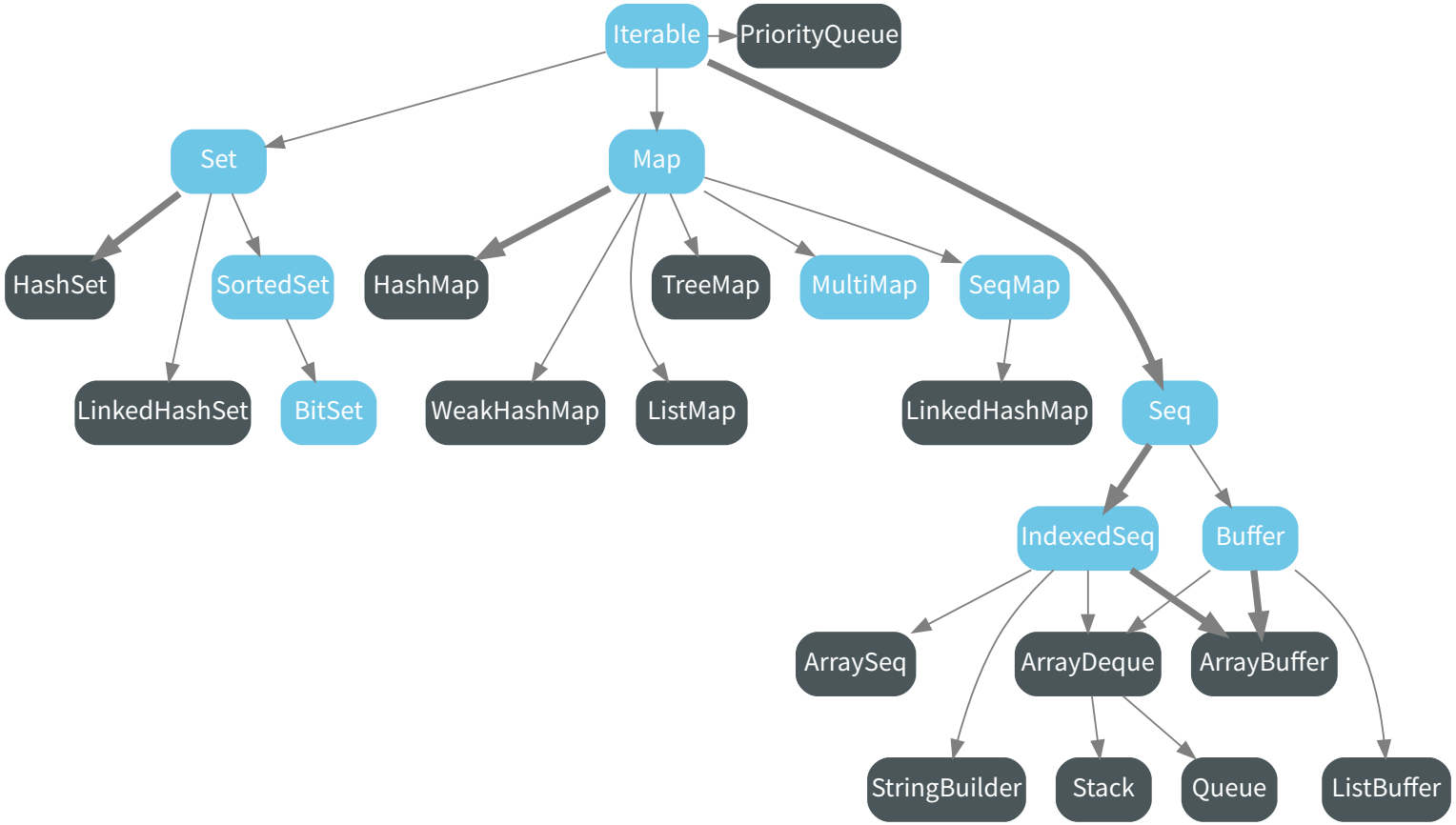
9. Seq トレイト

- 集合間演算: intersect, diff, distinct, distinctBy
  - ２つのシーケンスの要素に対して集合風の演算を実行したり重複を除いたりする

```
scala> val xa = (0 to 9).toList.intersect((2 to 5).toList)
val xa: List[Int] = List(2, 3, 4, 5)

scala> val xb = (0 to 9).toList.diff((2 to 5).toList)
val xb: List[Int] = List(0, 1, 6, 7, 8, 9)

scala> val xc = List(1, 1, 2, 2, 3).distinct
val xc: List[Int] = List(1, 2, 3)

scala> val xd = List(-1, 1, 2, -2, 3) distinctBy (x ⇒ x * x)
val xd: List[Int] = List(-1, 2, 3)
```

10. Buffer トレイト

ミュータブル Seq トレイトの１つ。



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

10. Buffer トレイト

Buffer は既存の要素の更新だけでなく、要素の挿入、削除、バッファーの末尾への効率的な新要素を追加をサポートする。Buffer の実装としてよく使われるのは ListBuffer と ArrayBuffer。

```
scala> import scala.collection.mutable._

scala> val xa = (1 to 3).toBuffer.append(9)
val xa: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 2, 3, 9)

scala> val xb = (1 to 3).toBuffer.append(9)
val xb: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 2, 3, 9)

scala> val xc = (1 to 3).toBuffer.appendAll((5 to 9).toList)
val xc: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 2, 3, 5, 6, 7, 8, 9)

scala> val xd = (1 to 3).toBuffer ++= ((5 to 9).toList)
val xd: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 2, 3, 5, 6, 7, 8, 9)

scala> val xe = (1 to 3).toBuffer.prepend(-4)
```

```
16 val xe: scala.collection.mutable.Buffer[Int] = ArrayBuffer(-4, 1, 2, 3)
17
18 scala> val xf = -4 +=: (1 to 3).toBuffer
19 val xf: scala.collection.mutable.Buffer[Int] = ArrayBuffer(-4, 1, 2, 3)
20
21 scala> val xg = (1 to 3).toBuffer.prependAll((-3 to -1).toBuffer)
22 val xg: scala.collection.mutable.Buffer[Int] = ArrayBuffer(-3, -2, -1, 1, 2, 3)
23
24 scala> val xh = (-3 to -1).toBuffer ++=: (1 to 3).toBuffer
25 val xh: scala.collection.mutable.Buffer[Int] = ArrayBuffer(-3, -2, -1, 1, 2, 3)
26
27 scala> val xi = (1 to 3).toBuffer
28 val xi: scala.collection.mutable.Buffer[Int] = ArrayBuffer(1, 2, 3)
29
30 scala> xi.insert(2, -4)
31
32 scala> println(xi)
33 ArrayBuffer(1, 2, -4, 3)
```

11. Set トレイト

Set は重複する要素を含まない Iterable。

- テスト: 集合が引数の要素を含んでいるかどうか

```
scala> val xs = (0 to 3).toSet
val xs: Set[Int] = Set(0, 1, 2, 3)

scala> println(xs(2))
true

scala> println(xs(4))
false
```

11. Set トレイト

イミュータブルとミュータブルで挙動が異なる。

イミュータブルなので新しい Set を返す。

```
scala> var s = collection.immutable.Set(1, 2, 3)
var s: Set[Int] = Set(1, 2, 3)

scala> println(s += 4)
()

scala> println(s -= 2)
()

scala> println(s)
Set(1, 3, 4)
```

イミュータブルとミュータブルで挙動が異なる。

ミュータブルなので既存の Set を変更している。

```
scala> val s = collection.mutable.Set(1, 2, 3)
val s: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3)

scala> println(s += 4)
HashSet(1, 2, 3, 4)

scala> println(s -= 2)
HashSet(1, 3, 4)

scala> println(s)
HashSet(1, 3, 4)
```

val に格納されたミュータブルコレクションは var に格納されたイミュータブルコレクションによって置き換え可能である。逆も可能。

12. Map トレイト

Map はキー/バリューのペア(写像(mapping)、対応関係(associations)などとも呼ばれる)である。

```
scala> val a = Map("a" → 1, "b" → 2, "c" → 3)
val a: Map[String, Int] = Map(a → 1, b → 2, c → 3)

scala> val b = Map(("a", 1), ("b", 2), ("c", 3))
val b: Map[String, Int] = Map(a → 1, b → 2, c → 3)
```

12. Map トレイト

- ルックアップ: apply, get, getOrElse, contains, isDefinedAt
  - キーからバリューを取り出すための部分関数

```
scala> val xs = Map("a" → 1, "b" → 2, "c" → 3)
val xs: Map[String, Int] = Map(a → 1, b → 2, c → 3)

scala> val a = xs get "b"
val a: Option[Int] = Some(2)

scala> val b = xs get "x"
val b: Option[Int] = None

scala> val c = xs apply "b"
val c: Int = 2

scala> val d = xs apply "x"
java.util.NoSuchElementException: key not found: x
  at scala.collection.immutable.Map$Map3.apply(Map.scala:417)
  ... 32 elided
```

12. Map トレイト

- 追加、更新: +, updated, ++, concat

```
scala> val xs = Map("a" → 1, "b" → 2, "c" → 3)
val xs: Map[String, Int] = Map(a → 1, b → 2, c → 3)

scala> val xa = xs + ("d" → 4)
val xa: Map[String, Int] = Map(a → 1, b → 2, c → 3, d → 4)

scala> val xb = xs + ("b" → 4)
val xb: Map[String, Int] = Map(a → 1, b → 4, c → 3)
```

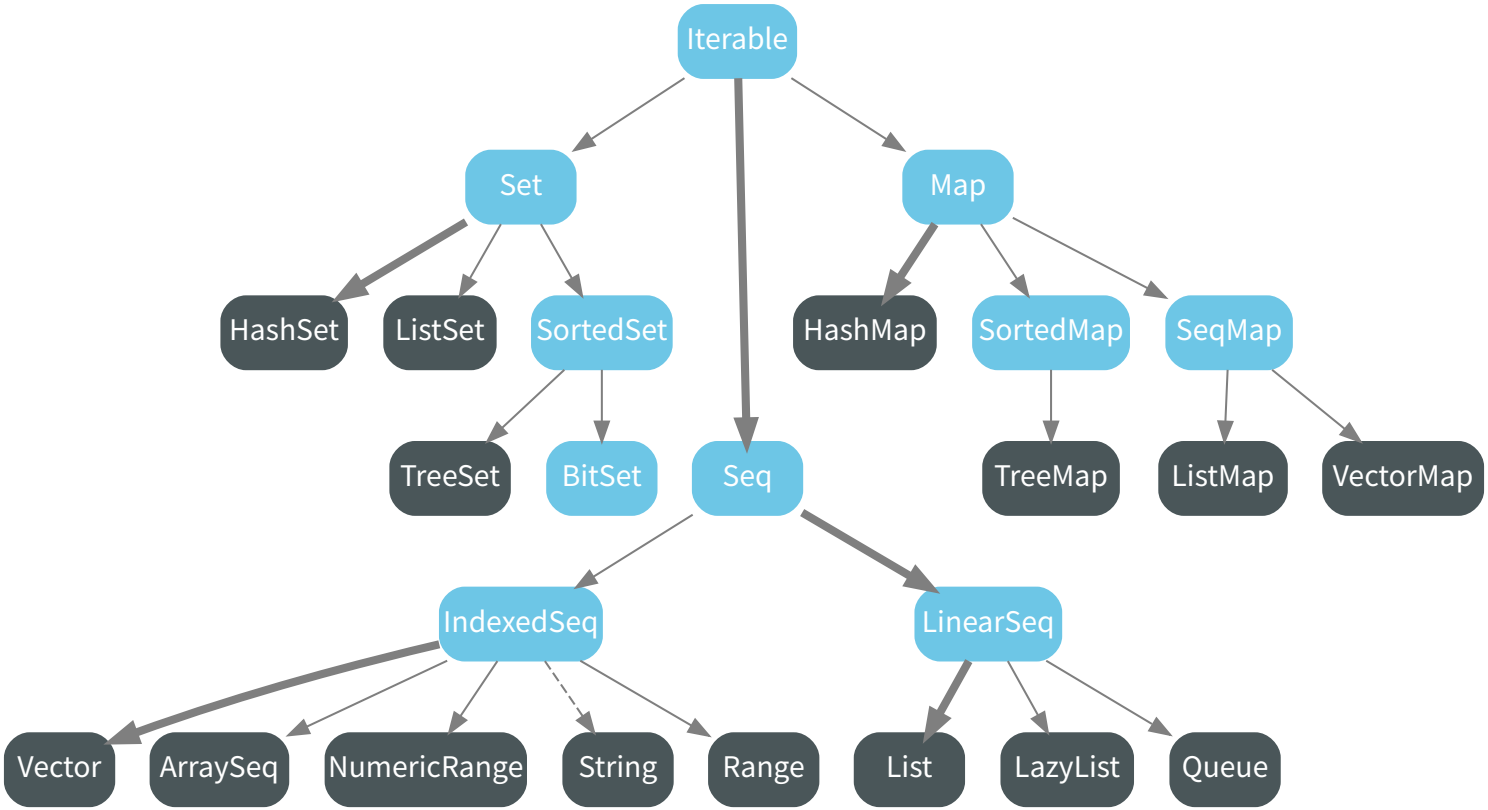
13. マップのキャッシュアクセス

getOrElseUpdate はキャッシュとして使っているマップへのアクセスで役に立つ。

```
scala> def f(s: String): String = {
     |   println("...zzzZZZ")
     |   Thread.sleep(100)
     |   s.reverse
     | }
def f(s: String): String
scala> val cache = collection.mutable.Map[String, String]()
val cache: scala.collection.mutable.Map[String, String] = HashMap()
scala> def cachedF(s: String): String = cache.getOrElseUpdate(s, f(s))
def cachedF(s: String): String

scala> println(cachedF("abc"))
...zzzZZZ
cba

scala> println(cachedF("abc"))
cba
```

14. 具象イミュータブルコレクションクラス

- List, LazyList, ArraySeq, Vector, Queue, Range, 圧縮ハッシュ配列マッププレフィックス木, 赤黒木, BitSet, VectorMap, ListMap



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

14. 具象イミュータブルコレクションクラス

- `List`: リスト
  - 有限のイミュータブルシーケンス
  - 先頭要素と先頭要素以外の部分リストへのアクセス、リストの先頭に新しい要素を挿入する `cons` 演算が定数時間で実行できる
  - 他の演算はリストの長さに比例した線形時間が必要

14. 具象イミュータブルコレクションクラス

- LazyList: 遅延リスト
  - 要素の計算を遅延実行できる
  - 長さが無限
  - パフォーマンス特性はリストと同じ

```
scala> val xs = 1 #:: 2 #:: 3 #:: LazyList.empty
val xs: LazyList[Int] = LazyList(<not computed>)

scala> println(xs)
LazyList(<not computed>)

scala> val xa = xs.toList
val xa: List[Int] = List(1, 2, 3)

scala> def fibFrom(a: Int, b: Int): LazyList[Int] = a #:: fibFrom(b, a + b)
def fibFrom(a: Int, b: Int): LazyList[Int]

scala> val fibs = fibFrom(1, 1).take(7)
val fibs: LazyList[Int] = LazyList(<not computed>)
```

```
15
16 scala> val f = fibs.toList
17 val f: List[Int] = List(1, 1, 2, 3, 5, 8, 13)
```

14. 具象イミュータブルコレクションクラス

- ArraySeq
  - リストでは下記の欠点がある
  - リストヘッドのアクセス、追加、削除は定数時間で処理できる
  - リストヘッド以外の要素のアクセス、変更はリスト内での要素の深さの線形時間を必要とする
  - ArraySeq では下記の特徴がある
  - コレクション内のどの要素にも一定時間でアクセスできる
  - 先頭への挿入は配列の長さの線形時間が必要
  - 1 個の要素の追加、更新では配列全体のコピーが必要になり配列の長さに比例した時間が必要

14. 具象イミュータブルコレクションクラス

- `Vector`
  - ベクトルの任意の要素に対するアクセスと更新は「実質的に一定時間」になる
  - リストヘッドへのアクセスや `ArraySeq` の要素の読み出しにかかる時間より長いが定数時間
  - ベクトルは広くて浅い木で表現される
  - すべてのノードが 32 個の要素か 32 個の部分木を持つ木構造

- 圧縮ハッシュ配列マッププレフィックス木
  - ハッシュトライ(Hash trie)はイミュータブルな集合、マップを効率的に実装するための標準的な方法
  - 圧縮ハッシュ配列マッププレフィックス木は JVM 上のハッシュトライの設計
  - すべてのノードが 32 個の要素か 32 個 の部分木を持つ木構造
  - 選択はハッシュコードに基づいて行われる

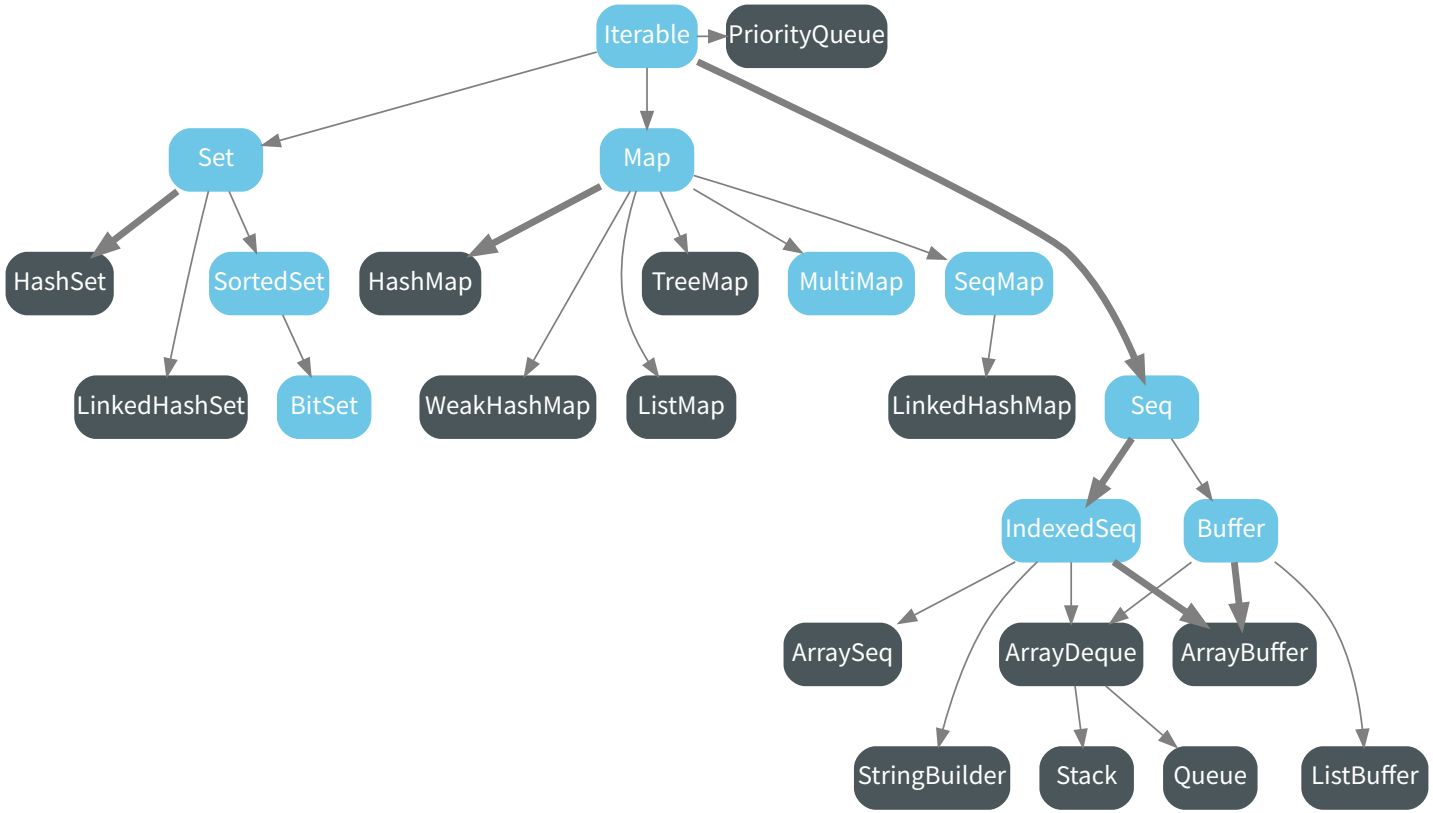
14. 具象イミュータブルコレクションクラス

- 赤黒木
  - 平衡二分木の一種
  - 演算は木のサイズの対数に比例する時間で終了する

```
scala> val xs = collection.immutable.TreeSet.empty[Int]
val xs: scala.collection.immutable.TreeSet[Int] = TreeSet()

scala> val xa = xs + 5 + 2 + 4
val xa: scala.collection.immutable.TreeSet[Int] = TreeSet(2, 4, 5)
```

15. 具象ミュータブルコレクションクラス

- `ArrayBuffer`, `ListBuffer`, `StringBuilder`, `ArrayDeque`, `Queue`, `Stack`, `ArraySeq`, Hash Table, Concurrent Map, `BitSet`



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

16. 配列

配列は Scala における特殊なタイプのコレクション。Scala 配列は 1 対 1 で Java 配列と対応している。

対応関係は下記のようになる。同時に Scala 配列は Java 配列よりも機能が豊富。

- Scala: `Array[Int]`
- Java: `int[]`

1. Scala 配列はジェネリックに使用することができる。`T` を型パラメータ、または抽象型として `Array[T]` を作成可能
2. Scala 配列は Scala シーケンスに対して互換性がある。`Seq[T]` が必要な場面で `Array[T]` を渡すことができる
3. Scala 配列はすべてのシーケンス演算をサポート

16. 配列


```
scala> val xs = (1 to 3).toArray
val xs: Array[Int] = Array(1, 2, 3)

scala> val xa = xs.map(x ⇒ x + 2)
val xa: Array[Int] = Array(3, 4, 5)

scala> val xb = xs.filter(x ⇒ x % 2 ≠ 0)
val xb: Array[Int] = Array(1, 3)

scala> val xc = xs.reverse
val xc: Array[Int] = Array(3, 2, 1)
```

ネイティブ配列の型としての表現は Seq のサブ型ではないので、配列がシーケンスのふりをすることはできない。配列が Seq として使われるときには暗黙のうちに scala.collection.mutable.ArraySeq という Seq のサブクラスで相列をラップしている。

```
scala> val xs : collection.Seq[Int] = (1 to 3).toArray
val xs: scala.collection.Seq[Int] = ArraySeq(1, 2, 3)

scala> val xa : Array[Int] = xs.toArray
val xa: Array[Int] = Array(1, 2, 3)

scala> val b = xs eq xa
val b: Boolean = false
```

配列に適用される暗黙の変換は、これ以外にも存在する。変換は単純に配列にすべてのシーケンスメソッドを「追加」するが配列自体をシーケンスに変換するわけではない。「追加」とはすべてのシーケンスメソッドをサポートする ArrayOps 型の別のオブジェクトで配列をラップするという意味である。ArrayOps オブジェクトはシーケンスメソッドの呼び出し後にアクセスできなくなる。

```
scala> val xs : collection.Seq[Int] = (1 to 3).toArray
val xs: scala.collection.Seq[Int] = ArraySeq(1, 2, 3)

scala> val xs = (1 to 3).toArray
val xs: Array[Int] = Array(1, 2, 3)

scala> val xa : collection.Seq[Int] = xs
val xa: scala.collection.Seq[Int] = ArraySeq(1, 2, 3)

scala> val xb = xa.reverse
val xb: scala.collection.Seq[Int] = ArraySeq(3, 2, 1)

```

```
scala> val xc : collection.ArrayOps[Int] = xs
val xc: scala.collection.ArrayOps[Int] = scala.collection.ArrayOps@2afe263

scala> val xd = xc.reverse
val xd: Array[Int] = Array(3, 2, 1)
```

通常は ArrayOps クラスの値を定義する必要はなく、単純に配列をレシーバーとして Seq メソッドを呼び出すだけで良い。暗黙の変換により ArrayOps オブジェクトが自動的に入り込んでくる。

```
scala> val xd = xs.reverse
val xd: Array[Int] = Array(3, 2, 1)

scala> val xe = intArrayOps(xs).reverse
val xe: Array[Int] = Array(3, 2, 1)
```

ArrayOps への変換は ArraySeq への変換の方が優先度が高い。ArrayOps への変換は Predef オブジェクトで定義されているのに対して ArraySeq への変換は Predef のスーパクラスである scala.LowPriorityImplicits で定義されている。サブクラス、サブオブジェクトでの暗黙の型変換は基底クラスでの暗黙の変換よりも優先度が高い。

```
// Array.scala
/*
 *  @hideImplicitConversion scala.Predef.booleanArrayOps
 *  @hideImplicitConversion scala.Predef.byteArrayOps
 *  @hideImplicitConversion scala.Predef.charArrayOps
 *  @hideImplicitConversion scala.Predef.doubleArrayOps
 *  @hideImplicitConversion scala.Predef.floatArrayOps
 *  @hideImplicitConversion scala.Predef.intArrayOps
 *  @hideImplicitConversion scala.Predef.longArrayOps
 *  @hideImplicitConversion scala.Predef.refArrayOps
 *  @hideImplicitConversion scala.Predef.shortArrayOps
 *  @hideImplicitConversion scala.Predef.unitArrayOps
 */
```

```
14
15 // ArrayOps.scala
16 final class ArrayOps[A](private val xs: Array[A]) extends AnyVal {
17   /** Returns a new array with the elements in reversed order. */
18   @inline def reverse: Array[A] = {
19     val len = xs.length
20     val res = new Array[A](len)
21     var i = 0
22     while(i < len) {
23       res(len-i-1) = xs(i)
24       i += 1
25     }
26     res
27   }
28 }
```

ジェネリックな型の指定

- `Array[T]` のようなジェネリック配列は実行時に Java のプリミティブ配列型にもオブジェクト配列にもなりうる
- これらのすべての型を包み込む実行時型は `AnyRef` 以外にない
- Scala コンパイラが生成するのは `Array[AnyRef]` である
- 実行時に `Array[T]` 型配列の要素へのアクセス、更新が発生するたびに一連の型テストで実際の配列型が確定し Java 配列の正しい配列演算を実行する
  - 型テストによって配列演算の速度はある程度遅くなる
  - 最大限のパフォーマンスが必要ならジェネリック配列ではなく具象型の配列を使った方がよい

ジェネリックな型配列を表現できるだけでは不足で、ジェネリックな配列を作成する方法も必要である。下記のように素直な実装では型パラメータ T が実行時に実際にどの型に対応づけられるかについての手がかりがコードに存在しないため与えられた情報だけでは型を判定できない。

```
scala> def evenElems[T](xs: Vector[T]): Array[T] =
     |   val arr = new Array[T]((xs.length + 1) / 2)
     |   for i ← 0 until xs.length by 2 do
     |     arr(i / 2) = xs(i)
     |   arr
     |
-- [E172] Type Error: ----------------------------------------------------------
2 |  val arr = new Array[T]((xs.length + 1) / 2)
  |                                         ^
  |                                         No ClassTag available for T
1 error found
```

実行時の実際の型引数が何かについてのヒントを与えることでコンパイラーの仕事を助ける。scala.reflect.ClassTag 型のクラスタグという形をとる。クラスタグは型引数の消去型の情報を与える。

```
scala> import scala.reflect.ClassTag
scala> def evenElems[T: ClassTag](xs: Vector[T]): Array[T] =
     |   val arr = new Array[T]((xs.length + 1) / 2)
     |   for i ← 0 until xs.length by 2 do
     |     arr(i / 2) = xs(i)
     |   arr
     |
def evenElems
  [T](xs: Vector[T])(implicit evidence$1: scala.reflect.ClassTag[T]): Array[T]
```

多くの場合、コンパイラーは自分でクラスタグを生成できる。消去型を予測するための情報が十分にある List[T] などの一部のジェネリック型でもクラスタグを生成できる。List[T] の消去型は List になる。Array[T] を作成するときに型パラメータ T のクラ

スタグを探す。コードに書かれていない ClassTag[T] 型の暗黙の引数を探す。そのような引数が見つかれば、それに基づいて正しい要素型の配列を作成する。見つからなければエラーメッセージを出力する。

evenElems の挙動を確認する。

```
scala> val xa = evenElems((1 to 5).toVector)
val xa: Array[Int] = Array(1, 3, 5)

scala> val xb = evenElems(('a' to 'x').toVector)
val xb: Array[Char] = Array(a, c, e, g, i, k, m, o, q, s, u, w)
```

Scala コンパイラは要素型のクラスタグを自動的に作成する。evenElems の暗黙の引数として渡している。コンパイラーは要素型が具象型ならどのようなものでもクラスタグを作成することができるが、引数自体がクラスタグのない他の型パラメータになっているとクラスタグを作成できない。

17. 等価性

コレクションライブラリーは等価性とハッシングについて統一的なアプローチをとっている。コレクションは集合、マップ、シーケンスに分類することができる。分類の異なるコレクションは同じ要素を格納していたとしても常に等価ではない。同じ分類内では、同じ要素を持つ場合に限り等しい。

コレクションは新しいコレクションを作成するメソッドを多数持っている。少なくとも 1 つのコレクションをレシーバーとし戻り値として別のコレクションを作成するため変換演算子(transformer)と呼ばれている。

変換演算子の実装方法は正格と遅延(または非正格)の 2 種類に分けられる。正格な変換演算子は新しいコレクションと共にすべての要素も構築する。非正格な変換演算子はコレクションのプロキシしか作成せず、要素はオンデマンドで構築される。

非正格な変換演算子の例として遅延マップ演算の実装を行う。lazyMap は引数のコレクション iter のすべての要素を反復処理することなく新しい Iterable を作成する。新しいコレクションの iterator が要素の要求を受け付けたときに引数の関数 f を要素に適用する。

```
scala> def lazyMap[T, U](iter: Iterable[T], f: T ⇒ U) = new Iterable[U]:
     |   def iterator = iter.iterator map f
     |
def lazyMap[T, U](iter: Iterable[T], f: T ⇒ U): Iterable[U]
```

すべての変換演算子を遅延実装している `LazyList` を除いて Scala のコレクションはすべての変換演算子がデフォルトで正格である。コレクションのビューによって、すべてのコレクションを遅延コレクションに、またはその逆に変換する体系的な方法が存在する。ビューは特殊なコレクションの一種であり何らかの基本コレクションに基づいているがすべての変換演算子を遅延実行している。

遅延処理を行わないと va のような例では中間コレクションが作成されてしまう。vd のようにビューを作成しておくと関数合成をして適用することができるため中間コレクションが作成されない。そのため遅延処理の方が無駄が少なく高速である。

```
scala> val v = (0 to 3).toVector
val v: Vector[Int] = Vector(0, 1, 2, 3)

scala> val va = v.map(x ⇒ x + 1).map(x ⇒ x * 2)
val va: Vector[Int] = Vector(2, 4, 6, 8)

scala> val vb = v.view
val vb: scala.collection.IndexedSeqView[Int] = IndexedSeqView(<not computed>)

scala> val vc = vb.to(Vector)
val vc: Vector[Int] = Vector(0, 1, 2, 3)

scala> val vd = v.view.map(x => x + 1).map(x => x * 2).to(Vector)
val vd: Vector[Int] = Vector(2, 4, 6, 8)
```

19. イテレーター

イテレーターはコレクションではなく、コレクションに含まれる要素に 1 つずつアクセスするための手法である。イテレーターの基本演算は next と hasNext の 2 つである。it.next() 呼び出しはイテレーターの次の要素を返し、イテレーターの状態を 1 つ先に進める。返す要素がない場合に next を呼び出すと NoSuchElementException が発生する。foreach を使用するとイテレーターが返すここの要素に対して関数を実行することができる。この場合でもイテレーターは末尾まで進んでしまうため注意が必要である。

```
scala> val it = Iterator((1 to 3)*)
val it: Iterator[Int] = <iterator>

scala> while true do
     |  println(it.next)
     |
1
2
3
java.util.NoSuchElementException: next on empty iterator
```

```
  ... 32 elided

scala> val it = Iterator((1 to 3)*)
val it: Iterator[Int] = <iterator>

scala> it foreach (x => println(x * x))
1
4
9

scala> it foreach (x => println(x * x))
// 何も表示されない

scala> val it = Iterator((1 to 3)*)
val it: Iterator[Int] = <iterator>

scala> for e <- it do
     |  println(e * e)
     |
1
```

```
4
9
```

20. まとめ

- コレクションにはイミュータブルとミュータブルのものがある
- コレクションは大きく分けて `Seq`, `Set`, `Map` から構成される

21. 疑問回答集

immutable なコレクションを使うメリットはあるのか

参考: https://scala-text.github.io/scala_text/collection.html

- 関数型プログラミングで多用する再帰との相性が良い
- 高階関数を用いて簡潔なプログラムを書くことができる
- 一度作成したコレクションが知らない箇所で変更されていない事を保証できる
- 並行に動作するプログラムの中で値を安全に受け渡すことができる