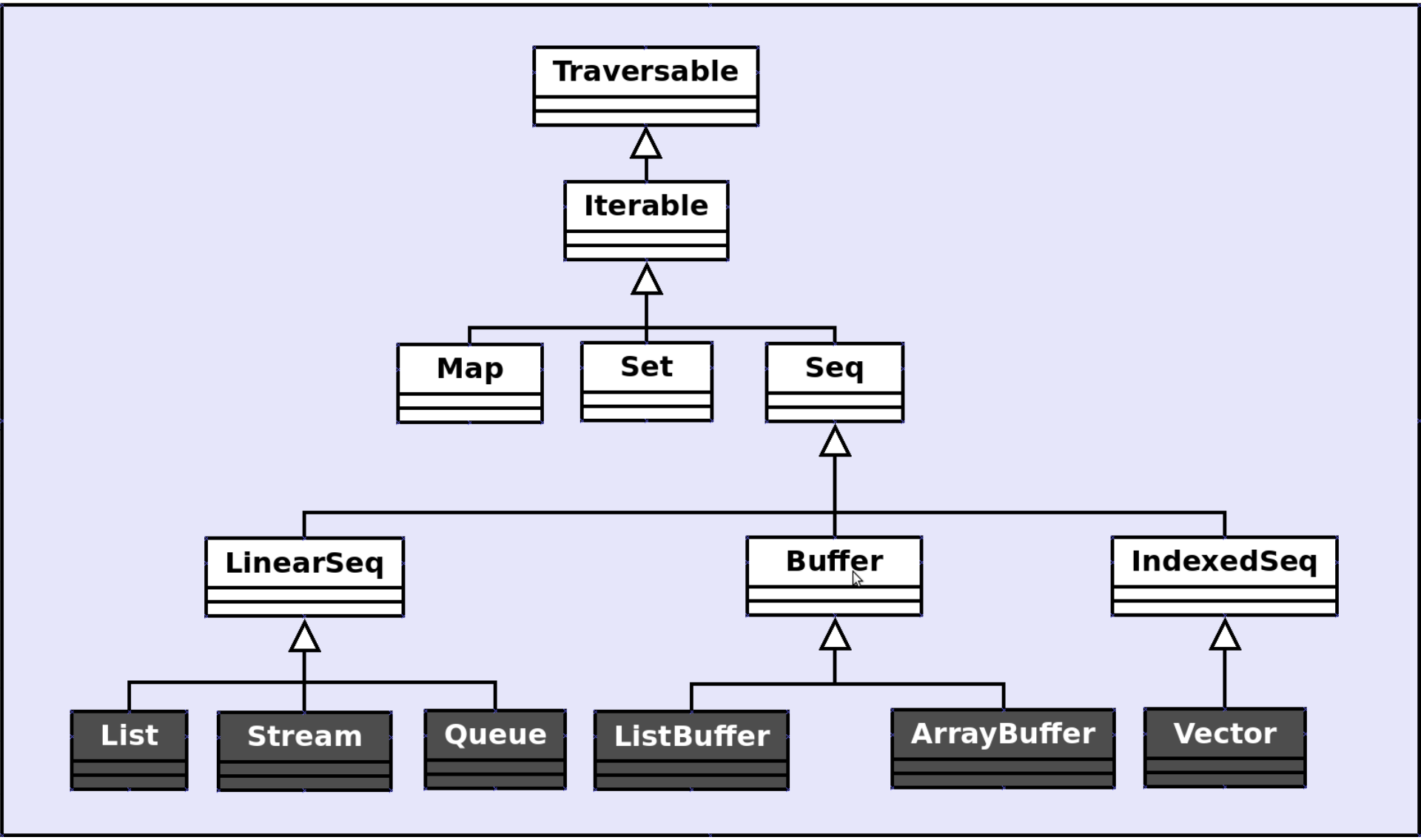
Гуляев Г.М.

Современные технологии программирования

Лекция 6. Коллекции, неявные преобразования, поиск по шаблону

Центр компетенции СПО Алтайского края

Иерархия коллекций



Коллекции Scala

- **Примеры:**

```
val t  = Traversable(1,2,3) // List(1, 2, 3)
val i  = Iterable(1,2,3) // List(1, 2, 3)
val sq = Seq(1,2,3) // List(1, 2, 3)
val is = IndexedSeq(1,2,3) // Vector(1, 2, 3)
val ls = scala.collection.LinearSeq(1,2,3) // List(1, 2, 3)
val st  = Set(1,2,3) // Set(1, 2, 3)
val m = Map(1 -> "a",2 -> "b",3 -> "c") // Map(1 -> a, 2 -> b, 3 -> c)
```

- **Основные рабочие коллекции делятся на изменяемые (mutable) и неизменяемые (immutable):**

```
List, Stream, Queue, Vector // неизменяемые
ListBuffer, ArrayBuffer // изменяемые
```

- **Отметим, что в число коллекций не входит Array, так как он напрямую транслируется в java.util.Array. Хотя все методы трейта Seq он также поддерживает.**
- **Особыми видами коллекций являются Map и Set, все остальные наследуют и реализуют методы трейта Seq.**

Класс List

- **List - основной класс для неизменяемых списков и вообще наиболее общая структура данных в функциональном программировании.**
- **Агрегатный оператор ++ для объединения списков:**

```
val odds = List(1,3,5,7,9)
val evens = List(2,4,6,8)
val nums = odds ++ evens // List(1,3,5,7,9,2,4,6,8)
val nums = odds.++(evens) // List(1,3,5,7,9,2,4,6,8)
```

- **Агрегатные операторы :: :+ для добавления элементов:**

```
val dig = 0 :: nums // List(0,1,3,5,7,9,2,4,6,8)
Nums :+9 // List(1,3,5,7,9,2,4,6,8,9)
val list1 = List("программирование", "на", "Scala")
val list2 = "Люди" :: "должны" :: "изучить" :: list1
```

- **Оператор :: без скобок выполняется справа налево, то есть определение выше эквивалентно следующему:**

```
val list2 = ("Люди" :: ("должны" :: ("изучить" :: list1)))
```

- **Еще одно эквивалентное определение:**

```
val list2 = list1.::("изучить").::("должны").::("Люди")
```

Класс List

- Как и для других коллекций, элементами **List** могут быть любые объекты. Пустой список обозначается **Nil**.

```
val list = List() // List[Nothing]
list == Nil // true
val list = List[Int]() // List[Int]
list == Nil // true
val list = List(2,"a") // List[Any]
list(1) // a
```

- Простая сортировка:

```
val list = List(1,-2,3,2,-1,0,-3)
list.sorted // List(-3, -2, -1, 0, 1, 2, 3)
list.sorted.reverse // List(3, 2, 1, 0, -1, -2, -3)
List("b","a").sorted // List(a, b)
List(1,"a").sorted // ошибка (у Any не задан порядок)
"Привет".sorted // Пвеирт (строки также являются Seq)
```

- Сортировка **sortWith**:

```
list.sortWith((x,y) => if (x*y>0) y<x else x<y) // List(-1, -2, -3, 0, 3, 2, 1)
List("a",1).sortWith((x,y) => x+"" < y+"") // List(1, a)
```

Класс List

- **Полезные функции:**

```
val list = List(1,-2,3,2,-1,0,-3)
list.head // 1
list.tail // List(-2, 3, 2, -1, 0, -3)
list.last // -3
list.take(4) // List(1, -2, 3, 2)
list.takeRight(4) // List(2, -1, 0, -3)
list.slice(3,6) // List(2, -1, 0)
list.sum // 0
list.min // -3
list.max // 3
list.contains(3) // true
list.indexOf(3) // 2
list.contains(5) // false
list.mkString // 1-232-10-3
list.mkString(",") // 1,-2,3,2,-1,0,-3
```

- **Подсчет, фильтрация и изменение элементов:**

```
list.count(x => x*x>1) // 4
list.filter(x => x >0) // List(1, 3, 2)
list.map(x => if (x<0) -x*x else x*x) // List(1, -4, 9, 4, -1, 0, -9)
```

Класс List

- **Разбиение на группы:**

```
val list = List(1,-2,3,2,-1,0,-3)
list.splitAt(4) // (List(1, -2, 3, 2),List(-1, 0, -3))
list.partition(x => x>0) // (List(1, 3, 2),List(-2, -1, 0, -3))
list.groupBy(x => x*x)
//  Map(4 -> List(-2, 2), 1 -> List(1, -1), 9 -> List(3, -3), 0 -> List(0))
```

- **Пересечение, разность, перестановки, сочетания:**

```
List(1,2,3) intersect List(2,3,4) // List(2, 3)
List(1,2,3) diff List(2,3,4) // List(1)
List(1,2,3).permutations.toList
// List(List(1, 2, 3), List(1, 3, 2), List(2, 1, 3), List(2, 3, 1), List(3, 1, 2), List(3, 2, 1))
List(1,2,3).combinations(2).toList // List(List(1, 2), List(1, 3), List(2, 3))
```

- **Слияние списков:**

```
List(List(1,2),List(3,4)).flatten // List(1, 2, 3, 4)
val list = List(1,2) zip List("a","b") // List((1,a), (2,b))
list.unzip // (List(1, 2),List(a, b))
```

- **Циклическая обработка:**

```
List(1,2,3,4).foldLeft(5)((x,y) => x+2*y) // 25
List(1,2,3,4).reduceLeft((x,y) => x+2*y) // 19
```

Класс Set

- На примере класса **List** были рассмотрены основные методы трейта **Seq**. Почти все из них имеются и у класса **Set**, используемому для множеств.
- Отметим различие в объединении:

(**++** равносильно **union**, **--** равносильно **diff**, **&** равносильно **intersect**)

```
List(1,2,3,4) ++ List(3,4,5,6) // List(1, 2, 3, 4, 3, 4, 5, 6)
Set(1,2,3,4) ++ Set(3,4,5,6) // Set(5, 1, 6, 2, 3, 4)
```

- Для множества (**Set**) не важен порядок элементов и невозможно дублирование элементов:

```
Set(1,2,3) == Set(3,1,2) // true
Set(1,1,2,2) // Set(1, 2)
Set(1,2,3) + 2 // Set(1, 2, 3)
```

- Для множеств отсутствуют методы сортировок и агрегатный метод добавления элементов **::**
- Множества, однако, являются **Iterable** и у них существует внутренний порядок для обхода всех элементов:

```
Set(1, 2, 3, 4, 5).toList // List(5, 1, 2, 3, 4)
Set(5, 4, 3, 2, 1).toList // List(5, 1, 2, 3, 4)
```

Неявные преобразования (implicit conversion)

- Функции объявленные как **implicit** задают неявное преобразование, которое происходит автоматически:

```
case class Hi(name: String) {def hi = println("Привет, "+name)}
implicit def toHi(s: String) = Hi(s) // неявное преобразование к Hi
"Вася".hi // Привет, Вася
```

- Компилятор Scala не находит у строки метода **hi**. Тогда он ищет в пределах видимости неявное преобразование к классу у которого есть такой метод.
- Применение: упрощение кода для часто повторяющихся преобразований и расширение уже существующих классов.
- Например, захотелось нам чтобы у натуральных чисел появился метод **sum**, который возвращал бы сумму цифр числа. И вот пожалуйста:

```
implicit def toList(n: Int) = (""+n).map(x => (""+x).toInt)
12345.sum // 15
```

Неявные преобразования (implicit conversion)

- **Пример расширения функционала класса java.io.File:**

```
// добавим функцию read чтения текста из файла
import  java.io.File
class RichFile(f: File) {
  def read = scala.io.Source.fromFile(f.getPath).mkString
}
implicit def toRichFile(f: File) = new RichFile(f)
val text = new File("name.txt").read // прочитали name.txt
```

- **Случаи поиска неявного преобразования компилятором:**
  1. Тип выражения не соответствует требуемому
  2. Отсутствует нужный метод у объекта
  3. Параметры метода не соответствует требуемым
- **Компилятор не использует преобразования, если можно скомпилировать без них и не выполняет одновременно несколько преобразований одного объекта.**
- **Неоднозначные преобразования приводят к ошибке.**

Неявные преобразования (implicit conversion)

- **Класс комплексные числа:**

```
case class Complex(x: Double, y: Double) {
  val d = x*x+y*y
  val mod = math.sqrt(d) // модуль
  def unary_~ = Complex(x,-y) // сопряженное число
  def inv = ~Complex(x/d,y/d) // обратное число
  def +(z: Complex) = Complex(x+z.x, y+z.y) // сложение
  def -(z: Complex) = Complex(x-z.x, y-z.y) // вычитание
  def *(z: Complex) = Complex(x*z.x-y*z.y, x*z.y+y*z.x) // умножение
  def /(z: Complex) = this * z.inv // деление
  def r(d: Double) = math.round(d * 100)/100d
  override def toString = {
    if (x==0) {
      if (y==0) "0" else r(y) + "*i"
    } else if (y==0) ""+r(x)
      else if (y>0) r(x)  + " + " + r(y) + "*i"
      else r(x) + " - " + r(-y)+ "*i"
  }
}
```

Неявные преобразования (implicit conversion)

- **Для удобства напишем неявные преобразования:**

```
implicit def intToComplex(x: Int) = Complex(x,0)
implicit def doubleToComplex(x: Double) = Complex(x,0)

val i = Complex(0,1)  // 1.0*i
~i  // -1.0*i
1 / i // -1.0*i
(1-i)/(1+i) // -1.0*i
val z = 5-3*i // 5.0 - 3.0*i
z * z // 16.0 - 30.0*i
(-2+5*i)/(1+3*i) + i + (-3+8*i)/(-1+2*i) // 5.1 + 1.7*i

def pow(c: Complex, n: Int): Complex = {
  if (n>0) c*pow(c,n-1)
  else if (n<0) 1/pow(c,-n)
  else 1
}

(1 to 100).map(n => pow(i,n)).reduceLeft((x,y) => x+y)  // 0
```

Оператор match

- Для выбора варианта из списка по шаблону в Scala существует очень мощное средство - оператор **match**:

```
expr match { // expr - любое выражение
  case pattern1 => result1
  case pattern2 => result2
  ...
}
```

- Простой пример:

```
val list = List(true, 1,"a", false)

for (x <- list) {
  x match {
    case true => println("начало")
    case false => println("конец")
    case _ => println("что-то внутри")
  }
}
```

- Подчеркивание (_) означает значение по-умолчанию (все что осталось). Выбор первого подходящего значения в порядке записи. Подчеркивание должно быть в конце.

Оператор match (примеры)

- **Может быть несколько подчеркиваний в конце:**

```
for (ch <- "+-3!") {
  var sign, digit = 0
  ch match {
    case '+' => sign = 1
    case '-' => sign = -1
    case _ if Character.isDigit(ch) => digit = Character.digit(ch, 10)
    case _ => sign = 0
  }
  println(ch + ": " + sign + " " + digit)
}
```

- **Выбор значения переменной:**

```
def number = {
  import scala.util.Random
  val x = new Random().nextInt(10)
  x  match {
    case 7 => println("Ура, счастливая семерка!")
    case other => println("Увы, это скучная цифра " + other)
  }
}
```

Оператор match (примеры)

- **Выбор типа объекта:**

```
val list = List(23, "Hello", 8.5, 'q')

for (x <- list) {
  x match {
    case i: Int => println("Это целое: " + i)
    case s: String => println("Это строка: " + s)
    case f: Double => println("Это вещественное: " + f)
    case other => println("Что-то другое: " + other)
  }
}
```

- **Выбор составного типа:**

```
for (o <- Array(Map("Иван" -> 42), Map(42 -> "Иван"), Array(42), Array("Иван"))) {
  println(o match {
    case m: Map[_,_] => "Это Map[_,_]"
    case a: Array[Int] => "Это Array[Int]"
    case a: Array[_] => "Это тоже Array, но не Array[Int]"
  })
}
```

Оператор match (примеры)

- **Выбор списка по шаблону:**

```
val list1 = List(1, 3, 23, 90)
val list2 = List(4, 18, 52)
val empty = List()
for (l <- List(list1, list2, empty)) {
  l match {
    case List(_, 3, _, _) => println("4 элемента, второй - 3")
    case List(_*) => println("Какой-то другой список")
  }
}
```

- **Выбор массива по его структуре:**

```
for (arr <- Array(Array(0), Array(1, 0), Array(0, 1, 0), Array(1, 1, 0))) {
  println(arr match {
    case Array(0) => "0"
    case Array(x, y) => x + " " + y
    case Array(0, _*) => "0 ..."
    case _ => "другое"
  })
}
```

Оператор match (примеры)

- **Выбор кортежей:**

```
for (pair <- Array((0, 1), (1, 0), (1, 1))) {
  println(pair match {
    case (0, _) => "0 ..."
    case (y, 0) => y + " 0"
    case _ => "на 1-2 месте не 0"
  })
}
```

- **Выбор кортежей по условию:**

```
for (t <- List(("Доброе", "утро!"), ("Good", "Morning!"))) {
  t match {
    case (one, two) if one == "Good" =>  println("Первый элемент: Good")
    case (one, two) =>  println("Эта пара: " + one + " " + two)
  }
}
```

- **Регулярные выражения:**

```
val pattern = "([0-9]+) ([а-я]+)".r
"99 бутылок" match {
  case pattern(num, item) => (num.toInt, item)
} // (99,бутылок)
```

Оператор match (примеры)

- **Выбор case-классов:**

```
case class Person(name: String, age: Int)
val alice = Person("Алиса", 25)
val bob = Person("Боб", 32)
val charlie = Person("Чарли", 32)
for (person <- List(alice, bob, charlie)) {
  person match {
    case Person("Алиса", 25) => println("Привет Алиса!")
    case Person("Боб", 32) => println("Привет Боб!")
    case Person(name, age) =>
            println(age+" года, зовут "+name+" - кто это?")
  }
}
```

- **Рекурсивный обход List:**

```
def process(l: List[Person]): Unit = l match {
  case head::tail => println(head); process(tail)
  case Nil => println("")
}
process(List(alice,bob,charlie))
```

Оператор match (примеры)

- **Вложенные match:**

```
class Role
case object Manager extends Role
case object Developer extends Role
case class Person(name: String, age: Int, role: Role)

val alice = Person("Алиса", 25, Developer)
val bob = Person("Боб", 32, Manager)
val charlie = Person("Чарли", 32, Developer)

for (item <- Map(1 -> alice, 2 -> bob, 3 -> charlie)) {
  item match {
    case (id, p: Person) => p.role match {
      case Manager => println (p.name+" - высокая зарплата")
      case _ => println (p.name+" - средняя зарплата")
    }
  }
}
```

Поиск по шаблону

- Поиск по шаблону (pattern matching) является мощным и эффективным способом извлечения информации из объектов
- Благодаря ему в Scala удается избегать необходимости хранить отдельно состояние или иерархию объектов. Он помогает находить решения многих задач в чисто функциональном стиле
- Добавление ключевого слова **case** при объявлении класса заставляет компилятор автоматически добавить в класс ряд полезных функций (геттеры, сеттеры, **toString**, **equals**, **hashCode**)
- Case-классы в сочетании с поиском по шаблону позволяют в нескольких строках кода решать достаточно сложные задачи, требующие в некоторых других языках написания нескольких страниц кода

Задания для самостоятельной работы

- Реализуйте для чисел **Int** и **Double** оператор **+%**, добавляющий процент к числу, например 120 +% 10 = 132.
- Список содержит целые числа, а также другие списки, такие же как и первоначальный. Получить список, содержащий только целые числа из всех вложенных списков. Пример:

  f(List(List(1, 1), 2, List(3, List(5, 8)))) = List(1, 1, 2, 3, 5, 8)
- Список содержит элементы любого типа. Среди элементов есть повторяющиеся. Получить список, содержащий каждый элемент только один раз (удалить все повторы).
- Элементами главного списка является другие списки. Отсортировать главный список, расположив внутренние списки по возрастанию количества их элементов.
- Список содержит целые числа и строки. Получить два списка из элементов исходного, выбирая в первый числа, а во второй строки.
- Список в качестве элементов содержит кортежи типа: (n, s), где n — целые числа, а s — строки. Получить два списка из элементов исходного, выбирая в первый числа, а во второй строки из кортежей.
- Получить два списка из элементов исходного, выбирая в первый элементы с четными индексами, а во второй с нечетными.

Спасибо за внимание !

www.altailand.ru