Лекция 4

Обобщённые типы в языке Scala

Обобщённый тип (generic type) – это тип, имеющий формальные типовые параметры

Формальный типовый параметр (type parameter) – это тип, который используется в объявлении обобщённого класса и при этом неизвестен во время компиляции обобщённого класса

Типовые параметры в объявлении обобщённого типа перечисляются в квадратных скобках после имени типа:

```
class Point2[T] (val x: T, val y: T)
```

Пример обобщённого типа с двумя типовыми параметрами:

```
class KeyValuePair[K, V] (val key: K, val value: V)
```

Параметризованный тип (parameterized type) создаётся на основе обобщённого путём указания фактических типовых параметров: `Point2[Int]`, `Point2[Double]` и т.д.

Использование параметризованных типов:

```
val p1 = new Point2[Double](1.5, -4.2)
  // p1: Point2[Double]

// типовые параметры могут быть выведены:
val p2 = new Point2(1, 2)
  // p2: Point2[Int]

val month_index = new KeyValuePair("April", 4)
  // month_index: KeyValuePair[String, Int]
```

Кроме обобщённых типов, типовые параметры могут присутствовать у методов. Например, как в методе `map` класса `Point2`:

```
1 class Point2[T] (val x: T, val y: T) {
2   def map[R] (f: T => R): Point2[R] =
3     new Point2[R](f(x), f(y))

5   override def toString: String = "(%s,%s)".format(x,y)
6 }

8 object Test {
9   def main(args: Array[String]): Unit = {
10    val p = new Point2(10, 5)
11    val r = p.map(_.toDouble/2)
12    println(r) // печать (5.0,2.5)
13  }
14 }
```

Для формальных типовых параметров можно указывать верхние и нижние границы:

`T <: H` – на место `T` можно подставить либо `H`, либо наследника `H`

`T >: L` – на место `T` можно подставить либо `L`, либо суперкласс `L`

Пример (указание верхней границы):

```
1 class Rect(val dx: Double, val dy: Double)
2 class Square(size: Double) extends Rect(size, size)

4 class Frame[T <: Rect]
5     (val padX: Double, val padY: Double, val shape: T) {
6   val width: Double = shape.dx + 2*padX
7   val height: Double = shape.dy + 2*padY
8   val area: Double = width * height
9 }
```

Кроме того, можно указывать сразу и нижнюю, и верхнюю границу:

`T >: L <: H`

Пример (указание и нижней, и верхней границы):

```
1  class Rect(val dx: Double, val dy: Double)
2  class Square(size: Double) extends Rect(size, size)

4  class Group[T <: Rect] private (list: List[T]) {
5    def this() = this(Nil)
6    def append[U >: T <: Rect] (x: U) =
7      new Group[U] (x :: list)
8    def dx: Double = list.foldLeft(0.0)(_ + _.dx)
9    def dy: Double = list.foldLeft(0.0)(_ + _.dy)
10 }

12 object Test {
13   def main(args: Array[String]) {
14     val grp: Group[Rect] = new Group[Square].
15       append(new Square(10.0)).
16       append(new Square(20.0)).
17       append(new Rect(5.0, 6.0))
18     println(grp.dx, grp.dy)
19   }
20 }
```

Пусть обобщённый тип `G` имеет формальный типовый параметр `T` и, возможно, другие типовые параметры: `G[..., T, ...]`

Пусть тип `A` – супертип (базовый класс) для типа `B`

Тогда типы `G[..., A, ...]` и `G[..., B, ...]`, полученные из `G` путём подстановки типов `A` и `B` на место параметра `T` (все остальные параметры у них совпадают), не связаны отношением наследования, т.е. обобщённый тип `G` – *инвариантен* по отношению к типовому параметру `T`

Тип `G` можно сделать *ковариантным* по отношению к `T`, если объявить его как `G[..., +T, ...]`. В этом случае `G[..., A, ...]` будет супертипом для `G[..., B, ...]`

Тип `G` можно сделать *контравариантным* по отношению к `T`, если объявить его как `G[..., -T, ...]`. В этом случае `G[..., A, ...]` будет подтипом `G[..., B, ...]`

Пример (ковариантность):

```
1 class Rect(val dx: Double, val dy: Double)
2 class Square(size: Double) extends Rect(size, size)

4 class Frame[+T <: Rect]
5     (val padX: Double, val padY: Double, val shape: T) {
6   val width: Double = shape.dx + 2*padX
7   val height: Double = shape.dy + 2*padY
8   val area: Double = width * height
9 }

11 object Test {
12   def perimeter(f: Frame[Rect]): Double =
13     (f.width + f.height)*2
14   def main(args: Array[String]) {
15     val box = new Frame[Square](5, 6, new Square(30))
16     println(perimeter(box)) // выведет 164.0
17   }
18 }
```

Ковариантность не всегда возможна. В частности, если обобщённый тип, ковариантный относительно параметра T, имеет метод, формальный параметр которого имеет тип T, то ковариантность невозможна

Например, следующий класс компилироваться не будет:

```
1 class Frame[+T <: Rect]
2     (val padX: Double, val padY: Double, val shape: T) {
3   val width: Double = shape.dx + 2*padX
4   val height: Double = shape.dy + 2*padY
5   val area: Double = width * height
6   def replace(newShape: T) =
7     new Frame[T] (padX, padY, newShape)
8 }
```

Если бы он откомпилировался, мы могли бы нарушить систему типов:

```
1 val squareBox = new Frame[Square](10, new Square(100))
2 val rectBox: Frame[Rect] = squareBox
3 val invalidBox = rectBox.replace(new Rect(20, 30))
4   // invalidBox.shape имеет тип Square, но содержит Rect
```

Пример (контравариантность):

```
1  class Framer[-T <: Rect]
2      (val width: Double, val height: Double) {
3    def getPadX(shape: T): Double = (width - shape.dx)/2
4    def getPadY(shape: T): Double = (height - shape.dy)/2
5  }

7  object Test {
8    def toFrames(lst: List[Square],
9                 f: Framer[Square]): List[Frame[Square]] =
10     lst.map(sq =>
11       new Frame[Square](f.getPadX(sq), f.getPadY(sq), sq))
12   def main(args: Array[String]) {
13     val framer = new Framer[Rect](30, 20)
14     val squares =
15       List(new Square(10), new Square(15), new Square(18))
16     toFrames(squares, framer).foreach(println(_))
17   }
18 }
```

Предположим, что есть два класса `A` и `B`, не связанные отношением наследования. И нам нужно полиморфно обрабатывать объекты этих классов

Для этого в Scala используется следующий приём: для каждого класса создаётся специальный объект-обработчик, выполняющий с объектами класса те операции, которые мы собираемся полиморфно вызывать

Например, нам нужно уметь полиморфно складывать как целые числа, так и строки (для строк сложение – это конкатенация)

Определим абстрактный обобщённый класс `Adder` с операцией сложения двух значений:

```
1 abstract class Adder[T] {
2   def sum(a: T, b: T): T
3 }
```

Предполагается, что обработчик целых чисел будет объектом класса, производного от `Adder[Int]`, а обработчик строк – объектом класса, производного от `Adder[String]`

Так как не имеет смысла создавать несколько экземпляров объектов-обработчиков, то мы оформим их в виде синглетонов и раположим в объекте-компаньоне класса `Adder`:

```
1 object Adder {
2   object int extends Adder[Int] {
3     override def sum(a: Int, b: Int): Int = a + b
4   }

6   object str extends Adder[String] {
7     override def sum(a: String, b: String): String = a + b
8   }
9 }
```

Теперь, например, если мы захотим уметь просуммировать три значения, мы можем написать обобщённый метод, принимающий `Adder[T]` в качестве параметра:

```
def sum3[T] (a:T, b:T, c:T, adder: Adder[T]): T =
  adder.sum(adder.sum(a, b), c)
```

Воспользуемся объектами `Adder`, чтобы написать полиморфную функцию `fibonacci`, которая в зависимости от переданного объекта-обработчика порождает либо последовательность чисел Фибоначчи, либо последовательность строк Фибоначчи:

```
1 object Test {
2   def fibonacci[T](a: T, b: T, adder: Adder[T]): LazyList[T] =
3     a #:: fibonacci[T](b, adder.sum(a, b), adder)

5   def main(args: Array[String]) {
6     fibonacci(1, 1, Adder.int).take(10).foreach(println(_))
7     fibonacci("a", "b", Adder.str).take(10).foreach(println(_))
8   }
9 }
```

Единствнное, что неудобно при вызове функции `fibonacci` — это то, что ей приходится явно передавать объект-обработчик. К счастью, это можно делать неявно благодаря *неявным объектам*.

Неявный объект – это объект, который может автоматически передаваться методам в качестве *неявного параметра*. Для того, чтобы объект стал неявным, его нужно объявить с модификатором `implicit`:

```
1 object Adder {
2   implicit object int extends Adder[Int] {
3     override def sum(a: Int, b: Int): Int = a + b
4   }

6   implicit object str extends Adder[String] {
7     override def sum(a: String, b: String): String = a + b
8   }
9 }
```

Параметр можно сделать неявным, если объявить его в отдельном списке параметров, предварив ключевым словом `implicit`:

```
def sum3[T] (a:T, b:T, c:T) (implicit adder: Adder[T]): T =
  adder.sum(adder.sum(a, b), c)
```

При вызове метода значение неявного параметра можно не указывать. Подразумевается, что компилятор сам найдёт и передаст методу доступный в точке вызова подходящий неявный объект.

Например, с помощью неявного параметра определение и использование функции `fibonacci` приобретает вид:

```
1 object Test {
2   def fibonacci[T](a: T, b: T)
3       (implicit adder: Adder[T]): LazyList[T] =
4     a #:: fibonacci[T](b, adder.sum(a, b))

6   def main(args: Array[String]) {
7     fibonacci(1, 1).take(10).foreach(println(_))
8     fibonacci("a", "b").take(10).foreach(println(_))
9   }
10 }
```

Функцию `fibonacci` можно переписать с помощью *контекстного ограничения*:

```
1 object Test {
2   def fibonacci[T: Adder](a: T, b: T): LazyList[T] =
3     a #:: fibonacci[T](b, implicitly[Adder[T]].sum(a, b))

5   def main(args: Array[String]) {
6     fibonacci(1, 1).take(10).foreach(println(_))
7     fibonacci("a", "b").take(10).foreach(println(_))
8   }
9 }
```

Контекстное ограничение вида `A : B` на типовый параметр `A` означает, что в точке вызова функции (или конструктора класса, если речь идёт о параметре конструктора) доступен неявный объект типа `B[A]`

Библиотечный метод `implicitly` в теле функции `fibonacci` возвращает ссылку на найденный неявный объект.