Functional programming with Scala

Collections

Djaouida ZAOUCHE, Romain DUJOL

CY TECH – ING2

2020 – 2021

Scala collection library

Definition

Container of an arbitrary number of elements **with same type**

Scala collection library

Definition

Container of an arbitrary number of elements **with same type**

Example (kinda) from already seen types : `Option`

Scala collection library

Definition

Container of an arbitrary number of elements **with same type**

Example (kinda) from already seen types : `Option`

Scala collection library

Scala collection library

Definition

Container of an arbitrary number of elements **with same type**

Example (kinda) from already seen types : `Option`

Scala collection library

- **New design since version 2.13** (see official documentation)
  (likely to be used in Scala 3)

Scala collection library

Definition

Container of an arbitrary number of elements **with same type**

Example (kinda) from already seen types : `Option`

Scala collection library

- **New design since version 2.13** (see official documentation)
  (likely to be used in Scala 3)
- For versions 2.8 to 2.12 :

Scala collection library

Definition

Container of an arbitrary number of elements **with same type**

Example (kinda) from already seen types : `Option`

Scala collection library

- **New design since version 2.13** (see official documentation)
  (likely to be used in Scala 3)
- For versions 2.8 to 2.12 :
  - see official documentation

Scala collection library

Definition

Container of an arbitrary number of elements **with same type**

Example (kinda) from already seen types : `Option`

Scala collection library

- **New design since version 2.13** (see official documentation)
  (likely to be used in Scala 3)
- For versions 2.8 to 2.12 :
  - see official documentation
  - see also official migration guide from 2.11 or 2.12 to 2.13

Scala collection library

Definition

Container of an arbitrary number of elements **with same type**

Example (kinda) from already seen types : `Option`

Scala collection library

- **New design since version 2.13** (see official documentation)
  (likely to be used in Scala 3)
- For versions 2.8 to 2.12 :
  - see official documentation
  - see also official migration guide from 2.11 or 2.12 to 2.13
- Standard usage is not changed much, but design is cleaner.

Scala collection library

Definition

Container of an arbitrary number of elements **with same type**

Example (kinda) from already seen types : `Option`

Scala collection library

- **New design since version 2.13** (see official documentation)
  (likely to be used in Scala 3)
- For versions 2.8 to 2.12 :
  - see official documentation
  - see also official migration guide from 2.11 or 2.12 to 2.13
- Standard usage is not changed much, but design is cleaner.

In this course

- **New design will be presented.**
- Some remarks about previous design may be adressed.

Scala collection library

Two main packages

- `scala.collection.immutable`
- `scala.collection.mutable`

Scala collection library

Two main packages

- `scala.collection.immutable`
- `scala.collection.mutable`

Of course, functional programming favors immutable collections *(and so does concurrent programming...)*

Scala collection library

Two main packages

- `scala.collection.immutable`
- `scala.collection.mutable`

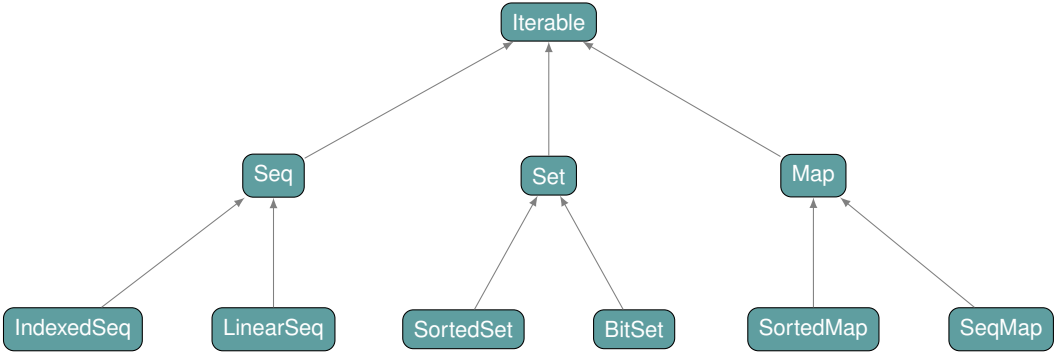
Of course, functional programming favors immutable collections *(and so does concurrent programming. . .)*

Rich set of. . .

- collection types (classes and traits)
- common methods for most collections
- methods for specific collections

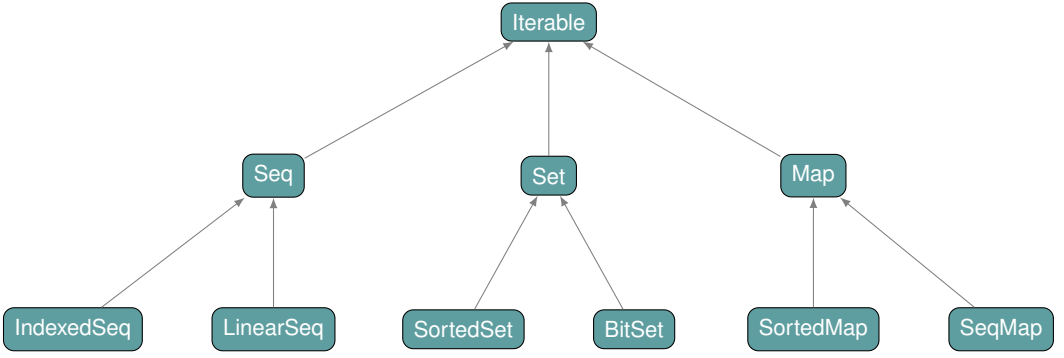
Top of hierarchy

Scala collection library



(Source : Scala 2.13 Collection API Charts)

Scala collection library



(Source : Scala 2.13 Collection API Charts)

Prior versions : **`Traversable`** at top of hierarchy

- introduced `foreach[U](f : A => U) : Unit`
  (now available in **`Iterable`** via **`IterableOnceOps`**)
- *removed in 2.13 due to underuse*

Scala collection hierarchy : **`Iterable`**

Any **`Iterable`** has an *iterator*

Scala collection hierarchy : `Iterable`

Any `Iterable` has an *iterator*

- Iterator is accessible via method `iterator`.

Scala collection hierarchy : `Iterable`

Any `Iterable` has an *iterator*

- Iterator is accessible via method `iterator`.
- An iterator is a mutable object with two important methods :

Scala collection hierarchy : `Iterable`

Any `Iterable` has an *iterator*

- Iterator is accessible via method `iterator`.
- An iterator is a mutable object with two important methods :
  - `hasNext` checks if there is elements left in the iteration ;

Scala collection hierarchy : `Iterable`

Any `Iterable` has an *iterator*

- Iterator is accessible via method `iterator`.
- An iterator is a mutable object with two important methods :
  - `hasNext` checks if there is elements left in the iteration ;
  - `next` gives the next element (if exists).

Scala collection hierarchy : **`Iterable`**

Any **`Iterable`** has an *iterator*

- Iterator is accessible via method `iterator`.
- An iterator is a mutable object with two important methods :
  - `hasNext` checks if there is elements left in the iteration ;
  - `next` gives the next element (if exists).

Example

```
scala> List(1, 3, 5) foreach { x => println(s"Current element is $x") }
Current element is 1
Current element is 3
Current element is 5
```

Scala collection hierarchy : **`Iterable`**

Any **`Iterable`** has an *iterator*

- Iterator is accessible via method `iterator`.
- An iterator is a mutable object with two important methods :
  - `hasNext` checks if there is elements left in the iteration ;
  - `next` gives the next element (if exists).

Example

```
scala> List(1, 3, 5) foreach { x => println(s"Current element is $x") }
Current element is 1
Current element is 3
Current element is 5

scala> val it = List(1, 3, 5).iterator
it: Iterator[Int] = non-empty iterator

scala> while (it.hasNext) println(s"Current element is ${it.next()})
Current element is 1
Current element is 3
Current element is 5
```

Specific collections

Specific collections

Seq

Scala collection hierarchy : **Seq**[**+T**]

Base trait for *sequences*

Scala collection hierarchy : **Seq** [+T]

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.

Scala collection hierarchy : **Seq** [**+T**]

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.
- Type parameter is **covariant**.

Scala collection hierarchy : `Seq[+T]`

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.
- Type parameter is **covariant**.
- Main methods : `head`, `tail`, `apply`, `length`
  + `update` (for mutable sequences)

Scala collection hierarchy : `Seq[+T]`

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.
- Type parameter is **covariant**.
- Main methods : `head`, `tail`, `apply`, `length`
  + `update` (for mutable sequences)
- *Default implementation* : **`List`**

Scala collection hierarchy : `Seq[+T]`

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.
- Type parameter is **covariant**.
- Main methods : `head`, `tail`, `apply`, `length`
  + `update` (for mutable sequences)
- *Default implementation* : `List`

Two main subtraits

Scala collection hierarchy : `Seq[+T]`

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.
- Type parameter is **covariant**.
- Main methods : `head`, `tail`, `apply`, `length`
  + `update` (for mutable sequences)
- *Default implementation* : `List`

Two main subtraits

- `IndexedSeq[T]` for sequences with fast random access, i.e. efficient methods `apply` and `length`

Scala collection hierarchy : `Seq[+T]`

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.
- Type parameter is **covariant**.
- Main methods : `head`, `tail`, `apply`, `length`
  + `update` (for mutable sequences)
- *Default implementation* : **`List`**

Two main subtraits

- **`IndexedSeq[T]`** for sequences with fast random access, i.e. efficient methods `apply` and `length`
  - *Default implementation* : **`Vector[T]`**

Scala collection hierarchy : `Seq[+T]`

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.
- Type parameter is **covariant**.
- Main methods : `head`, `tail`, `apply`, `length`
  + `update` (for mutable sequences)
- *Default implementation* : **`List`**

Two main subtraits

- **`IndexedSeq[T]`** for sequences with fast random access, i.e. efficient methods `apply` and `length`
  - *Default implementation* : **`Vector[T]`**
- **`LinearSeq[T]`** for sequences with linear random access, i.e. efficient methods `head` and `tail`

Scala collection hierarchy : `Seq[+T]`

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.
- Type parameter is **covariant**.
- Main methods : `head`, `tail`, `apply`, `length`
  + `update` (for mutable sequences)
- *Default implementation* : `List`

Two main subtraits

- `IndexedSeq[T]` for sequences with fast random access, i.e. efficient methods `apply` and `length`
  - *Default implementation* : `Vector[T]`
- `LinearSeq[T]` for sequences with linear random access, i.e. efficient methods `head` and `tail`
  - *Default implementation* : `List[T]`

Scala collection hierarchy : `Seq[+T]`

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.
- Type parameter is **covariant**.
- Main methods : `head`, `tail`, `apply`, `length`
  + `update` (for mutable sequences)
- *Default implementation* : `List`

Two main subtraits

- `IndexedSeq[T]` for sequences with fast random access, i.e. efficient methods `apply` and `length`
  - *Default implementation* : `Vector[T]`
- `LinearSeq[T]` for sequences with linear random access, i.e. efficient methods `head` and `tail`
  - *Default implementation* : `List[T]`
  - *Other implementations* : `Queue[T]`, `LazyList[T]`

Scala collection hierarchy : `Seq[+T]`

Base trait for *sequences*

- As name suggests, a sequence is an **ordered** collection.
- Type parameter is **covariant**.
- Main methods : `head`, `tail`, `apply`, `length`
  + `update` (for mutable sequences)
- *Default implementation* : **`List`**

Two main subtraits

- **`IndexedSeq[T]`** for sequences with fast random access, i.e. efficient methods `apply` and `length`
  - *Default implementation* : **`Vector[T]`**
- **`LinearSeq[T]`** for sequences with linear random access, i.e. efficient methods `head` and `tail`
  - *Default implementation* : **`List[T]`**
  - *Other implementations* : **`Queue[T]`**, **`LazyList[T]`**

*NB: In Scala, any **`String`** object is implicitly seen as a **`Seq[Char]`**.*

Scala collection hierarchy : Seq[+T]

Example

```
scala> Seq.empty(ou Seq())
res0: Seq[Nothing] = List()
```

Scala collection hierarchy : **Seq**[+T]

Example

```
scala> Seq.empty(ou Seq())
res0: Seq[Nothing] = List()

scala> val s = Seq(5, 1, 3, 1)
s: Seq[Int] = List(5, 1, 3, 1)
```

Scala collection hierarchy : **Seq**[+T]

Example

```
scala> Seq.empty(ou Seq())
res0: Seq[Nothing] = List()

scala> val s = Seq(5, 1, 3, 1)
s: Seq[Int] = List(5, 1, 3, 1)

scala> s.length
res1: Int = 4
```

Scala collection hierarchy : **Seq**[+T]

Example

```
scala> Seq.empty(ou Seq())
res0: Seq[Nothing] = List()

scala> val s = Seq(5, 1, 3, 1)
s: Seq[Int] = List(5, 1, 3, 1)

scala> s.length
res1: Int = 4

scala> s.head
res2: Int = 5
```

Scala collection hierarchy : Seq[+T]

Example

```
scala> Seq.empty(ou Seq())
res0: Seq[Nothing] = List()

scala> val s = Seq(5, 1, 3, 1)
s: Seq[Int] = List(5, 1, 3, 1)

scala> s.length
res1: Int = 4

scala> s.head
res2: Int = 5

scala> s.tail
res3: Seq[Int] = List(1, 3, 1)
```

Scala collection hierarchy : **Seq** [**+T**]

Example

```
scala> Seq.empty (ou Seq())
res0: Seq[Nothing] = List()

scala> val s = Seq(5, 1, 3, 1)
s: Seq[Int] = List(5, 1, 3, 1)

scala> s.length
res1: Int = 4

scala> s.head
res2: Int = 5

scala> s.tail
res3: Seq[Int] = List(1, 3, 1)

scala> s(1) (shortcut for s.apply(1))
res4: Int = 1 (WARNING ! Indexing starts at 0)
```

Scala collection hierarchy : **Seq** [**+T**]

Example

```
scala> Seq.empty (ou Seq())
res0: Seq[Nothing] = List()

scala> val s = Seq(5, 1, 3, 1)
s: Seq[Int] = List(5, 1, 3, 1)

scala> s.length
res1: Int = 4

scala> s.head
res2: Int = 5

scala> s.tail
res3: Seq[Int] = List(1, 3, 1)

scala> s(1) (shortcut for s.apply(1))
res4: Int = 1 (WARNING ! Indexing starts at 0)

scala> s == Seq(1, 1, 3, 5)
res5: Boolean = false
```

Scala collection hierarchy : **Seq**[+T]

Sequence operators

Scala collection hierarchy : **Seq**[**+T**]

Sequence operators

- ++ : concatenation

Scala collection hierarchy : Seq[+T]

Sequence operators

- ++ : concatenation
- :+ / +: : append/prepend element
  Hint: COLon (:) is on the COLlection side

Scala collection hierarchy : `Seq[+T]`

Sequence operators

- `++` : concatenation
- `:+` / `+:` : append/prepend element
  Hint: COLon (:) is on the COLlection side
- `reverse` : mirror sequence

Scala collection hierarchy : **Seq**[+T]

Sequence operators

- `++` : concatenation
- `:+` / `+:` : append/prepend element
  Hint: COLon (:) is on the COLlection side
- `reverse` : mirror sequence

Example

```
scala> Seq(5, 1, 3) ++ Seq(1, 4)
res0: Seq[Int] = List(5, 1, 3, 1, 4)
```

Scala collection hierarchy : **Seq**[+T]

Sequence operators

- `++` : concatenation
- `:+` / `+:` : append/prepend element
  Hint: COLon (:) is on the COLlection side
- `reverse` : mirror sequence

Example

```
scala> Seq(5, 1, 3) ++ Seq(1, 4)
res0: Seq[Int] = List(5, 1, 3, 1, 4)

scala> Seq(5, 1, 3) :+ 1
res1: Seq[Int] = List(5, 1, 3, 1)
```

Scala collection hierarchy : Seq[+T]

Sequence operators

- `++` : concatenation
- `:+` / `+:` : append/prepend element
  Hint: COLon (:) is on the COLlection side
- `reverse` : mirror sequence

Example

```
scala> Seq(5, 1, 3) ++ Seq(1, 4)
res0: Seq[Int] = List(5, 1, 3, 1, 4)

scala> Seq(5, 1, 3) :+ 1
res1: Seq[Int] = List(5, 1, 3, 1)

scala> 2 +: Seq(5, 1, 3)
res1: Seq[Int] = List(2, 5, 1, 3)
```

Scala collection hierarchy : **Seq** [+T]

Sequence operators

- `++` : concatenation
- `:+` / `+:` : append/prepend element
  Hint: COLon (:) is on the COLlection side
- `reverse` : mirror sequence

Example

```
scala> Seq(5, 1, 3) ++ Seq(1, 4)
res0: Seq[Int] = List(5, 1, 3, 1, 4)

scala> Seq(5, 1, 3) :+ 1
res1: Seq[Int] = List(5, 1, 3, 1)

scala> 2 +: Seq(5, 1, 3)
res1: Seq[Int] = List(2, 5, 1, 3)

scala> Seq(5, 1, 3).reverse
res2: Seq[Int] = List(3, 1, 5)
```

Specific collections

Set

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

- As name suggests, a set is an **unordered** collection **with no duplicates**.

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

- As name suggests, a set is an **unordered** collection **with no duplicates**.
- Type parameter is **invariant**.

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

- As name suggests, a set is an **unordered** collection **with no duplicates**.
- Type parameter is **invariant**.
- Main methods : `contains`, `diff`
  + `addOne`, `subtractOne` (for mutable sets)

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

- As name suggests, a set is an **unordered** collection **with no duplicates**.
- Type parameter is **invariant**.
- Main methods : `contains`, `diff`
  + `addOne`, `subtractOne` (for mutable sets)
- *Default implementation* : `HashSet[T]`

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

- As name suggests, a set is an **unordered** collection **with no duplicates**.
- Type parameter is **invariant**.
- Main methods : `contains`, `diff`
  + `addOne`, `subtractOne` (for mutable sets)
- *Default implementation* : `HashSet[T]`
- Subtrait `SortedSet[T]` is designed for sorted sets.
  Default implementation : `TreeSet[T]`

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

- As name suggests, a set is an **unordered** collection **with no duplicates**.
- Type parameter is **invariant**.
- Main methods : `contains`, `diff`
  + `addOne`, `subtractOne` (for mutable sets)
- *Default implementation* : `HashSet[T]`
- Subtrait `SortedSet[T]` is designed for sorted sets.
  Default implementation : `TreeSet[T]`

Set operators

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

- As name suggests, a set is an **unordered** collection **with no duplicates**.
- Type parameter is **invariant**.
- Main methods : `contains`, `diff`
  + `addOne`, `subtractOne` (for mutable sets)
- *Default implementation* : `HashSet[T]`
- Subtrait `SortedSet[T]` is designed for sorted sets.
  Default implementation : `TreeSet[T]`

Set operators

- `++` : set union

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

- As name suggests, a set is an **unordered** collection **with no duplicates**.
- Type parameter is **invariant**.
- Main methods : `contains`, `diff`
  + `addOne`, `subtractOne` (for mutable sets)
- *Default implementation* : `HashSet[T]`
- Subtrait `SortedSet[T]` is designed for sorted sets.
  Default implementation : `TreeSet[T]`

Set operators

- `++` : set union
- `&` : set intersection

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

- As name suggests, a set is an **unordered** collection **with no duplicates**.
- Type parameter is **invariant**.
- Main methods : `contains`, `diff`
  + `addOne`, `subtractOne` (for mutable sets)
- *Default implementation* : `HashSet[T]`
- Subtrait `SortedSet[T]` is designed for sorted sets.
  Default implementation : `TreeSet[T]`

Set operators

- `++` : set union
- `&` : set intersection
- `&~` : set difference

Scala collection hierarchy : `Set[T]`

Base trait for *sets*

- As name suggests, a set is an **unordered** collection **with no duplicates**.
- Type parameter is **invariant**.
- Main methods : `contains`, `diff`
  + `addOne`, `subtractOne` (for mutable sets)
- *Default implementation* : `HashSet[T]`
- Subtrait `SortedSet[T]` is designed for sorted sets.
  Default implementation : `TreeSet[T]`

Set operators

- `++` : set union
- `&` : set intersection
- `&~` : set difference
- `apply` : equivalent for `contains`

Scala collection hierarchy : `Set[T]`

Example

```
scala> val s = Set(5, 1, 3, 1)
s: scala.collection.immutable.Set[Int] = Set(5, 1, 3)
```

Scala collection hierarchy : `Set[T]`

Example

```
scala> val s = Set(5, 1, 3, 1)
s: scala.collection.immutable.Set[Int] = Set(5, 1, 3)

scala> s contains 4
res0: Boolean = false
```

Scala collection hierarchy : `Set[T]`

Example

```
scala> val s = Set(5, 1, 3, 1)
s: scala.collection.immutable.Set[Int] = Set(5, 1, 3)

scala> s contains 4
res0: Boolean = false

scala> s ++ Set(2, 3, 4)
res1: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)
```

Scala collection hierarchy : `Set[T]`

Example

```
scala> val s = Set(5, 1, 3, 1)
s: scala.collection.immutable.Set[Int] = Set(5, 1, 3)

scala> s contains 4
res0: Boolean = false

scala> s ++ Set(2, 3, 4)
res1: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)

scala> s & Set(2,3,4)
res2: scala.collection.immutable.Set[Int] = Set(3)
```

Scala collection hierarchy : `Set[T]`

Example

```
scala> val s = Set(5, 1, 3, 1)
s: scala.collection.immutable.Set[Int] = Set(5, 1, 3)

scala> s contains 4
res0: Boolean = false

scala> s ++ Set(2, 3, 4)
res1: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)

scala> s & Set(2,3,4)
res2: scala.collection.immutable.Set[Int] = Set(3)

scala> s &~ Set(2,3,4)
res3: scala.collection.immutable.Set[Int] = Set(1, 5)
```

Scala collection hierarchy : `Set[T]`

Example

```
scala> val s = Set(5, 1, 3, 1)
s: scala.collection.immutable.Set[Int] = Set(5, 1, 3)

scala> s contains 4
res0: Boolean = false

scala> s ++ Set(2, 3, 4)
res1: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)

scala> s & Set(2,3,4)
res2: scala.collection.immutable.Set[Int] = Set(3)

scala> s &~ Set(2,3,4)
res3: scala.collection.immutable.Set[Int] = Set(1, 5)

scala> s == Set(1, 3, 5)
res4: Boolean = true
```

Scala collection hierarchy : `Set[T]`

Example

```
scala> val s = Set(5, 1, 3, 1)
s: scala.collection.immutable.Set[Int] = Set(5, 1, 3)

scala> s contains 4
res0: Boolean = false

scala> s ++ Set(2, 3, 4)
res1: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)

scala> s & Set(2,3,4)
res2: scala.collection.immutable.Set[Int] = Set(3)

scala> s &~ Set(2,3,4)
res3: scala.collection.immutable.Set[Int] = Set(1, 5)

scala> s == Set(1, 3, 5)
res4: Boolean = true

scala> s(1) (shortcut for s.apply(1))
res5: Boolean = true
```

Scala collection hierarchy : `Set[T]`

Empty set `Set.empty`

Scala collection hierarchy : `Set[T]`

Empty set `Set.empty`

- Since parameter type is invariant, `Set[Nothing]` is not a subtype of `Set[T]`.

Scala collection hierarchy : `Set[T]`

Empty set `Set.empty`

- Since parameter type is invariant, `Set[Nothing]` is not a subtype of `Set[T]`.
- **So exact type of elements must be known beforehand.**

Scala collection hierarchy : `Set[T]`

Empty set `Set.empty`

- Since parameter type is invariant, `Set[Nothing]` is not a subtype of `Set[T]`.
- **So exact type of elements must be known beforehand.**

Example

```
scala> val emptySet = Set.empty(ou Set())
emptySet: scala.collection.immutable.Set[Nothing] = Set()

scala> emptySet + 1
<console>:13: error: type mismatch;
 found   : Int(1)
 required: Nothing
       emptySet + 1
                  ^
```

Scala collection hierarchy : `Set[T]`

Empty set `Set.empty`

- Since parameter type is invariant, `Set[Nothing]` is not a subtype of `Set[T]`.
- **So exact type of elements must be known beforehand.**

Example

```
scala> val emptySet = Set.empty(ou Set())
emptySet: scala.collection.immutable.Set[Nothing] = Set()

scala> emptySet + 1
<console>:13: error: type mismatch;
 found   : Int(1)
 required: Nothing
       emptySet + 1
                  ^

scala> val emptySet = Set.empty[Int](ou Set[Int]())
emptySet: scala.collection.immutable.Set[Int] = Set()

scala> emptySet + 1
res0: scala.collection.immutable.Set[Int] = Set(1)
```

Specific collections

Map

Scala collection hierarchy : `Map[K, +V]`

Base trait for *maps*

Scala collection hierarchy : `Map[K, +V]`

Base trait for *maps*

- Type parameter for keys is **invariant**.

Scala collection hierarchy : `Map[K, +V]`

Base trait for *maps*

- Type parameter for keys is **invariant**.
- Type parameter for values is **covariant**.

Scala collection hierarchy : `Map[K, +V]`

Base trait for *maps*

- Type parameter for keys is **invariant**.
- Type parameter for values is **covariant**.
- Main methods : `get`, `removed`
  + `addOne`, `subtractOne` (for mutable maps)

Scala collection hierarchy : `Map[K, +V]`

Base trait for *maps*

- Type parameter for keys is **invariant**.
- Type parameter for values is **covariant**.
- Main methods : `get`, `removed`
  + `addOne`, `subtractOne` (for mutable maps)
- *Default implementation* : `HashMap`

Scala collection hierarchy : `Map[K, +V]`

Base trait for *maps*

- Type parameter for keys is **invariant**.
- Type parameter for values is **covariant**.
- Main methods : `get`, `removed`
  + `addOne`, `subtractOne` (for mutable maps)
- *Default implementation* : `HashMap`

Subtraits

Scala collection hierarchy : `Map[K, +V]`

Base trait for *maps*

- Type parameter for keys is **invariant**.
- Type parameter for values is **covariant**.
- Main methods : `get`, `removed`
  + `addOne`, `subtractOne` (for mutable maps)
- *Default implementation* : `HashMap`

Subtraits

- `SeqMap[K, V]` for maps as sequences of key/value mappings *(version 2.13 only)*

Scala collection hierarchy : `Map[K, +V]`

Base trait for *maps*

- Type parameter for keys is **invariant**.
- Type parameter for values is **covariant**.
- Main methods : `get`, `removed`
  + `addOne`, `subtractOne` (for mutable maps)
- *Default implementation* : `HashMap`

Subtraits

- `SeqMap[K, V]` for maps as sequences of key/value mappings *(version 2.13 only)*
  - *Default implementation* : `VectorMap[K, V]`

Scala collection hierarchy : `Map[K, +V]`

Base trait for *maps*

- Type parameter for keys is **invariant**.
- Type parameter for values is **covariant**.
- Main methods : `get`, `removed`
  + `addOne`, `subtractOne` (for mutable maps)
- *Default implementation* : `HashMap`

Subtraits

- `SeqMap[K, V]` for maps as sequences of key/value mappings *(version 2.13 only)*
  - *Default implementation* : `VectorMap[K, V]`
- `SortedMap` for maps ordered with respect to keys

Scala collection hierarchy : `Map[K, +V]`

Base trait for *maps*

- Type parameter for keys is **invariant**.
- Type parameter for values is **covariant**.
- Main methods : `get`, `removed`
  + `addOne`, `subtractOne` (for mutable maps)
- *Default implementation* : `HashMap`

Subtraits

- `SeqMap[K, V]` for maps as sequences of key/value mappings *(version 2.13 only)*
  - *Default implementation* : `VectorMap[K, V]`
- `SortedMap` for maps ordered with respect to keys
  - *Default implementation* : `TreeMap[K, V]`

Scala collection hierarchy : `Map[K, +V]`

Example

```
scala> val m = Map("dze" -> "Cergy", "rdl" -> "Pau")
m: scala.collection.immutable.Map[String, String]
   = Map(dze -> Cergy, rdl -> Pau)
```

Scala collection hierarchy : `Map[K, +V]`

Example

```
scala> val m = Map("dze" -> "Cergy", "rdl" -> "Pau")
m: scala.collection.immutable.Map[String, String]
   = Map(dze -> Cergy, rdl -> Pau)

scala> m("rdl")
res0: String = Pau
```

Scala collection hierarchy : `Map[K, +V]`

Example

```
scala> val m = Map("dze" -> "Cergy", "rdl" -> "Pau")
m: scala.collection.immutable.Map[String, String]
   = Map(dze -> Cergy, rdl -> Pau)

scala> m("rdl")
res0: String = Pau

scala> m("bge")
java.util.NoSuchElementException: key not found: bge
  at scala.collection.immutable.Map$Map2.apply(Map.scala:298)
  ... 40 elided
```

Scala collection hierarchy : `Map[K, +V]`

Example

```
scala> val m = Map("dze" -> "Cergy", "rdl" -> "Pau")
m: scala.collection.immutable.Map[String, String]
   = Map(dze -> Cergy, rdl -> Pau)

scala> m("rdl")
res0: String = Pau

scala> m("bge")
java.util.NoSuchElementException: key not found: bge
  at scala.collection.immutable.Map$Map2.apply(Map.scala:298)
  ... 40 elided

scala> m.get("bge")
res1: Option[String] = None
```

Scala collection hierarchy : `Map[K, +V]`

Example

```
scala> val m = Map("dze" -> "Cergy", "rdl" -> "Pau")
m: scala.collection.immutable.Map[String, String]
   = Map(dze -> Cergy, rdl -> Pau)

scala> m("rdl")
res0: String = Pau

scala> m("bge")
java.util.NoSuchElementException: key not found: bge
  at scala.collection.immutable.Map$Map2.apply(Map.scala:298)
  ... 40 elided

scala> m.get("bge")
res1: Option[String] = None

scala> val m1 = m + ("bge" -> "Cergy")
m1 : scala.collection.immutable.Map[String, String]
     = Map(dze -> Cergy, rdl -> Pau, bge -> Cergy)
```

Scala collection hierarchy : `Map[K, +V]`

Example

```
scala> val m = Map("dze" -> "Cergy", "rdl" -> "Pau")
m: scala.collection.immutable.Map[String, String]
   = Map(dze -> Cergy, rdl -> Pau)

scala> m("rdl")
res0: String = Pau

scala> m("bge")
java.util.NoSuchElementException: key not found: bge
  at scala.collection.immutable.Map$Map2.apply(Map.scala:298)
  ... 40 elided

scala> m.get("bge")
res1: Option[String] = None

scala> val m1 = m + ("bge" -> "Cergy")
m1 : scala.collection.immutable.Map[String, String]
     = Map(dze -> Cergy, rdl -> Pau, bge -> Cergy)

scala> m1 - "rdl"
res2 : scala.collection.immutable.Map[String, String]
       = Map(dze -> Cergy, bge -> Cergy)
```

Scala collection hierarchy : `Map[K, +V]`

Empty map `Map.empty`

Scala collection hierarchy : `Map[K, +V]`

Empty map `Map.empty`

- Since parameter type K is invariant, `Map[Nothing,V]` is not a subtype of `Map[T,V]`.

Scala collection hierarchy : `Map[K, +V]`

Empty map `Map.empty`

- Since parameter type K is invariant, `Map[Nothing,V]` is not a subtype of `Map[T,V]`.
- **So exact type of keys must be known beforehand.**

Scala collection hierarchy : `Map[K, +V]`

Empty map `Map.empty`

- Since parameter type K is invariant, `Map[Nothing,V]` is not a subtype of `Map[T,V]`.
- **So exact type of keys must be known beforehand.**

Example

```
scala> val emptyMap = Map.empty(ou Map())
emptySet: scala.collection.immutable.Map[Nothing,Nothing] = Map()

scala> emptyMap + ("test" -> "failed")
<console>:13: error: type mismatch;
 found   : (String, String)
 required: (Nothing, ?)
       emptyMap + ("test" -> "failed")
                           ^
```

Scala collection hierarchy : `Map[K, +V]`

Empty map `Map.empty`

- Since parameter type K is invariant, `Map[Nothing,V]` is not a subtype of `Map[T,V]`.
- **So exact type of keys must be known beforehand.**

Example

```
scala> val emptyMap = Map.empty(ou Map())
emptySet: scala.collection.immutable.Map[Nothing,Nothing] = Map()

scala> emptyMap + ("test" -> "failed")
<console>:13: error: type mismatch;
 found   : (String, String)
 required: (Nothing, ?)
       emptyMap + ("test" -> "failed")
                           ^

scala> val emptyMap = Map.empty[String, String]
(ou Map[String, String]())
emptySet: scala.collection.immutable.Map[String] = Map()

scala> emptyMap + ("test" -> "success")
res0: scala.collection.immutable.Map[String, String] = Map(test -> success)
```

Specific collections

Conversions

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` $\leftrightarrow$ `Set[T]`

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` $\leftrightarrow$ `Set[T]`

- from sets to sequences : `to(Seq)`

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` $\leftrightarrow$ `Set[T]`

- from sets to sequences : `to(Seq)`
  - `to(IndexedSeq)`, `to(Vector)`

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` $\leftrightarrow$ `Set[T]`

- from sets to sequences : `to(Seq)`
  - `to(IndexedSeq)`, `to(Vector)`
  - `to(LinearSeq)`, `to(List)`

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` $\leftrightarrow$ `Set[T]`

- from sets to sequences : `to(Seq)`
  - `to(IndexedSeq)`, `to(Vector)`
  - `to(LinearSeq)`, `to(List)`
- from sequences to sets : `to(Set)`

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` $\leftrightarrow$ `Set[T]`

- from sets to sequences : `to(Seq)`
  - `to(IndexedSeq)`, `to(Vector)`
  - `to(LinearSeq)`, `to(List)`
- from sequences to sets : `to(Set)`
  - `to(SortedSet)`

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` $\leftrightarrow$ `Set[T]`

- from sets to sequences : `to(Seq)`
  - `to(IndexedSeq)`, `to(Vector)`
  - `to(LinearSeq)`, `to(List)`
- from sequences to sets : `to(Set)`
  - `to(SortedSet)`

Conversion `Map[K, V]` $\leftrightarrow$ `Seq[(K, V)]` or `Set[(K, V)]`

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` $\leftrightarrow$ `Set[T]`

- from sets to sequences : `to(Seq)`
  - `to(IndexedSeq)`, `to(Vector)`
  - `to(LinearSeq)`, `to(List)`
- from sequences to sets : `to(Set)`
  - `to(SortedSet)`

Conversion `Map[K, V]` $\leftrightarrow$ `Seq[(K, V)]` or `Set[(K, V)]`

- from maps to sequences/sets : same as above

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` ↔ `Set[T]`

- from sets to sequences : `to(Seq)`
  - `to(IndexedSeq)`, `to(Vector)`
  - `to(LinearSeq)`, `to(List)`
- from sequences to sets : `to(Set)`
  - `to(SortedSet)`

Conversion `Map[K, V]` ↔ `Seq[(K, V)]` or `Set[(K, V)]`

- from maps to sequences/sets : same as above
- from sets/sequences to maps : `to(Map)`

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` ↔ `Set[T]`

- from sets to sequences : `to(Seq)`
  - `to(IndexedSeq)`, `to(Vector)`
  - `to(LinearSeq)`, `to(List)`
- from sequences to sets : `to(Set)`
  - `to(SortedSet)`

Conversion `Map[K, V]` ↔ `Seq[(K, V)]` or `Set[(K, V)]`

- from maps to sequences/sets : same as above
- from sets/sequences to maps : `to(Map)`

Syntax introduced in version 2.13

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` ↔ `Set[T]`

- from sets to sequences : `to(Seq)`
  - `to(IndexedSeq)`, `to(Vector)`
  - `to(LinearSeq)`, `to(List)`
- from sequences to sets : `to(Set)`
  - `to(SortedSet)`

Conversion `Map[K, V]` ↔ `Seq[(K, V)]` or `Set[(K, V)]`

- from maps to sequences/sets : same as above
- from sets/sequences to maps : `to(Map)`

Syntax introduced in version 2.13

- Old syntax : `to[Seq]`, `to[Set]`, `to[Map]`, etc...

Scala collection hierarchy : conversions with `to(_)`

Conversion `Seq[T]` ↔ `Set[T]`

- from sets to sequences : `to(Seq)`
  - `to(IndexedSeq)`, `to(Vector)`
  - `to(LinearSeq)`, `to(List)`
- from sequences to sets : `to(Set)`
  - `to(SortedSet)`

Conversion `Map[K, V]` ↔ `Seq[(K, V)]` or `Set[(K, V)]`

- from maps to sequences/sets : same as above
- from sets/sequences to maps : `to(Map)`

Syntax introduced in version 2.13

- Old syntax : `to[Seq]`, `to[Set]`, `to[Map]`, etc...
- Other available syntax : `toSeq`, `toSet`, `toMap`, etc...

Common methods

Common methods

Basic methods and search methods

Common methods on `Iterable[T]`

Simple methods

Common methods on `Iterable[T]`

Simple methods

- `isEmpty` and `nonEmpty` $\equiv$ `!isEmpty`

Common methods on `Iterable[T]`

Simple methods

- `isEmpty` and `nonEmpty` $\equiv$ `!isEmpty`
- `size` (`length` is an alias of `size` for sequences)

Common methods on `Iterable[T]`

Simple methods

- `isEmpty` and `nonEmpty` $\equiv$ `!isEmpty`
- `size` (`length` is an alias of `size` for sequences)

Search methods

Common methods on `Iterable[T]`

Simple methods

- `isEmpty` and `nonEmpty` $\equiv$ `!isEmpty`
- `size` (`length` is an alias of `size` for sequences)

Search methods

- `exists(p : T => Boolean) : Boolean`
  Checks if at least one element `x` satisfies `p(x) == true`

Common methods on `Iterable[T]`

Simple methods

- `isEmpty` and `nonEmpty` $\equiv$ `!isEmpty`
- `size` (`length` is an alias of `size` for sequences)

Search methods

- `exists(p : T => Boolean) : Boolean`
  Checks if at least one element x satisfies `p(x) == true`
- `forall(p : T => Boolean) : Boolean`
  Checks if all elements x satisfy `p(x) == true`

Common methods on `Iterable[T]`

Simple methods

- `isEmpty` and `nonEmpty` $\equiv$ `!isEmpty`
- `size` (`length` is an alias of `size` for sequences)

Search methods

- `exists(p : T => Boolean) : Boolean`
  Checks if at least one element x satisfies `p(x) == true`
- `forall(p : T => Boolean) : Boolean`
  Checks if all elements x satisfy `p(x) == true`
- `count(p : T => Boolean) : Boolean`
  Counts number of elements satisfying `p(x) == true`

Common methods on `Iterable[T]`

Simple methods

- `isEmpty` and `nonEmpty` $\equiv$ `!isEmpty`
- `size` (`length` is an alias of `size` for sequences)

Search methods

- `exists(p : T => Boolean) : Boolean`
  Checks if at least one element x satisfies `p(x) == true`
- `forall(p : T => Boolean) : Boolean`
  Checks if all elements x satisfy `p(x) == true`
- `count(p : T => Boolean) : Boolean`
  Counts number of elements satisfying `p(x) == true`
- `find(p : T => Boolean) : Option[T]`
  Find first element x satisfying `p(x) == true`
  *(returns **`None`** if no such element is found)*

Common methods

Filter methods

Filter methods on `Iterable[T]`

Filter methods

Filter methods on `Iterable[T]`

Filter methods

- `filter(p : T => Boolean)`
  Returns subcollection of all elements x satisfying `p(x) == true`

Filter methods on `Iterable[T]`

Filter methods

- `filter(p : T => Boolean)`
  Returns subcollection of all elements x satisfying `p(x) == true`
- `filterNot(p : T => Boolean)`
  ≡ `filter(x => !p(x))`

Filter methods on `Iterable[T]`

Filter methods

- `filter(p : T => Boolean)`
  Returns subcollection of all elements x satisfying `p(x) == true`
- `filterNot(p : T => Boolean)`
  $\equiv$ `filter(x => !p(x))`
- `partition(p : T => Boolean)`
  $\equiv$ `(filter p, filterNot p)`
  (more efficient than mere calls to `filter` and `filterNot`)

Filter methods on `Iterable[T]`

Filter methods

- `filter(p : T => Boolean)`
  Returns subcollection of all elements x satisfying `p(x) == true`
- `filterNot(p : T => Boolean)`
  $\equiv$ `filter(x => !p(x))`
- `partition(p : T => Boolean)`
  $\equiv$ `(filter p, filterNot p)`
  (more efficient than mere calls to `filter` and `filterNot`)

Features

Filter methods on `Iterable[T]`

Filter methods

- `filter(p : T => Boolean)`
  Returns subcollection of all elements x satisfying `p(x) == true`
- `filterNot(p : T => Boolean)`
  $\equiv$ `filter(x => !p(x))`
- `partition(p : T => Boolean)`
  $\equiv$ `(filter p, filterNot p)`
  (more efficient than mere calls to `filter` and `filterNot`)

Features

- **Type result is the same as actual type of input collection**
  e.g. if called on `List[T]`, result has type `List[T]`

Filter methods on `Iterable[T]`

Filter methods

- `filter(p : T => Boolean)`
  Returns subcollection of all elements x satisfying `p(x) == true`
- `filterNot(p : T => Boolean)`
  $\equiv$ `filter(x => !p(x))`
- `partition(p : T => Boolean)`
  $\equiv$ `(filter p, filterNot p)`
  (more efficient than mere calls to `filter` and `filterNot`)

Features

- **Type result is the same as actual type of input collection**
  e.g. if called on `List[T]`, result has type `List[T]`
- When called on ordered collections, *relative order is preserved*.

Common methods

Extraction methods

Extraction methods on `Iterable[T]`

Fixed extraction methods

Extraction methods on `Iterable[T]`

Fixed extraction methods

- `take(n : Int)` / `takeRight(n : Int)`
  Take first/last n elements of collection

Extraction methods on `Iterable[T]`

Fixed extraction methods

- `take(n : Int)` / `takeRight(n : Int)`
  Take first/last n elements of collection
- `drop(n : Int)` / `dropRight(n : Int)`
  Drop first/last n elements of collection

Extraction methods on `Iterable[T]`

Fixed extraction methods

- `take(n : Int)` / `takeRight(n : Int)`
  Take first/last n elements of collection
- `drop(n : Int)` / `dropRight(n : Int)`
  Drop first/last n elements of collection
- `splitAt(n : Int)` ≡ `(take n, drop n)`
  (more efficient than mere calls to `take` and `drop`)

Extraction methods on `Iterable[T]`

Fixed extraction methods

- `take(n : Int)` / `takeRight(n : Int)`
  Take first/last n elements of collection
- `drop(n : Int)` / `dropRight(n : Int)`
  Drop first/last n elements of collection
- `splitAt(n : Int)` ≡ `(take n, drop n)`
  (more efficient than mere calls to `take` and `drop`)

Features

Extraction methods on `Iterable[T]`

Fixed extraction methods

- `take(n : Int)` / `takeRight(n : Int)`
  Take first/last n elements of collection
- `drop(n : Int)` / `dropRight(n : Int)`
  Drop first/last n elements of collection
- `splitAt(n : Int)` ≡ `(take n, drop n)`
  (more efficient than mere calls to `take` and `drop`)

Features

- **Type result is the same as actual type of input collection**
  e.g. if called on `List[T]`, result has type `List[T]`

Extraction methods on `Iterable[T]`

Fixed extraction methods

- `take(n : Int)` / `takeRight(n : Int)`
  Take first/last n elements of collection
- `drop(n : Int)` / `dropRight(n : Int)`
  Drop first/last n elements of collection
- `splitAt(n : Int)` ≡ `(take n, drop n)`
  (more efficient than mere calls to `take` and `drop`)

Features

- **Type result is the same as actual type of input collection**
  e.g. if called on `List[T]`, result has type `List[T]`
- When called on unordered collections, **method is not guaranteed to be pure**.

Extraction methods on `Iterable[T]`

Conditional extraction methods

Extraction methods on `Iterable[T]`

Conditional extraction methods

- `takeWhile(p : T => Boolean)`
  Take elements while elements x satisfies `p(x) == true`

Extraction methods on `Iterable[T]`

Conditional extraction methods

- `takeWhile(p : T => Boolean)`
  Take elements while elements x satisfies `p(x) == true`
- `dropWhile(p : T => Boolean)`
  Drop elements while elements x satisfies `p(x) == true`

Extraction methods on `Iterable[T]`

Conditional extraction methods

- `takeWhile(p : T => Boolean)`
  Take elements while elements x satisfies `p(x) == true`
- `dropWhile(p : T => Boolean)`
  Drop elements while elements x satisfies `p(x) == true`
- `span(p : T => Boolean)`
  $\equiv$ `(takeWhile p, dropWhile p)`
  (more efficient than mere calls to `takeWhile` and `dropWhile`)

Extraction methods on `Iterable[T]`

Conditional extraction methods

- `takeWhile(p : T => Boolean)`
  Take elements while elements x satisfies `p(x) == true`
- `dropWhile(p : T => Boolean)`
  Drop elements while elements x satisfies `p(x) == true`
- `span(p : T => Boolean)`
  $\equiv$ `(takeWhile p, dropWhile p)`
  (more efficient than mere calls to `takeWhile` and `dropWhile`)

Features

- **Type result is the same as actual type of input collection**
  e.g. if called on `List[T]`, result has type `List[U]`
- When called on unordered collections, **method is not guaranteed to be pure**.

Common methods

Transformation methods

Transformation methods on `Iterable[T]`

Transformation methods

- `flatten` "flattens" a collection of collections
- `map(f : T => U)`
  Returns the collection of results of `f` for all elements
- `flatMap(f : T => Iterable[U])` ≡ `map(f).flatten`

Features

Transformation methods on `Iterable[T]`

Transformation methods

- `flatten` "flattens" a collection of collections
- `map(f : T => U)`
  Returns the collection of results of `f` for all elements
- `flatMap(f : T => Iterable[U])` ≡ `map(f).flatten`

Features

- **Type result is the same as actual type of input collection**
  e.g. if called on `List[T]`, result has type `List[U]`

Transformation methods on `Iterable[T]`

Transformation methods

- `flatten` "flattens" a collection of collections
- `map(f : T => U)`
  Returns the collection of results of `f` for all elements
- `flatMap(f : T => Iterable[U])` ≡ `map(f).flatten`

Features

- **Type result is the same as actual type of input collection**
  e.g. if called on `List[T]`, result has type `List[U]`
- When called on ordered collections, *relative order is preserved*.

Transformation methods on `Iterable[T]`

Example

```
scala> Seq(Seq(5, 1, 1), Seq(1)).flatten
res0: Seq[Int] = List(5, 1, 1, 1)
```

Transformation methods on `Iterable[T]`

Example

```
scala> Seq(Seq(5, 1, 1), Seq(1)).flatten
res0: Seq[Int] = List(5, 1, 1, 1)

scala> Set(Seq(5, 1, 1), Seq(1)).flatten
res1: scala.collection.immutable.Set[Int] = Set(5, 1)
```

Transformation methods on `Iterable[T]`

Example

```
scala> Seq(Seq(5, 1, 1), Seq(1)).flatten
res0: Seq[Int] = List(5, 1, 1, 1)

scala> Set(Seq(5, 1, 1), Seq(1)).flatten
res1: scala.collection.immutable.Set[Int] = Set(5, 1)

scala> Seq(Set(5, 1, 1), Set(1)).flatten
res2: scala.collection.immutable.Set[Int] = Seq(5, 1, 1)
```

Transformation methods on `Iterable[T]`

Example

```
scala> Seq(Seq(5, 1, 1), Seq(1)).flatten
res0: Seq[Int] = List(5, 1, 1, 1)

scala> Set(Seq(5, 1, 1), Seq(1)).flatten
res1: scala.collection.immutable.Set[Int] = Set(5, 1)

scala> Seq(Set(5, 1, 1), Set(1)).flatten
res2: scala.collection.immutable.Set[Int] = Seq(5, 1, 1)

scala> Seq(5, 1, 2) map { 1.0 / _ }
res3: Seq[Double] = List(0.2, 1.0, 0.5)
```

Transformation methods on `Iterable[T]`

Example

```
scala> Seq(Seq(5, 1, 1), Seq(1)).flatten
res0: Seq[Int] = List(5, 1, 1, 1)

scala> Set(Seq(5, 1, 1), Seq(1)).flatten
res1: scala.collection.immutable.Set[Int] = Set(5, 1)

scala> Seq(Set(5, 1, 1), Set(1)).flatten
res2: scala.collection.immutable.Set[Int] = Seq(5, 1, 1)

scala> Seq(5, 1, 2) map { 1.0 / _ }
res3: Seq[Double] = List(0.2, 1.0, 0.5)

scala> Set(5, 1, 2) map { 1.0 / _ }
res4: scala.collection.immutable.Set[Double] = Set(0.2, 1.0, 0.5)
```

Transformation methods on `Iterable[T]`

Example

```
scala> Seq(Seq(5, 1, 1), Seq(1)).flatten
res0: Seq[Int] = List(5, 1, 1, 1)

scala> Set(Seq(5, 1, 1), Seq(1)).flatten
res1: scala.collection.immutable.Set[Int] = Set(5, 1)

scala> Seq(Set(5, 1, 1), Set(1)).flatten
res2: scala.collection.immutable.Set[Int] = Seq(5, 1, 1)

scala> Seq(5, 1, 2) map { 1.0 / _ }
res3: Seq[Double] = List(0.2, 1.0, 0.5)

scala> Set(5, 1, 2) map { 1.0 / _ }
res4: scala.collection.immutable.Set[Double] = Set(0.2, 1.0, 0.5)

scala> Seq(5, 1, 2) flatMap { x => List(x, x) }
res5 : Seq[Int] = List(5, 5, 1, 1, 2, 2)
```

Transformation methods on `Iterable[T]`

Example

```
scala> Seq(Seq(5, 1, 1), Seq(1)).flatten
res0: Seq[Int] = List(5, 1, 1, 1)

scala> Set(Seq(5, 1, 1), Seq(1)).flatten
res1: scala.collection.immutable.Set[Int] = Set(5, 1)

scala> Seq(Set(5, 1, 1), Set(1)).flatten
res2: scala.collection.immutable.Set[Int] = Seq(5, 1, 1)

scala> Seq(5, 1, 2) map { 1.0 / _ }
res3: Seq[Double] = List(0.2, 1.0, 0.5)

scala> Set(5, 1, 2) map { 1.0 / _ }
res4: scala.collection.immutable.Set[Double] = Set(0.2, 1.0, 0.5)

scala> Seq(5, 1, 2) flatMap { x => List(x, x) }
res5 : Seq[Int] = List(5, 5, 1, 1, 2, 2)

scala> Set(5, 1, 2) flatMap { x => List(x, x) }
res6 : scala.collection.immutable.Set[Int] = Set(5, 1, 2)
```

`for`-comprehensions by example

`for (x <- c) yield f(x)` ≡ `c map f`

```
scala> for (x <- Seq(5, 1, 2)) yield 1.0 / x
res0: Seq[Double] = List(0.2, 1.0, 0.5)

scala> Seq(5, 1, 2) map { x => 1.0 / x }
res1: Seq[Double] = List(0.2, 1.0, 0.5)
```

`for`-comprehensions by example

```
for (x <- c) yield f(x) ≡ c map f
```

```
scala> for (x <- Seq(5, 1, 2)) yield 1.0 / x
res0: Seq[Double] = List(0.2, 1.0, 0.5)

scala> Seq(5, 1, 2) map { x => 1.0 / x }
res1: Seq[Double] = List(0.2, 1.0, 0.5)
```

```
for (x <- c if cond(x)) yield f(x) ≡ (c filter cond) map f
```

```
scala> for (x <- Seq(5, 1, 2) if x > 1) yield 1.0 / x
res2: Seq[Double] = Seq(0.2, 0.5)

scala> (Seq(5, 1, 2) filter {x => x > 1}) map { x => 1.0 / x }
res3: Seq[Double] = Seq(0.2, 0.5)
```

`for`-comprehensions by example

```
for (x <- c) yield f(x) ≡ c map f
```

```
scala> for (x <- Seq(5, 1, 2)) yield 1.0 / x
res0: Seq[Double] = List(0.2, 1.0, 0.5)

scala> Seq(5, 1, 2) map { x => 1.0 / x }
res1: Seq[Double] = List(0.2, 1.0, 0.5)
```

```
for (x <- c if cond(x)) yield f(x) ≡ (c filter cond) map f
```

```
scala> for (x <- Seq(5, 1, 2) if x > 1) yield 1.0 / x
res2: Seq[Double] = Seq(0.2, 0.5)

scala> (Seq(5, 1, 2) filter {x => x > 1}) map { x => 1.0 / x }
res3: Seq[Double] = Seq(0.2, 0.5)
```

```
for (x <- c1 ; y <- c2) yield f(x, y) ≡ c1 flatMap (x => c2 map (y => f(x, y)))
```

```
scala> for (x <- Seq(5, 1, 2) ; y <- Seq(2, 3)) yield x + y
res4: Seq[Int] = List(7, 8, 3, 4, 4, 5)

scala> Seq(5, 1, 2) flatMap (x => Seq(2, 3) map (y => x + y))
res4: Seq[Int] = List(7, 8, 3, 4, 4, 5)
```

`for`-comprehensions by example

Do not forget the `yield` keyword !

`for`-comprehensions by example

Do not forget the `yield` keyword !

Example

```
scala> def couplesWithSum(n : Int) =
         for (i <- 0 to n ;
              j <- 0 to n if i + j == n)
           yield (i, j)
couplesWithSum(n: Int): IndexedSeq[(Int, Int)]

scala> couplesWithSum(4)
res1: IndexeqSeq[(Int, Int)] = Vector((0,4), (1,3), (2,2), (3,1), (4,0))
```

`for`-comprehensions by example

Do not forget the `yield` keyword !

Example

```
scala> def couplesWithSum(n : Int) =
         for (i <- 0 to n ;
              j <- 0 to n if i + j == n)
           yield (i, j)
couplesWithSum(n: Int): IndexedSeq[(Int, Int)]

scala> couplesWithSum(4)
res1: IndexeqSeq[(Int, Int)] = Vector((0,4), (1,3), (2,2), (3,1), (4,0))
```

Think DBMS. . .

`for`-comprehensions by example

Do not forget the `yield` keyword !

Example

```
scala> def couplesWithSum(n : Int) =
         for (i <- 0 to n ;
              j <- 0 to n if i + j == n)
           yield (i, j)
couplesWithSum(n: Int): IndexedSeq[(Int, Int)]

scala> couplesWithSum(4)
res1: IndexeqSeq[(Int, Int)] = Vector((0,4), (1,3), (2,2), (3,1), (4,0))
```

Think DBMS...

- Object $\equiv$ Item in database

`for`-comprehensions by example

Do not forget the `yield` keyword !

Example

```
scala> def couplesWithSum(n : Int) =
         for (i <- 0 to n ;
              j <- 0 to n if i + j == n)
           yield (i, j)
couplesWithSum(n: Int): IndexedSeq[(Int, Int)]

scala> couplesWithSum(4)
res1: IndexeqSeq[(Int, Int)] = Vector((0,4), (1,3), (2,2), (3,1), (4,0))
```

Think DBMS...

- Object ≡ Item in database
- Collection ≡ Table in database

`for`-comprehensions by example

Do not forget the `yield` keyword !

Example

```
scala> def couplesWithSum(n : Int) =
         for (i <- 0 to n ;
              j <- 0 to n if i + j == n)
           yield (i, j)
couplesWithSum(n: Int): IndexedSeq[(Int, Int)]

scala> couplesWithSum(4)
res1: IndexeqSeq[(Int, Int)] = Vector((0,4), (1,3), (2,2), (3,1), (4,0))
```

Think DBMS...

- Object ≡ Item in database
- Collection ≡ Table in database
- `for (x <- table)` ≡ `SELECT * FROM table`

`for`-comprehensions by example

Do not forget the `yield` keyword !

Example

```
scala> def couplesWithSum(n : Int) =
         for (i <- 0 to n ;
              j <- 0 to n if i + j == n)
           yield (i, j)
couplesWithSum(n: Int): IndexedSeq[(Int, Int)]

scala> couplesWithSum(4)
res1: IndexeqSeq[(Int, Int)] = Vector((0,4), (1,3), (2,2), (3,1), (4,0))
```

Think DBMS...

- Object ≡ Item in database
- Collection ≡ Table in database
- `for (x <- table)` ≡ `SELECT * FROM table`
- `if cond(x)` ≡ `WHERE cond(x)`

Common methods

Aggregation methods

Aggregation methods on `Iterable[T]`

`fold`-like aggregation methods

Aggregation methods on `Iterable[T]`

`fold`-like aggregation methods

- `foldLeft[U](z : U)(op : (U, T) => U) : U`
  ≡ `op(...(op(op(z, a_1), a_2) ...), a_n)`

Aggregation methods on `Iterable[T]`

`fold`-like aggregation methods

- `foldLeft[U](z : U)(op : (U, T) => U) : U`
  ≡ `op(...(op(op(z, a_1), a_2) ...), a_n)`
- `foldRight[U](z : U)(op : (T, U) => U) : U`
  ≡ `op(a1, op(a2, op(... , op(an, z))))`

Aggregation methods on `Iterable[T]`

`fold`-like aggregation methods

- `foldLeft[U](z : U)(op : (U, T) => U) : U`
  $\equiv$ `op(...(op(op(z, a_1), a_2) ...), a_n)`
- `foldRight[U](z : U)(op : (T, U) => U) : U`
  $\equiv$ `op(a1, op(a2, op(... , op(an, z))))`
- `fold[T1 >: T](z : T1)(op : (T1, T1) => T1) : T1`
  behaves like above **if op is associative**

Aggregation methods on `Iterable[T]`

`fold`-like aggregation methods

- `foldLeft[U](z : U)(op : (U, T) => U) : U`
  ≡ `op(...(op(op(z, a_1), a_2) ...), a_n)`
- `foldRight[U](z : U)(op : (T, U) => U) : U`
  ≡ `op(a1, op(a2, op(... , op(an, z))))`
- `fold[T1 >: T](z : T1)(op : (T1, T1) => T1) : T1`
  behaves like above **if op is associative**

Features

Aggregation methods on `Iterable[T]`

`fold`-like aggregation methods

- `foldLeft[U](z : U)(op : (U, T) => U) : U`
  $\equiv$ `op(...(op(op(z, a_1), a_2) ...), a_n)`
- `foldRight[U](z : U)(op : (T, U) => U) : U`
  $\equiv$ `op(a1, op(a2, op(... , op(an, z))))`
- `fold[T1 >: T](z : T1)(op : (T1, T1) => T1) : T1`
  behaves like above **if op is associative**

Features

- If collection is empty, result is always `z`.

Aggregation methods on `Iterable[T]`

`fold`-like aggregation methods

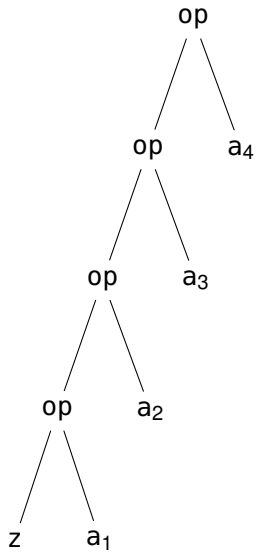
- `foldLeft[U](z : U)(op : (U, T) => U) : U`
  ≡ `op(...(op(op(z, a_1), a_2) ...), a_n)`
- `foldRight[U](z : U)(op : (T, U) => U) : U`
  ≡ `op(a1, op(a2, op(... , op(an, z))))`
- `fold[T1 >: T](z : T1)(op : (T1, T1) => T1) : T1`
  behaves like above **if op is associative**

Features

- If collection is empty, result is always `z`.
- When called on unordered collections, **method is not guaranteed to be pure**
  (except `fold` if `op` is associative AND commutative)

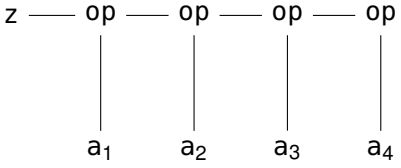
Folding process for `foldLeft`

Example on 4-element list `List(a1, a2, a3, a4)`



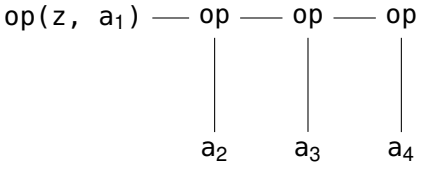
Folding process for `foldLeft`

```
(List(a1, a2, a3, a4) foldLeft z)(op)
```



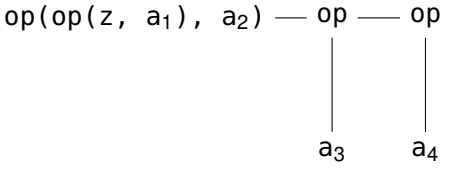
Folding process for `foldLeft`

```
(List(a1, a2, a3, a4) foldLeft z)(op)
```



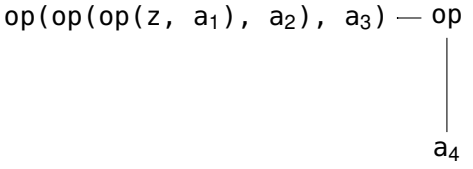
Folding process for `foldLeft`

$(\texttt{List}(a_1, a_2, a_3, a_4)\ \texttt{foldLeft}\ z)(op)$



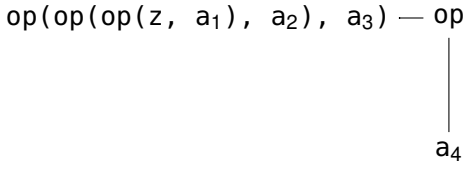
Folding process for `foldLeft`

```
(List(a1, a2, a3, a4) foldLeft z)(op)
```



Folding process for `foldLeft`

```
(List(a1, a2, a3, a4) foldLeft z)(op)
```

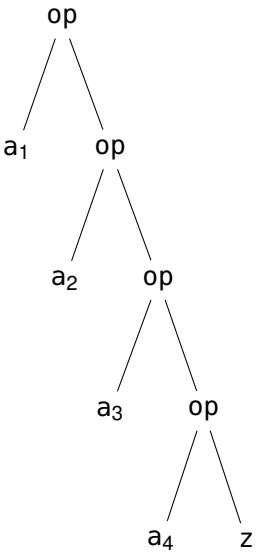


Résultat final

```
op(op(op(op(z, a1), a2), a3), a4)
```

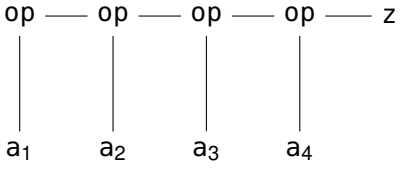
Folding process for `foldRight`

Example on 4-element list `List`(a_1, a_2, a_3, a_4)



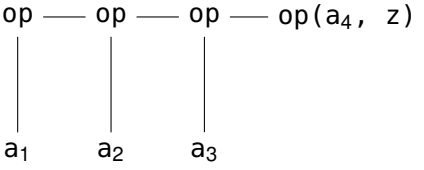
Folding process for `foldRight`

```
(List(a1, a2, a3, a4) foldRight z)(op)
```



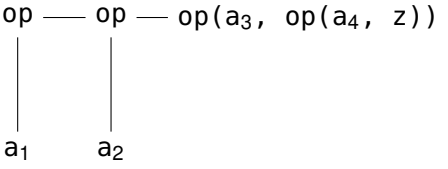
Folding process for `foldRight`

```
(List(a1, a2, a3, a4) foldRight z)(op)
```



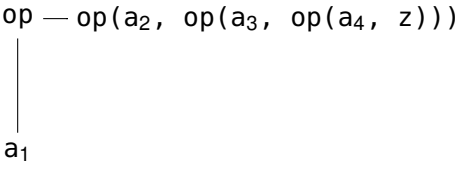
Folding process for `foldRight`

```
(List(a1, a2, a3, a4) foldRight z)(op)
```



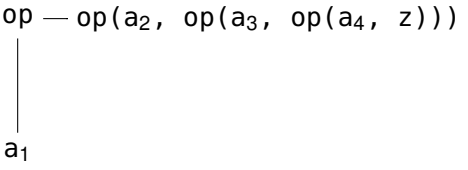
Folding process for `foldRight`

$$(\texttt{List}(a_1, a_2, a_3, a_4)\ \texttt{foldRight}\ z)(op)$$



Folding process for `foldRight`

```
(List(a1, a2, a3, a4) foldRight z)(op)
```



Résultat final

```
op(a1, op(a2, op(a3, op(a4, z))))
```

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

- In `foldLeft` :

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

- In `foldLeft` :
  - result is accumulated in the *left* parameter of `op` ;

foldLeft vs foldRight

How to tell foldLeft from foldRight ?

- In `foldLeft` :
  - result is accumulated in the *left* parameter of `op` ;
  - elements are taken from the *left* ;

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

- In `foldLeft` :
  - result is accumulated in the *left* parameter of `op` ;
  - elements are taken from the *left* ;
  - tree is folded *leftwise*.

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

- In `foldLeft` :
  - result is accumulated in the *left* parameter of `op` ;
  - elements are taken from the *left* ;
  - tree is folded *leftwise*.
- In `foldRight` :

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

- In `foldLeft` :
  - result is accumulated in the *left* parameter of `op` ;
  - elements are taken from the *left* ;
  - tree is folded *leftwise*.
- In `foldRight` :
  - result is accumulated in the *right* parameter of `op` ;

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

- In `foldLeft` :
  - result is accumulated in the *left* parameter of `op` ;
  - elements are taken from the *left* ;
  - tree is folded *leftwise*.
- In `foldRight` :
  - result is accumulated in the *right* parameter of `op` ;
  - elements are taken from the *right* ;

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

- In `foldLeft` :
  - result is accumulated in the *left* parameter of `op` ;
  - elements are taken from the *left* ;
  - tree is folded *leftwise*.
- In `foldRight` :
  - result is accumulated in the *right* parameter of `op` ;
  - elements are taken from the *right* ;
  - tree is folded *rightwise*.

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

- In `foldLeft` :
  - result is accumulated in the *left* parameter of `op` ;
  - elements are taken from the *left* ;
  - tree is folded *leftwise*.
- In `foldRight` :
  - result is accumulated in the *right* parameter of `op` ;
  - elements are taken from the *right* ;
  - tree is folded *rightwise*.

Both are inherently recursive

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

- In `foldLeft` :
  - result is accumulated in the *left* parameter of `op` ;
  - elements are taken from the *left* ;
  - tree is folded *leftwise*.
- In `foldRight` :
  - result is accumulated in the *right* parameter of `op` ;
  - elements are taken from the *right* ;
  - tree is folded *rightwise*.

Both are inherently recursive

- `foldLeft` is inherently tail-recursive.

`foldLeft` vs `foldRight`

How to tell `foldLeft` from `foldRight` ?

- In `foldLeft` :
  - result is accumulated in the *left* parameter of `op` ;
  - elements are taken from the *left* ;
  - tree is folded *leftwise*.
- In `foldRight` :
  - result is accumulated in the *right* parameter of `op` ;
  - elements are taken from the *right* ;
  - tree is folded *rightwise*.

Both are inherently recursive

- `foldLeft` is inherently tail-recursive.
- `foldRight` is not.

Aggregations methods on `Iterable[T]`

Compute all combinations of elements of a set

```
scala> def combinations[T](s : Set[T]) =
         (s foldLeft Set(Set[T]())) { case (z, x) => z ++ (z map { _ + x }) }
combinations: [T](s: Seq[T])Seq[Seq[T]]
```

Aggregations methods on `Iterable[T]`

Compute all combinations of elements of a set

```
scala> def combinations[T](s : Set[T]) =
         (s foldLeft Set(Set[T]())) { case (z, x) => z ++ (z map { _ + x }) }
combinations: [T](s: Seq[T])Seq[Seq[T]]

scala> combinations((1 to 3).to(Set))
res0: Set[Set[Int]] = HashSet(Set(), Set(1, 3), Set(2), Set(2, 3), Set(3),
                              Set(1, 2), Set(1), Set(1, 2, 3))
```

Aggregations methods on `Iterable[T]`

Compute all combinations of elements of a set

```
scala> def combinations[T](s : Set[T]) =
         (s foldLeft Set(Set[T]())) { case (z, x) => z ++ (z map { _ + x }) }
combinations: [T](s: Seq[T])Seq[Seq[T]]

scala> combinations((1 to 3).to(Set))
res0: Set[Set[Int]] = HashSet(Set(), Set(1, 3), Set(2), Set(2, 3), Set(3),
                              Set(1, 2), Set(1), Set(1, 2, 3))
```

Find corresponding existing methods

Aggregations methods on `Iterable[T]`

Compute all combinations of elements of a set

```
scala> def combinations[T](s : Set[T]) =
         (s foldLeft Set(Set[T]())) { case (z, x) => z ++ (z map { _ + x }) }
combinations: [T](s: Seq[T])Seq[Seq[T]]

scala> combinations((1 to 3).to(Set))
res0: Set[Set[Int]] = HashSet(Set(), Set(1, 3), Set(2), Set(2, 3), Set(3),
                              Set(1, 2), Set(1), Set(1, 2, 3))
```

Find corresponding existing methods

- `fold(0)(_ + _)`

Aggregations methods on `Iterable[T]`

Compute all combinations of elements of a set

```
scala> def combinations[T](s : Set[T]) =
         (s foldLeft Set(Set[T]())) { case (z, x) => z ++ (z map { _ + x }) }
combinations: [T](s: Seq[T])Seq[Seq[T]]

scala> combinations((1 to 3).to(Set))
res0: Set[Set[Int]] = HashSet(Set(), Set(1, 3), Set(2), Set(2, 3), Set(3),
                              Set(1, 2), Set(1), Set(1, 2, 3))
```

Find corresponding existing methods

- `fold(0)(_ + _)`
- `foldLeft(0){ case (z, x) => z + 1 }`

Aggregations methods on `Iterable[T]`

Compute all combinations of elements of a set

```
scala> def combinations[T](s : Set[T]) =
         (s foldLeft Set(Set[T]())) { case (z, x) => z ++ (z map { _ + x }) }
combinations: [T](s: Seq[T])Seq[Seq[T]]

scala> combinations((1 to 3).to(Set))
res0: Set[Set[Int]] = HashSet(Set(), Set(1, 3), Set(2), Set(2, 3), Set(3),
                              Set(1, 2), Set(1), Set(1, 2, 3))
```

Find corresponding existing methods

- `fold(0)(_ + _)`
- `foldLeft(0){ case (z, x) => z + 1 }`
- `foldLeft(0){ case (z, x) => if (p(x)) z+1 else z }`

Aggregations methods on `Iterable[T]`

`reduce`-like aggregation methods

Aggregations methods on `Iterable[T]`

reduce-like aggregation methods

- `reduceLeft[T1 >: T](op : (T1, T) => T1) : T1`
  ≡ `(tail foldLeft head)(op)`

Aggregations methods on `Iterable[T]`

reduce-like aggregation methods

- `reduceLeft[T1 >: T](op : (T1, T) => T1) : T1`
  `≡ (tail foldLeft head)(op)`
- `reduceRight[T1 >: T](op : (T, T1) => T1) : T1`
  `≡ (init foldLeft last)(op)`

Aggregations methods on `Iterable[T]`

reduce-like aggregation methods

- `reduceLeft[T1 >: T](op : (T1, T) => T1) : T1`
  $\equiv$ `(tail foldLeft head)(op)`
- `reduceRight[T1 >: T](op : (T, T1) => T1) : T1`
  $\equiv$ `(init foldLeft last)(op)`
- `reduce[T1 >: T](op : (T1, T1) => T1) : T1`
  behaves like above **if op is associative**

Aggregations methods on `Iterable[T]`

`reduce`-like aggregation methods

- `reduceLeft[T1 >: T](op : (T1, T) => T1) : T1`
  ≡ `(tail foldLeft head)(op)`
- `reduceRight[T1 >: T](op : (T, T1) => T1) : T1`
  ≡ `(init foldLeft last)(op)`
- `reduce[T1 >: T](op : (T1, T1) => T1) : T1`
  behaves like above **if op is associative**

Features

Aggregations methods on `Iterable[T]`

`reduce`-like aggregation methods

- `reduceLeft[T1 >: T](op : (T1, T) => T1) : T1`
  $\equiv$ `(tail foldLeft head)(op)`
- `reduceRight[T1 >: T](op : (T, T1) => T1) : T1`
  $\equiv$ `(init foldLeft last)(op)`
- `reduce[T1 >: T](op : (T1, T1) => T1) : T1`
  behaves like above **if op is associative**

Features

- *If collection is empty, methods throw an exception.*
  `Option`-valued version are also defined.

Aggregations methods on `Iterable[T]`

`reduce`-like aggregation methods

- `reduceLeft[T1 >: T](op : (T1, T) => T1) : T1`
  $\equiv$ `(tail foldLeft head)(op)`
- `reduceRight[T1 >: T](op : (T, T1) => T1) : T1`
  $\equiv$ `(init foldLeft last)(op)`
- `reduce[T1 >: T](op : (T1, T1) => T1) : T1`
  behaves like above **if op is associative**

Features

- *If collection is empty, methods throw an exception.*
  `Option`-valued version are also defined.
- When called on unordered collections, **method is not guaranteed to be pure**
  (except `reduce` if `op` is associative AND commutative)

Aggregations methods on `Iterable[T]`

`fold` vs `reduce`

Aggregations methods on `Iterable[T]`

`fold` vs `reduce`

- `fold` is more general

Aggregations methods on `Iterable[T]`

`fold` vs `reduce`

- `fold` is more general
- `reduce` does not need a `z` parameter.

Aggregations methods on `Iterable[T]`

`fold` vs `reduce`

- `fold` is more general
- `reduce` does not need a `z` parameter.

Examples

```
scala> Set(5, 1, 2) reduce (_ + _)
res0: Int = 8
```

Aggregations methods on `Iterable[T]`

`fold` vs `reduce`

- `fold` is more general
- `reduce` does not need a `z` parameter.

Examples

```
scala> Set(5, 1, 2) reduce (_ + _)
res0: Int = 8

scala> Set[Int]() reduce (_ + _)
java.lang.UnsupportedOperationException: empty.reduceLeft
  at scala.collection.IterableOnceOps.reduceLeft(IterableOnce.scala:723)
  at scala.collection.IterableOnceOps.reduceLeft$(IterableOnce.scala:720)
  at scala.collection.AbstractIterable.reduceLeft(Iterable.scala:920)
  at scala.collection.IterableOnceOps.reduce(IterableOnce.scala:692)
  at scala.collection.IterableOnceOps.reduce$(IterableOnce.scala:692)
  at scala.collection.AbstractIterable.reduce(Iterable.scala:920)
  ... 40 elided
```

Aggregations methods on `Iterable[T]`

`fold` vs `reduce`

- `fold` is more general
- `reduce` does not need a `z` parameter.

Examples

```
scala> Set(5, 1, 2) reduce (_ + _)
res0: Int = 8

scala> Set[Int]() reduce (_ + _)
java.lang.UnsupportedOperationException: empty.reduceLeft
  at scala.collection.IterableOnceOps.reduceLeft(IterableOnce.scala:723)
  at scala.collection.IterableOnceOps.reduceLeft$(IterableOnce.scala:720)
  at scala.collection.AbstractIterable.reduceLeft(Iterable.scala:920)
  at scala.collection.IterableOnceOps.reduce(IterableOnce.scala:692)
  at scala.collection.IterableOnceOps.reduce$(IterableOnce.scala:692)
  at scala.collection.AbstractIterable.reduce(Iterable.scala:920)
  ... 40 elided

scala> Set[Int]() reduceOption (_ + _)
res1: Option[Int] = None
```

Common methods

Grouping methods

Grouping methods on `Iterable[T]`

Grouping methods

Grouping methods on `Iterable[T]`

Grouping methods

- `groupBy[K](f : T => K)`
  Builds a map such that value at any key is the subcollection of elements whose values by `f` is that key.

Grouping methods on `Iterable[T]`

Grouping methods

- `groupBy[K](f : T => K)`
  Builds a map such that value at any key is the subcollection of elements whose values by `f` is that key.
- `groupMap[K, U](key : T => K)(f : T => U)`
  $\equiv$ `groupBy(key).map(k -> group => k -> group map f)`
  Same as before, but applies in addition `f` to each element of subcollections

Grouping methods on `Iterable[T]`

Grouping methods

- `groupBy[K](f : T => K)`
  Builds a map such that value at any key is the subcollection of elements whose values by `f` is that key.
- `groupMap[K, U](key : T => K)(f : T => U)`
  $\equiv$ `groupBy(key).map(k -> group => k -> group map f)`
  Same as before, but applies in addition `f` to each element of subcollections
- `groupMapReduce[K, U](key : T => K)(f : T => U)(op : (U, U) => U)`
  $\equiv$ `groupMap(key)(f).map(k -> c => k -> c reduce op)`
  Same as before, but replaces transformed subcollections by their aggregation

Grouping methods on `Iterable[T]`

Grouping methods

- `groupBy[K](f : T => K)`
  Builds a map such that value at any key is the subcollection of elements whose values by `f` is that key.
- `groupMap[K, U](key : T => K)(f : T => U)`
  $\equiv$ `groupBy(key).map(k -> group => k -> group map f)`
  Same as before, but applies in addition `f` to each element of subcollections
- `groupMapReduce[K, U](key : T => K)(f : T => U)(op : (U, U) => U)`
  $\equiv$ `groupMap(key)(f).map(k -> c => k -> c reduce op)`
  Same as before, but replaces transformed subcollections by their aggregation

New in version 2.13 : `groupMap`, `groupMapReduce`

Common methods on `Iterable[T]`

Combining methods

Common methods on `Iterable[T]`

Combining methods

- `zip(other : Iterable[U]) : Iterable[(T, U)]`
  Builds the collection of pairs taken element-wise

Common methods on `Iterable[T]`

Combining methods

- `zip(other : Iterable[U]) : Iterable[(T, U)]`
  Builds the collection of pairs taken element-wise
- `unzip`
  Reverse operation of `zip` and return a pair collection from a collection of pairs (i.e. T must be a couple of types)

Common methods on `Iterable[T]`

Combining methods

- `zip(other : Iterable[U]) : Iterable[(T, U)]`
  Builds the collection of pairs taken element-wise
- `unzip`
  Reverse operation of `zip` and return a pair collection from a collection of pairs (i.e. T must be a couple of types)
- `zipWithIndex : Iterable[(T, Int)]`
  Builds the collection of pairs element/index

Common methods on `Iterable[T]`

Combining methods

- `zip(other : Iterable[U]) : Iterable[(T, U)]`
  Builds the collection of pairs taken element-wise
- `unzip`
  Reverse operation of `zip` and return a pair collection from a collection of pairs (i.e. T must be a couple of types)
- `zipWithIndex : Iterable[(T, Int)]`
  Builds the collection of pairs element/index

Current list is not exhaustive !

Refer to official API documentation

Particular collections

Particular collections

Option

`Option` can be treated as an `Iterable`

How so ?

`Option` can be treated as an `Iterable`

How so ?

- An `Option[T]` object can be seen as a collection of at most one element.

`Option` can be treated as an `Iterable`

How so ?

- An `Option[T]` object can be seen as a collection of at most one element.
- So `None` is seen as the empty collection.

`Option` can be treated as an `Iterable`

How so ?

- An `Option[T]` object can be seen as a collection of at most one element.
- So `None` is seen as the empty collection.

Result transmission made easier

`Option` can be treated as an `Iterable`

How so ?

- An `Option[T]` object can be seen as a collection of at most one element.
- So `None` is seen as the empty collection.

Result transmission made easier

- `o map f`
  $\equiv$ `o match { case Some(x) => Some(f(x)) ; case None => None }`

`Option` can be treated as an `Iterable`

How so ?

- An `Option[T]` object can be seen as a collection of at most one element.
- So `None` is seen as the empty collection.

Result transmission made easier

- `o map f`
  ≡ `o match { case Some(x) => Some(f(x)) ; case None => None }`
- `o flatMap f` with `f : T => Option[U]`
  ≡ `o match { case Some(x) => f(x) ; case None => None }`

`Option` can be treated as an `Iterable`

How so ?

- An `Option[T]` object can be seen as a collection of at most one element.
- So `None` is seen as the empty collection.

Result transmission made easier

- `o map f`
  ≡ `o match { case Some(x) => Some(f(x)) ; case None => None }`
- `o flatMap f` with `f : T => Option[U]`
  ≡ `o match { case Some(x) => f(x) ; case None => None }`
- `(o fold z)(op)`
  ≡ `o match { case Some(x) => op(x) ; case None => z }`

`Option` can be treated as an `Iterable`

How so ?

- An `Option[T]` object can be seen as a collection of at most one element.
- So `None` is seen as the empty collection.

Result transmission made easier

- `o map f`
  ≡ `o match { case Some(x) => Some(f(x)) ; case None => None }`
- `o flatMap f` with `f : T => Option[U]`
  ≡ `o match { case Some(x) => f(x) ; case None => None }`
- `(o fold z)(op)`
  ≡ `o match { case Some(x) => op(x) ; case None => z }`

NB: `o getOrElse z` ≡ `o match { case Some(x) => x ; case None => z }`

`Option` can be treated as an `Iterable`

How so ?

- An `Option[T]` object can be seen as a collection of at most one element.
- So `None` is seen as the empty collection.

Result transmission made easier

- `o map f`
  ≡ `o match { case Some(x) => Some(f(x)) ; case None => None }`
- `o flatMap f` with `f : T => Option[U]`
  ≡ `o match { case Some(x) => f(x) ; case None => None }`
- `(o fold z)(op)`
  ≡ `o match { case Some(x) => op(x) ; case None => z }`

NB: `o getOrElse z` ≡ `o match { case Some(x) => x ; case None => z }`

`Option`-valued collections

`Option` can be treated as an `Iterable`

How so ?

- An `Option[T]` object can be seen as a collection of at most one element.
- So `None` is seen as the empty collection.

Result transmission made easier

- `o map f`
  ≡ `o match { case Some(x) => Some(f(x)) ; case None => None }`
- `o flatMap f` with `f : T => Option[U]`
  ≡ `o match { case Some(x) => f(x) ; case None => None }`
- `(o fold z)(op)`
  ≡ `o match { case Some(x) => op(x) ; case None => z }`

NB: `o getOrElse z` ≡ `o match { case Some(x) => x ; case None => z }`

`Option`-valued collections

- `flatten` remove all `None`-valued elements.

Particular collections

LazyList

`LazyList[T]`

As name suggests, a lazy list has. . .

LazyList[T]

As name suggests, a lazy list has. . .

- head and tail evaluated lazily and once ;

`LazyList[T]`

As name suggests, a lazy list has. . .

- head and tail evaluated lazily and once ;
- *asking tail make head automatically evaluated*.

`LazyList[T]`

As name suggests, a lazy list has. . .

- head and tail evaluated lazily and once ;
- *asking tail make head automatically evaluated*.

So infinite collections are conceptually possible.

LazyList[T]

As name suggests, a lazy list has. . .

- head and tail evaluated lazily and once ;
- *asking tail make head automatically evaluated*.

So infinite collections are conceptually possible.

Example : list of **all** possible integers

Already defined as a method in class ***LazyList***

```
scala> def from(n : Int) : LazyList[Int] =
         {println(s"$n is computed") ; n} #:: from(n + 1)
def from(n: Int): LazyList[Int]
```

LazyList[T]

As name suggests, a lazy list has. . .

- head and tail evaluated lazily and once ;
- *asking tail make head automatically evaluated*.

So infinite collections are conceptually possible.

Example : list of **all** possible integers

Already defined as a method in class ***LazyList***

```
scala> def from(n : Int) : LazyList[Int] =
         {println(s"$n is computed") ; n} #:: from(n + 1)
def from(n: Int): LazyList[Int]

scala> val naturals = from(0)
naturals: LazyList[Int] = LazyList(<not computed>)
```

LazyList[T]

As name suggests, a lazy list has. . .

- head and tail evaluated lazily and once ;
- *asking tail make head automatically evaluated*.

So infinite collections are conceptually possible.

Example : list of **all** possible integers

Already defined as a method in class ***LazyList***

```
scala> def from(n : Int) : LazyList[Int] =
         {println(s"$n is computed") ; n} #:: from(n + 1)
def from(n: Int): LazyList[Int]

scala> val naturals = from(0)
naturals: LazyList[Int] = LazyList(<not computed>)
```

How is infinity handled by methods ?

- some methods cannot : `fold` and likes, `reduce` and likes, . . .
- some methods may : `exists`, `forall`, `find`, . . .
- some methods can : `filter`, `map`, `flatMap`, . . .

LazyList[T]

As name suggests, a lazy list has. . .

- head and tail evaluated lazily and once ;
- *asking tail make head automatically evaluated*.

So infinite collections are conceptually possible.

Example : list of **all** possible integers

Already defined as a method in class ***LazyList***

```
scala> def from(n : Int) : LazyList[Int] =
         {println(s"$n is computed") ; n} #:: from(n + 1)
def from(n: Int): LazyList[Int]

scala> val naturals = from(0)
naturals: LazyList[Int] = LazyList(<not computed>)
```

How is infinity handled by methods ?

- some methods cannot : `fold` and likes, `reduce` and likes, . . .
- some methods may : `exists`, `forall`, `find`, . . .
- some methods can : `filter`, `map`, `flatMap`, . . .

Expected behaviour is documented in API.

LazyList[T]

Examples

```
scala> val inverses = naturals.tail map { 1.0 / _ }
0 is computed
inverses: LazyList[Double] = LazyList(<not computed>)
```

LazyList[T]

Examples

```
scala> val inverses = naturals.tail map { 1.0 / _ }
0 is computed
inverses: LazyList[Double] = LazyList(<not computed>)

scala> inverses take 3
res0: LazyList[Double] = LazyList(<not computed>)
```

LazyList[T]

Examples

```
scala> val inverses = naturals.tail map { 1.0 / _ }
0 is computed
inverses: LazyList[Double] = LazyList(<not computed>)

scala> inverses take 3
res0: LazyList[Double] = LazyList(<not computed>)

scala> (inverses take 3).to(List)
1 is computed
2 is computed
3 is computed
res1: List[Double] = List(1.0, 0.5, 0.3333333333333333)
```

LazyList[T]

Examples

```
scala> val inverses = naturals.tail map { 1.0 / _ }
0 is computed
inverses: LazyList[Double] = LazyList(<not computed>)

scala> inverses take 3
res0: LazyList[Double] = LazyList(<not computed>)

scala> (inverses take 3).to(List)
1 is computed
2 is computed
3 is computed
res1: List[Double] = List(1.0, 0.5, 0.3333333333333333)

scala> naturals exists { _ > 0 }
res2: Boolean = true
```

LazyList[T]

Examples

```
scala> val inverses = naturals.tail map { 1.0 / _ }
0 is computed
inverses: LazyList[Double] = LazyList(<not computed>)

scala> inverses take 3
res0: LazyList[Double] = LazyList(<not computed>)

scala> (inverses take 3).to(List)
1 is computed
2 is computed
3 is computed
res1: List[Double] = List(1.0, 0.5, 0.3333333333333333)

scala> naturals exists { _ > 0 }
res2: Boolean = true

scala> naturals exists { _ < 0 }
4 is computed
5 is computed
... (endless !)
```

LazyList[T]

Erathostenes' sieve

To build the list of prime numbers from the list of natural numbers :

LazyList[T]

Erathostenes' sieve

To build the list of prime numbers from the list of natural numbers :

1. Start from $n = 2$.

LazyList[T]

Erathostenes' sieve

To build the list of prime numbers from the list of natural numbers :

1. Start from $n = 2$.
2. Remove any multiple of n.

LazyList[T]

Erathostenes' sieve

To build the list of prime numbers from the list of natural numbers :

1. Start from $n = 2$.
2. Remove any multiple of n.
3. Set n as next non-removed integer and go to step 2.

LazyList[T]

Erathostenes' sieve

To build the list of prime numbers from the list of natural numbers :

1. Start from $n = 2$.
2. Remove any multiple of n.
3. Set n as next non-removed integer and go to step 2.

```
scala> def sieve(l : LazyList[Int]) : LazyList[Int] =
         l.head #:: sieve(l.tail filterNot { _ % l.head == 0 })
sieve(l: LazyList[Int]): LazyList[Int]
```

LazyList[T]

Erathostenes' sieve

To build the list of prime numbers from the list of natural numbers :

1. Start from $n = 2$.
2. Remove any multiple of n.
3. Set n as next non-removed integer and go to step 2.

```
scala> def sieve(l : LazyList[Int]) : LazyList[Int] =
         l.head #:: sieve(l.tail filterNot { _ % l.head == 0 })
sieve(l: LazyList[Int]): LazyList[Int]

scala> val primes = sieve(LazyList.from(2))
primes: LazyList[Int] = LazyList(<not computed>)
```

LazyList[T]

Erathostenes' sieve

To build the list of prime numbers from the list of natural numbers :

1. Start from $n = 2$.
2. Remove any multiple of n.
3. Set n as next non-removed integer and go to step 2.

```
scala> def sieve(l : LazyList[Int]) : LazyList[Int] =
         l.head #:: sieve(l.tail filterNot { _ % l.head == 0 })
sieve(l: LazyList[Int]): LazyList[Int]

scala> val primes = sieve(LazyList.from(2))
primes: LazyList[Int] = LazyList(<not computed>)

scala> (primes take 10).to(List)
res4: List[Int] = List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

LazyList[T]

Erathostenes' sieve

To build the list of prime numbers from the list of natural numbers :

1. Start from $n = 2$.
2. Remove any multiple of n.
3. Set n as next non-removed integer and go to step 2.

```
scala> def sieve(l : LazyList[Int]) : LazyList[Int] =
         l.head #:: sieve(l.tail filterNot { _ % l.head == 0 })
sieve(l: LazyList[Int]): LazyList[Int]

scala> val primes = sieve(LazyList.from(2))
primes: LazyList[Int] = LazyList(<not computed>)

scala> (primes take 10).to(List)
res4: List[Int] = List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

Before version 2.12 : Stream[T]

LazyList[T]

Erathostenes' sieve

To build the list of prime numbers from the list of natural numbers :

1. Start from $n = 2$.
2. Remove any multiple of n.
3. Set n as next non-removed integer and go to step 2.

```
scala> def sieve(l : LazyList[Int]) : LazyList[Int] =
         l.head #:: sieve(l.tail filterNot { _ % l.head == 0 })
sieve(l: LazyList[Int]): LazyList[Int]

scala> val primes = sieve(LazyList.from(2))
primes: LazyList[Int] = LazyList(<not computed>)

scala> (primes take 10).to(List)
res4: List[Int] = List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

Before version 2.12 : Stream[T]

- Head was not lazily evaluated.

Parallel collections

Motivation

Parallel collections

Motivation

- Improve performances of computation

Parallel collections

Motivation

- Improve performances of computation
- Some computations may be done independently because they depend on independent elements.

Parallel collections

Motivation

- Improve performances of computation
- Some computations may be done independently because they depend on independent elements.
- Hence they may be performed in parallel on several threads.

Parallel collections

Motivation

- Improve performances of computation
- Some computations may be done independently because they depend on independent elements.
- Hence they may be performed in parallel on several threads.

How to use parallel collections ?

Parallel collections

Motivation

- Improve performances of computation
- Some computations may be done independently because they depend on independent elements.
- Hence they may be performed in parallel on several threads.

How to use parallel collections ?

- Before version 2.13 : package `scala.collection.parallel`

Parallel collections

Motivation

- Improve performances of computation
- Some computations may be done independently because they depend on independent elements.
- Hence they may be performed in parallel on several threads.

How to use parallel collections ?

- Before version 2.13 : package `scala.collection.parallel`
- From version 2.13 : separate module `scala-parallel-collections`
  Dependency to add in `build.sbt` (package name preserved)

Parallel collections

Motivation

- Improve performances of computation
- Some computations may be done independently because they depend on independent elements.
- Hence they may be performed in parallel on several threads.

How to use parallel collections ?

- Before version 2.13 : package `scala.collection.parallel`
- From version 2.13 : separate module `scala-parallel-collections`
  Dependency to add in `build.sbt` (package name preserved)

Make standard collections parallelizable : `.par`

- Version 2.13 requires specific import :

```
import scala.collection.parallel.CollectionConverters._
```

Parallel collections

Sequential vs parallel collections

```
import scala.collection.parallel.CollectionConverters._

object Parallel extends App
  {
    def elapsedNano[T](chunk : => T, times : Int = 1) : Long =
      {
        val start = System.nanoTime() // from Java
        for (i <- 0 until times) { val res = chunk }
        System.nanoTime() - start
      }

    val c = 0 to 1000000
    println(s"${elapsedNano(c     count { _ %3 == 0 }, 1000)/1000000.0} ms")
    println(s"${elapsedNano(c.par count { _ %3 == 0 }, 1000)/1000000.0} ms")
  }
```

Parallel collections and (im)mutability

Mutability implies state management

- Shared variable needs explicit lock to ensure correctness.

Parallel collections and (im)mutability

Mutability implies state management

- Shared variable needs explicit lock to ensure correctness.

Immutability implies *referential transparency*

Parallel collections and (im)mutability

Mutability implies state management

- Shared variable needs explicit lock to ensure correctness.

Immutability implies *referential transparency*

- Think “purity” : same input always yield same results

Parallel collections and (im)mutability

Mutability implies state management

- Shared variable needs explicit lock to ensure correctness.

Immutability implies *referential transparency*

- Think “purity” : same input always yield same results
- Applicable to `val` but not `var`

Parallel collections and (im)mutability

Mutability implies state management

- Shared variable needs explicit lock to ensure correctness.

Immutability implies *referential transparency*

- Think “purity” : same input always yield same results
- Applicable to **`val`** but not **`var`**

$\Rightarrow$ No need for explicit lock

Parallel collections and (im)mutability

Mutability implies state management

- Shared variable needs explicit lock to ensure correctness.

Immutability implies *referential transparency*

- Think “purity” : same input always yield same results
- Applicable to `val` but not `var`

$\Rightarrow$ No need for explicit lock

Immutability is highly recommended for parallel computations

Parallel collections and (im)mutability

Mutability implies state management

- Shared variable needs explicit lock to ensure correctness.

Immutability implies *referential transparency*

- Think “purity” : same input always yield same results
- Applicable to `val` but not `var`

$\Rightarrow$ No need for explicit lock

Immutability is highly recommended for parallel computations

(At least, use mutable state as rarely as possible. . . !)

Parallel collections and (im)mutability

Immutability vs mutable state

```
import scala.collection.parallel._
import scala.collection.parallel.CollectionConverters._

object MutableIsEvil extends App
  {
    val c = 1 to 1000000
    def isPrime(n : Int) = (2 to math.sqrt(n).toInt) forall { n % _ != 0 }

    // Immutable case
    for (i <- 1 to 3) println(s"count isPrime = ${c.par count isPrime}")

    // Mutable case with a counter
    def countPrimes(c : ParSeq[Int]) = {
        var count = 0
        c foreach { n => if (isPrime(n)) count = count + 1 }
        count
      }
    for (i <- 1 to 3) println(s"countPrimes = ${countPrimes(c.par)}")
  }
```