**18.4. Metody kolekcji**

Cechy Seq, Map oraz Set dziedziczą pewne wspólne cechy, a wraz z nimi dziedziczą również metody w nich zadeklarowane, co sprawia, że istnieje wiele wspólnych metod, które mogą być wykonywane na różnych rodzajach kolekcji. Istnieją również metody specyficzne dla pewnych rodzajów kolekcji. W tym podrozdziale zaprezentowanych jest wiele (ale z pewnością nie wszystkie) z metod dostępnych w kolekcjach.

Metoda foreach wykonuje przekazaną jej w argumencie funkcję na kolejnych elementach przetwarzanej kolekcji. Użycie metody foreach ilustrują następujące przykłady.

scala> **Seq('a','b','c').foreach(println)**

a

b

c

scala> **Map('a'->1,'b'->2).foreach(println)**

(a,1)

(b,2)

scala> **Set('a','b','c').foreach(println)**

a

b

c

Metody isEmpty oraz nonEmpty pozwalają sprawdzić, czy kolekcja jest pusta. Jeśli tak jest, to metoda isEmpty zwraca wartość true, a metoda nonEmpty zwraca wartość false. W przeciwnym przypadku metoda isEmpty zwraca wartość false, a metoda nonEmpty zwraca wartość true.

scala> **List().isEmpty**

res3: Boolean = true

scala> **List(1,2,3).isEmpty**

res4: Boolean = false

scala> **List().nonEmpty**

res5: Boolean = false

scala> **List(1,2,3).nonEmpty**

res6: Boolean = true

scala> **Map().isEmpty**

res7: Boolean = true

scala> **Set(1,2,3).nonEmpty**

res8: Boolean = true

Metoda size zwraca rozmiar kolekcji.

scala> **Set(1).size**

res9: Int = 1

scala> **Seq(1,2,3,4).size**

res10: Int = 4

scala> **Map('a->1,'b->2).size**

res11: Int = 2

Metoda forall zwraca wartość logiczną określającą, czy wszystkie elementy kolekcji spełniają podany predykat logiczny (funkcję zwracającą wartość logiczną).

scala> **Map("abc"->3,"ab"->2,"qw"->2,"w"->1).forall(p => p.\_1.length == p.\_2)**

res12: Boolean = true

scala> **Seq(1,2,3,4,5).forall(\_ > 0)**

res13: Boolean = true

scala> **Set(1,2,-3,4,5).forall(\_ > 0)**

res14: Boolean = false

Metoda exists zwraca wartość logiczną określającą, czy istnieje przynajmniej jeden element kolekcji, który spełnia podany predykat logiczny.

scala> **Map("abc"->3,"ab"->2,"qw"->2,"w"->1).exists(p => p.\_1.length != p.\_2)**

res15: Boolean = false

scala> **Seq(1,2,3,4,5).exists(\_ < 0)**

res16: Boolean = false

scala> **Set(1,2,-3,4,5).exists(\_ < 0)**

res17: Boolean = true

Metoda find szuka w kolekcji elementu spełniającego podaną funkcję logiczną i zwraca pierwszy taki znaleziony element opakowany w obiekt Some lub zwraca wartość None, jeśli nie znajdzie żadnego takiego elementu.

scala> **Map("abc"->3,"ab"->2,"qw"->2,"w"->1).find(p => p.\_1.length != p.\_2)**

res18: Option[(String, Int)] = None

scala> **Seq(1,2,3,4,5).find(\_ > 3)**

res19: Option[Int] = Some(4)

scala> **Set(1,2,-5,4,-3).find(\_ < 0)**

res20: Option[Int] = Some(-3)

Metoda count zwraca liczbę elementów kolekcji spełniających podany w argumencie tej metody predykat logiczny.

scala> **List(1,2,3,4,5,6,7).count(\_ % 2 == 0)**

res21: Int = 3

scala> **Set("a","bc","qwe","qw").count(\_.length == 2)**

res22: Int = 2

Metoda filter zwraca kolekcję składającą się z tych elementów, dla których podany predykat przyjmuje wartość true. Metoda filterNot zwraca kolekcję składającą się z tych elementów, dla których podany predykat przyjmuje wartość false.

scala> **List(1,2,3,4,5,6,7,8).filter(\_ % 2 == 0)**

res23: List[Int] = List(2, 4, 6, 8)

scala> **Set("a","abc","abcde","abcd").filterNot(\_.size >= 4)**

res24: scala.collection.immutable.Set[String] = Set(a, abc)

scala> **Map(1->3,2->2,3->2,5->5).filter{ case (k,v) => k == v }**

res25: scala.collection.immutable.Map[Int,Int] = Map(2 -> 2, 5 -> 5)

Metoda partition zwraca w dwuelementowej krotce dwie kolekcje. Pierwsza zawiera elementy, dla których predykat zwrócił wartość true, a druga elementy, dla których predykat zwrócił wartość false.

scala> **Seq(1,-2,3,-4,5).partition(\_ > 0)**

res26: (Seq[Int], Seq[Int]) = (List(1, 3, 5),List(-2, -4))

Metoda filterKeys działa na mapach. Zwraca mapę zawierającą tylko te pary z pierwotnej mapy, których klucze spełniają warunek logiczny.

scala> **Map("a"->3,"B"->2,"c"->1) filterKeys (k => k == k.toLowerCase)**

res27: scala.collection.immutable.Map[String,Int] = Map(a -> 3, c -> 1)

Metoda foldLeft ma dwa parametry: wartość początkową oraz dwuargumentową funkcję. W przypadku wykonania jej na kolekcji pustej, metoda zwraca wartość początkową. W przypadku kolekcji jednoelementowej wynikiem jest rezultat ewaluacji funkcji na wartości początkowej i na elemencie kolekcji. W przypadku kolekcji z większą liczbą elementów wynik całej metody jest uzyskiwany poprzez wykonywanie przekazanej funkcji najpierw na wartości początkowej i pierwszym elemencie, a następnie kolejno na wyniku wykonania funkcji oraz kolejnych elementach kolekcji. Wynik ostatniej z takich operacji staje się wynikiem całej metody. Poniższe przykłady pokazują operacje sumujące elementy kolekcji.

scala> **List[Int]().foldLeft(0)(\_ + \_)**

res28: Int = 0

scala> **List(1).foldLeft(0)(\_ + \_)**

res29: Int = 1

scala> **List(1,2).foldLeft(0)(\_ + \_)**

res30: Int = 3

scala> **List(1,2,3).foldLeft(0)(\_ + \_)**

res31: Int = 6

scala> **List(1,2,3,4).foldLeft(0)(\_ + \_)**

res32: Int = 10

Metoda foldRight jest podobna do foldLeft, ale elementy na których wykonywane są kolejne operacje nie są brane od początku, a od końca kolekcji.

scala> **List(1,2,3,4).foldRight(0)(\_ + \_)**

res33: Int = 10

scala> **List("a","b","c","d").foldLeft("\*")(\_ + \_)**

res34: String = \*abcd

scala> **List("a","b","c","d").foldRight("\*")(\_ + \_)**

res35: String = abcd\*

scala> **List("a","b","c","d").foldLeft("\*")((acc,elem) => acc + elem\*acc.length)**

res36: String = \*abbccccdddddddd

scala> **List("a","b","c","d").foldRight("\*")((elem,acc) => elem\*acc.length + acc)**

res37: String = aaaaaaaabbbbccd\*

Metody min oraz max zwracają — odpowiednio — najmniejszy i największy element kolekcji. Metody minBy oraz maxBy zwracają element, który po przekształceniu przez podaną funkcję jest odpowiednio najmniejszy i największy.

scala> **Seq(1,-4,3,-6,4).min**

res38: Int = -6

scala> **Seq(1,-4,3,-6,4).max**

res39: Int = 4

scala> **Map(1->1, 0->3).max**

res40: (Int, Int) = (1,1)

scala> **Seq(1,-4,3,-6,4).minBy(n => n\*n)**

res41: Int = 1

scala> **Seq(1,-4,3,-6,4).maxBy(n => n\*n)**

res42: Int = -6

Metody mkString służą do utworzenia tekstowych reprezentacji kolekcji. Jedna z wersji tej metody przyjmuje trzy parametry, reprezentujące odpowiednio prefiks, separator elementów oraz sufiks tekstowej reprezentacji. Inna wersja ma jeden parametr, oznaczający separator elementów, a prefiks i sufiks uznaje za puste. Wreszcie istnieje też bezparametrowa wersja, która łączy tekstowe reprezentacje elementów bez rozdzielania ich separatorem oraz bez poprzedzania prefiksem i sufiksem.

scala> **Seq(1,2,3,4).mkString("[",",","]")**

res43: String = [1,2,3,4]

scala> **Seq(1,2,3,4).mkString(",")**

res44: String = 1,2,3,4

scala> **Seq(1,2,3,4).mkString**

res45: String = 1234

Metoda head zwraca pierwszy element niepustej kolekcji. W przypadku pustej kolekcji metoda zwraca wyjątek.

scala> **List(4,5,6).head**

res46: Int = 4

scala> **Map(3->'b',1->'a').head**

res47: (Int, Char) = (3,b)

scala> **Set('a','b',2,3).head**

res48: Int = 97

scala> **Stream(2).head**

res49: Int = 2

scala> **List().head**

java.util.NoSuchElementException: head of empty list

at scala.collection.immutable.Nil$.head(List.scala:420)

at scala.collection.immutable.Nil$.head(List.scala:417)

... 33 elided

Metoda tail zwraca pozostałą część niepustej kolekcji, niezawierającą pierwszego elementu. W przypadku pustej kolekcji metoda zwraca wyjątek.

scala> **List(4,5,6).tail**

res51: List[Int] = List(5, 6)

scala> **Map(3->'b',1->'a').tail**

res52: scala.collection.immutable.Map[Int,Char] = Map(1 -> a)

scala> **Set('a','b',2,3).tail**

res53: scala.collection.immutable.Set[Int] = Set(98, 2, 3)

scala> **Stream(2).tail**

res54: scala.collection.immutable.Stream[Int] = Stream()

scala> **List().tail**

java.lang.UnsupportedOperationException: tail of empty list

at scala.collection.immutable.Nil$.tail(List.scala:422)

at scala.collection.immutable.Nil$.tail(List.scala:417)

... 33 elided

Metoda last zwraca ostatni element niepustej kolekcji. W przypadku pustej kolekcji metoda zwraca wyjątek.

scala> **List(4,5,6).last**

res56: Int = 6

scala> **Map(3->'b',1->'a').last**

res57: (Int, Char) = (1,a)

scala> **Set('a','b',2,3).last**

res58: Int = 3

scala> **Stream(2).last**

res59: Int = 2

scala> **Map().last**

java.util.NoSuchElementException: next on empty iterator

at scala.collection.Iterator$$anon$2.next(Iterator.scala:39)

at scala.collection.Iterator$$anon$2.next(Iterator.scala:37)

at scala.collection.IterableLike$class.head(IterableLike.scala:107)

at scala.collection.AbstractIterable.head(Iterable.scala:54)

at scala.collection.TraversableLike$class.last(TraversableLike.scala:459)

at scala.collection.AbstractTraversable.last(Traversable.scala:104)

... 33 elided

Metoda init zwraca pozostałą część niepustej kolekcji, niezawierającą ostatniego elementu. W przypadku pustej kolekcji metoda zwraca wyjątek.

scala> **List(4,5,6).init**

res61: List[Int] = List(4, 5)

scala> **Map(3->'b',1->'a').init**

res62: scala.collection.immutable.Map[Int,Char] = Map(3 -> b)

scala> **Set('a','b',2,3).init**

res63: scala.collection.immutable.Set[Int] = Set(97, 98, 2)

scala> **Stream(2).init**

res64: scala.collection.immutable.Stream[Int] = Stream()

scala> **Set().init**

java.lang.UnsupportedOperationException: empty.init

at scala.collection.TraversableLike$class.init(TraversableLike.scala:479)

at scala.collection.AbstractTraversable.init(Traversable.scala:104)

... 33 elided

Metody ++ oraz ++: łączą ze sobą dwie kolekcje. W przypadku metody ++ lewy operand, a w przypadku metody ++: prawy operand, wpływa na typ kolekcji wynikowej.

scala> **Set(1,2,3,4) ++ Seq(9,8,7)**

res66: scala.collection.immutable.Set[Int] = Set(1, 9, 2, 7, 3, 8, 4)

scala> **Set(1,2,3,4) ++: Seq(9,8,7)**

res67: Seq[Int] = List(1, 2, 3, 4, 9, 8, 7)

Metody take i drop zwracają kolekcję — odpowiednio — zawierającą żądaną liczbę początkowych elementów lub pomijającą taką liczbę początkowych elementów. W przypadku kolekcji krótszych, niż podana liczba elementów, metoda take zwraca całą kolekcję, a metoda drop zwraca kolekcję pustą.

scala> **List(1,2,3,4,5).take(2)**

res68: List[Int] = List(1, 2)

scala> **List(1,2,3,4,5).drop(2)**

res69: List[Int] = List(3, 4, 5)

scala> **Set(1).take(2)**

res70: scala.collection.immutable.Set[Int] = Set(1)

scala> **Set(1).drop(2)**

res71: scala.collection.immutable.Set[Int] = Set()

Metoda splitAt zwraca dwie kolekcje (w dwuelementowej krotce). Pierwsza kolekcja odpowiada wynikowi wykonania metody take, a druga wykonania metody drop, z takim samym argumentem, jak argument metody splitAt.

scala> **List(1,2,3,4,5).splitAt(2)**

res72: (List[Int], List[Int]) = (List(1, 2),List(3, 4, 5))

scala> **Set(1).splitAt(2)**

res73: (scala.collection.immutable.Set[Int], scala.collection.immutable.Set[Int]) = (Set(1),Set())

Metoda slice przyjmuje dwa argumenty liczbowe i zwraca te elementy kolekcji, których indeksy są większe lub równe pierwszemu z argumentów oraz mniejsze od drugiego z argumentów.

scala> **Seq('a','b','c','d').slice(1,3)**

res74: Seq[Char] = List(b, c)

scala> **Seq('a','b','c','d').slice(2,4)**

res75: Seq[Char] = List(c, d)

scala> **Seq('a','b','c','d').slice(2,3)**

res76: Seq[Char] = List(c)

scala> **Set('a','b','c','d').slice(1,3)**

res77: scala.collection.immutable.Set[Char] = Set(b, c)

scala> **Map(1->2,2->3).slice(1,3)**

res78: scala.collection.immutable.Map[Int,Int] = Map(2 -> 3)

Metoda takeWhile zwraca tyle kolejnych elementów z początku kolekcji, ile spełnia podany w argumencie tej metody predykat logiczny. Metoda dropWhile zwraca pozostałe elementy kolekcji, oprócz tych elementów początkowych, które spełniają predykat.

scala> **List(1,2,3,4,5,1,2,3,4).takeWhile(\_ < 4)**

res79: List[Int] = List(1, 2, 3)

scala> **List(1,2,3,4,5,1,2,3,4).dropWhile(\_ < 4)**

res80: List[Int] = List(4, 5, 1, 2, 3, 4)

scala> **Set(1).takeWhile(\_ > 0)**

res81: scala.collection.immutable.Set[Int] = Set(1)

scala> **Set(1).dropWhile(\_ > 0)**

res82: scala.collection.immutable.Set[Int] = Set()

Metoda span zwraca dwie kolekcje (w dwuelementowej krotce). Pierwsza kolekcja odpowiada wynikowi wykonania metody takeWhile, a druga wykonania metody dropWhile, z takim samym argumentem, jak argument metody span.

scala> **List(1,2,3,4,5,1,2,3,4).span(\_ < 4)**

res83: (List[Int], List[Int]) = (List(1, 2, 3),List(4, 5, 1, 2, 3, 4))

scala> **Set(1).span(\_ > 0)**

res84: (scala.collection.immutable.Set[Int], scala.collection.immutable.Set[Int]) = (Set(1),Set())

Metoda map wykonuje podaną w jej argumencie operację na elementach kolekcji i zwraca kolekcję składającą się z wyników tych operacji.

scala> **Seq(1,2,3).map(\_ + 1)**

res85: Seq[Int] = List(2, 3, 4)

scala> **Set(1,2).map(\_.toString)**

res86: scala.collection.immutable.Set[String] = Set(1, 2)

scala> **Map(1->"a",2->"b").map(pair => pair.\_1.toString + pair.\_2)**

res87: scala.collection.immutable.Iterable[String] = List(1a, 2b)

Metoda mapValues działa na mapach. Zwraca mapę składającą się z par zawierających klucze z oryginalnej kolekcji oraz wyniki wykonania podanej funkcji na wartościach z oryginalnej kolekcji.

scala> **Map("a"->3,"B"->2) mapValues (\_ \* 2)**

res88: scala.collection.immutable.Map[String,Int] = Map(a -> 6, B -> 4)

Metoda flatMap wykonuje podaną w jej argumencie operację na elementach kolekcji i zwraca kolekcję składającą się z połączonych wyników tych operacji.

scala> **List(1,2,3).flatMap(a => List(a,a))**

res89: List[Int] = List(1, 1, 2, 2, 3, 3)

Metoda groupBy grupuje elementy kolekcji według rezultatu podanej w argumencie operacji działającej na elementach kolekcji i zwraca mapę zawierającą pogrupowane wartości. Klucze w mapie odpowiadają różnym wynikom operacji. Wartościami mapy są kolekcje grupujące te elementy wejściowej kolekcji, dla których operacja dała wynik równy wartości klucza.

scala> **Seq(1,2,3,4,5,6,7,8,9,10).groupBy(\_ % 3)**

res90: scala.collection.immutable.Map[Int,Seq[Int]] = Map(2 -> List(2, 5, 8), 1 -> List(1, 4, 7, 10), 0 -> List(3, 6, 9))

scala> **Set("a","b","cd","efg","qwe").groupBy(\_.size)**

res91: scala.collection.immutable.Map[Int,scala.collection.immutable.Set[String]] = Map(2 -> Set(cd), 1 -> Set(a, b), 3 -> Set(efg, qwe))

Metoda view tworzy nieścisłą wersję kolekcji. Wszystkie elementy, które zawiera ścisła kolekcja, są wyliczone zanim kolekcja może być użyta. Natomiast kolekcja nieścisła może być używana nawet, jeśli nie wszystkie jej elementy zostały obliczone. Metoda force służy do wymuszenia obliczenia elementów kolekcji.

scala> **Seq("a","bc","def","d","e","ef","f").view.filter(\_.size == 1)**

res92: scala.collection.SeqView[String,Seq[String]] = SeqViewF(...)

scala> **Seq("a","bc","def","d","e","ef","f").view.filter(\_.length == 1).force**

res93: Seq[String] = List(a, d, e, f)

Metoda withFilter tworzy nieścisłą wersję kolekcji ze zdefiniowanym za pomocą podanej funkcji logicznej filtrem. Na takiej filtrowanej wersji kolekcji można wywoływać kolejne metody withFilter, a także metody map, flatMap i foreach.

scala> **Seq(1,2,3,4,5,6,7,8,9,10).withFilter(\_ % 2 == 0).foreach(println)**

2

4

6

8

10

scala> **Seq(1,2,3,4,5,6,7,8,9,10).withFilter(\_ % 2 == 0).map(\_ \* 2)**

res95: Seq[Int] = List(4, 8, 12, 16, 20)

scala> **val collection = Seq(1,2,3,4,5,6,7,8,9,10).withFilter(\_ % 2 == 0).**

| withFilter(\_ > 7)

collection: scala.collection.generic.FilterMonadic[Int,Seq[Int]] = scala.collection.TraversableLike$WithFilter@f31a95

scala> **collection.foreach(println)**

8

10

scala> **collection.map(\_ \* 2)**

res97: Seq[Int] = List(16, 20)

scala> **collection.flatMap(x => Seq(x,x))**

res98: Seq[Int] = List(8, 8, 10, 10)

Metoda iterator zwraca obiekt zwany iteratorem, który pozwala odwiedzić kolejno elementy kolekcji. Iterator posiada metody next i hasNext. Metoda next zwraca kolejny element kolekcji lub kończy się wyjątkiem, jeśli nie ma już więcej dostępnych elementów. Metoda hasNext zwraca wartość logiczną informującą o tym, czy są jeszcze elementy dostępne do zwrócenia przez wywołania metody next.

scala> **val iter = Seq(5,4,3,2,1).iterator**

iter: Iterator[Int] = non-empty iterator

scala> **iter.hasNext**

res99: Boolean = true

scala> **iter.next**

res100: Int = 5

scala> **iter.next**

res101: Int = 4

scala> **iter.next**

res102: Int = 3

scala> **iter.next**

res103: Int = 2

scala> **iter.next**

res104: Int = 1

scala> **iter.hasNext**

res105: Boolean = false

scala> **iter.next**

java.util.NoSuchElementException: next on empty iterator

at scala.collection.Iterator$$anon$2.next(Iterator.scala:39)

at scala.collection.Iterator$$anon$2.next(Iterator.scala:37)

at scala.collection.LinearSeqLike$$anon$1.next(LinearSeqLike.scala:47)

... 33 elided

scala> **iter.hasNext**

res107: Boolean = false

Metoda grouped tworzy iterator udostępniający elementy kolekcji w grupach o określonym rozmiarze. Ostatnia z grup może mieć mniej elementów niż pozostałe.

scala> **Seq(1,2,3,4,5).grouped(2).toList**

res108: List[Seq[Int]] = List(List(1, 2), List(3, 4), List(5))

Metoda sliding również tworzy iterator udostępniający elementy kolekcji w grupach o określonym rozmiarze. Pierwszy parametr metody określa rozmiar grupy. Drugi parametr, który posiada argument domyślny równy 1, określa liczbę elementów, o którą przesuwa się początek kolejnej grupy w stosunku do początku poprzedniej. Ostatnia z grup może mieć mniej elementów niż pozostałe.

scala> **Seq(1,2,3,4,5,6).sliding(3,2).toList**

res109: List[Seq[Int]] = List(List(1, 2, 3), List(3, 4, 5), List(5, 6))

scala> **Seq(1,2,3,4,5,6).sliding(2,4).toList**

res110: List[Seq[Int]] = List(List(1, 2), List(5, 6))

scala> **Seq(1,2,3,4,5).sliding(2).toList**

res111: List[Seq[Int]] = List(List(1, 2), List(2, 3), List(3, 4), List(4, 5))

Metoda zip tworzy na podstawie elementów kolekcji, na której jest wywołana oraz innej kolekcji, podanej jako argument, nową kolekcję, której elementami są pary (krotki dwuelementowe) składające się z kolejnych elementów obu kolekcji. Długość utworzonej kolekcji jest równa długości krótszej z kolekcji wejściowych.

scala> **Seq(1,2,3,4,5).zip(Seq('a','b','c','d'))**

res112: Seq[(Int, Char)] = List((1,a), (2,b), (3,c), (4,d))

Metoda zipAll działa podobnie do metody zip, z tym że utworzona kolekcja krotek dwuelementowych ma długość równą długości dłuższej z kolekcji wejściowych. Brakujące elementy krótszej kolekcji są zastępowane w kolekcji wynikowej jedną z wartości podanych jako argumenty metody.

scala> **Seq(1,2,3,4,5).zipAll(Seq('a','b','c','d'),9,'z')**

res113: Seq[(Int, Char)] = List((1,a), (2,b), (3,c), (4,d), (5,z))

scala> **Seq(1,2,3).zipAll(Seq('a','b','c','d'),9,'z')**

res114: Seq[(Int, Char)] = List((1,a), (2,b), (3,c), (9,d))

Metoda zipWithIndex tworzy nową kolekcję, której elementami są pary (krotki dwuelementowe) składające się z kolejnych elementów kolekcji oraz indeksów określających ich pozycje w kolekcji.

scala> **Seq('a','b','c','d').zipWithIndex**

res115: Seq[(Char, Int)] = List((a,0), (b,1), (c,2), (d,3))

scala> **Set('a','b','c','d').zipWithIndex**

res116: scala.collection.immutable.Set[(Char, Int)] = Set((a,0), (b,1), (c,2), (d,3))

Metoda contains sprawdza, czy dany element należy do kolekcji. W przypadku mapy sprawdzane są klucze.

scala> **Set('a','b','c').contains('b')**

res117: Boolean = true

scala> **Set('a','b','c').contains('d')**

res118: Boolean = false

scala> **Seq(1,2,3,4).contains(2)**

res119: Boolean = true

scala> **Map('a->1,'b->2).contains('a)**

res120: Boolean = true

Metoda reverse zwraca sekwencję z odwróconą kolejnością elementów.

scala> **Seq(1,2,3,4).reverse**

res121: Seq[Int] = List(4, 3, 2, 1)

Metoda startsWith sprawdza, czy sekwencja zaczyna się od podanej w argumencie sekwencji, a metoda endsWith sprawdza, czy się taką sekwencją kończy. Dwuargumentowa wersja metody startsWith sprawdza, czy podana w pierwszym argumencie sekwencja jest fragmentem sekwencji od pozycji podanej jako drugi argument.

scala> **Seq("a","B","CD","F","GH").startsWith(Seq("a","B"))**

res122: Boolean = true

scala> **Seq("a","B","CD","F","GH").startsWith(Seq("CD","F"))**

res123: Boolean = false

scala> **Seq("a","B","CD","F","GH").startsWith(Seq("CD","F"),1)**

res124: Boolean = false

scala> **Seq("a","B","CD","F","GH").startsWith(Seq("CD","F"),2)**

res125: Boolean = true

scala> **Seq("a","B","CD","F","GH").endsWith(Seq("CD","F"))**

res126: Boolean = false

scala> **Seq("a","B","CD","F","GH").endsWith(Seq("CD","F","GH"))**

res127: Boolean = true

Metody get i getOrElse mogą być użyte do uzyskania z mapy wartości odpowiadającej podanemu kluczowi. Metoda get zwraca opcję zawierającą znalezioną wartość lub opcję pustą (None) jeśli wartość odpowiadająca kluczowi nie została znaleziona. Metoda getOrElse zwraca wartość odpowiadającą podanemu kluczowi, jeśli taki klucz istnieje w mapie, a wartość domyślną w przeciwnym przypadku.

scala> **Map("a"->3,"b"->2).get("b")**

res128: Option[Int] = Some(2)

scala> **Map("a"->3,"b"->2).get("x")**

res129: Option[Int] = None

scala> **Map("a"->3,"b"->2).getOrElse("b",99)**

res130: Int = 2

scala> **Map("a"->3,"b"->2).getOrElse("x",99)**

res131: Int = 99

Metody +: i :+ dodają element do sekwencji, odpowiednio, na początku i na końcu.

scala> **"x" +: Seq("a","b","c","d")**

res132: Seq[String] = List(x, a, b, c, d)

scala> **Seq("a","b","c","d") :+ "x"**

res133: Seq[String] = List(a, b, c, d, x)

Metody + i - działają na zbiorach i mapach. Metoda dodaje + element do zbioru bądź parę do mapy. Metoda + występuje też w wersjach akceptujących w krotce więcej niż jeden element/parę. Metoda - usuwa element ze zbioru lub usuwa z mapy parę o podanym kluczu.

scala> **Set('a','b','c') + 'c'**

res134: scala.collection.immutable.Set[Char] = Set(a, b, c)

scala> **Set('a','b','c') + 'd'**

res135: scala.collection.immutable.Set[Char] = Set(a, b, c, d)

scala> **Set('a','b','c') - 'b'**

res136: scala.collection.immutable.Set[Char] = Set(a, c)

scala> **Set('a','b','c') - 'd'**

res137: scala.collection.immutable.Set[Char] = Set(a, b, c)

scala> **Map("a"->3,"b"->2) + ("b"->5)**

res138: scala.collection.immutable.Map[String,Int] = Map(a -> 3, b -> 5)

scala> **Map("a"->3,"b"->2) + ("c"->6)**

res139: scala.collection.immutable.Map[String,Int] = Map(a -> 3, b -> 2, c -> 6)

scala> **Map("a"->3,"b"->2) + (("b",5),("c",6))**

res140: scala.collection.immutable.Map[String,Int] = Map(a -> 3, b -> 5, c -> 6)

scala> **Set('a','b','c','d') + ('e','f','g')**

res141: scala.collection.immutable.Set[Char] = Set(e, f, a, b, g, c, d)

Metoda union dodaje do siebie dwie sekwencje. Metody union i | tworzą zbiór będący sumą dwóch zbiorów.

scala> **Seq(1,2,3,4).union(Seq(2,3,4,5))**

res142: Seq[Int] = List(1, 2, 3, 4, 2, 3, 4, 5)

scala> **Set('a','b','c') union Set('d','c','b')**

res143: scala.collection.immutable.Set[Char] = Set(a, b, c, d)

scala> **Set('a','b','c') | Set('d','c','b')**

res144: scala.collection.immutable.Set[Char] = Set(a, b, c, d)

Metoda intersect oblicza część wspólną dwóch sekwencji. Metody intersect i & tworzą zbiór będący iloczynem dwóch zbiorów.

scala> **Seq(1,2,3,4).intersect(Seq(2,3,4,5))**

res145: Seq[Int] = List(2, 3, 4)

scala> **Seq(1,2,3,4).intersect(Seq(5,4,3,2))**

res146: Seq[Int] = List(2, 3, 4)

scala> **Set('a','b','c') intersect Set('d','c','b')**

res147: scala.collection.immutable.Set[Char] = Set(b, c)

scala> **Set('a','b','c') & Set('d','c','b')**

res148: scala.collection.immutable.Set[Char] = Set(b, c)

Metoda diff oblicza różnicę między elementami sekwencji. Metody diff i &~ tworzą zbiór będący różnicą dwóch zbiorów.

scala> **Seq(1,2,3,4).diff(Seq(2,3,4,5))**

res149: Seq[Int] = List(1)

scala> **Set('a','b','c') diff Set('d','c','b')**

res150: scala.collection.immutable.Set[Char] = Set(a)

scala> **Set('a','b','c') &~ Set('d','c','b')**

res151: scala.collection.immutable.Set[Char] = Set(a)

Metoda subsetOf sprawdza, czy jeden ze zbiorów jest podzbiorem drugiego.

scala> **Set('a','b','c') subsetOf Set('d','c','b')**

res152: Boolean = false

scala> **Set('a','b','c') subsetOf Set('c','b')**

res153: Boolean = false

scala> **Set('c','b') subsetOf Set('a','b','c')**

res154: Boolean = true

Metoda distinct tworzy sekwencję pozbawioną duplikatów.

scala> **Seq(1,2,4,1,3,4,3).distinct**

res155: Seq[Int] = List(1, 2, 4, 3)

Metoda sorted sortuje sekwencję. Metoda sortBy sortuje sekwencję według wartości podanej funkcji. Metoda sortWith sortuje sekwencję zgodnie z podaną funkcją porównującą dwa elementy sekwencji.

scala> **Seq("a","A","b","d","B","c").sorted**

res156: Seq[String] = List(A, B, a, b, c, d)

scala> **Seq("a","A","b","d","B","c").sortBy(\_.toUpperCase)**

res157: Seq[String] = List(a, A, b, B, c, d)

scala> **Seq("a","A","b","d","B","c").sortWith(\_ < \_)**

res158: Seq[String] = List(A, B, a, b, c, d)

scala> **Seq("a","A","b","d","B","c").sortWith(\_ > \_)**

res159: Seq[String] = List(d, c, b, a, B, A)

Metoda keySet zwraca zbiór zawierający wszystkie klucze z mapy, a metoda keys zwraca obiekt typu Iterable zawierający takie klucze. Metoda keysIterator zwraca iterator po wszystkich kluczach mapy.

scala> **Map("a"->3,"b"->2,"c"->1).keySet**

res160: scala.collection.immutable.Set[String] = Set(a, b, c)

scala> **Map("a"->3,"b"->2,"c"->1).keys**

res161: Iterable[String] = Set(a, b, c)

scala> **Map("a"->3,"b"->2,"c"->1).keysIterator.toList**

res162: List[String] = List(a, b, c)

Metoda values zwraca obiekt typu Iterable, zawierający wszystkie wartości z mapy. Metoda valuesIterator zwraca iterator po wszystkich wartościach mapy.

scala> **Map("a"->3,"b"->2,"c"->1).values**

res163: Iterable[Int] = MapLike(3, 2, 1)

scala> **Map("a"->3,"b"->2,"c"->1).valuesIterator.toList**

res164: List[Int] = List(3, 2, 1)