# Scala

Василий Купоросов

19 декабря 2018 г.

## Содержание

# 1. Основы

## 1.1. Немного истории

В Java до версии 5 не было `generic`'ов. Затем с участием Мартина Одерски, исследователя в области языков программирования, дженерики добавили. Тем не менее, стало понятно, что в рамках обратной совместимости нормально это сделать не получится. Тогда Одерски решил создать свой язык, с нормальной системой типов. Изначально Scala была академическим языком (для исследований в области ЯП), но постепенно она переросла в промышленный. Как академический язык, Scala развивалась силами студентов, которые пилили и добавляли свои фичи. Это привело к тому, что фич стало очень много, некоторые из них позволяют делать одно и то же действие разными способами. В связи с этим существуют разные мнения, как надо писать на этом языке. Однако есть официальный стайлгайд, которого желательно придерживаться.

Scala совмещает в себе функциональную и объектно-ориентированную парадигмы. Начнем мы с ООП, так как это более привычно читателям. Многие вещи уже знакомы нам из Котлина и Java, на них подробно останавливаться не будем.

## 1.2. Классы

### 1.2.1. `class`

Обычный класс, как в Java. Внутри можно заводить поля и методы.

```
class Foo {
  val bar: Int = 42
  def foo(): Int = bar + 1
}
```

У класса бывают примарный и побочный конструкторы, побочными пользуются редко. Побочный конструктор объявляется ключевым словом `this`.

```
class Bar(innerFoo: Int) {
  this(doubleFoo: Double) = // побочный конструктор
    this(doubleFoo.toInt)   // вызывает примарный

  def foo(): Int = innerFoo // используем значение из примерного конструктора
}
```

### 1.2.2. `trait`

`trait` – это интерфейс. Интерфейсы в скале поддерживают дефолтную реализацию методов. Также в интерфейсе можно завести немутабельное поле без значения.

```
trait Bar {
  val bar: Double
  def foo(): Int = 42
}
```

### 1.2.3. `object`

Объекты и `companion`-объекты в Scala есть. Они мощнее чем в Котлине, но об этом позже. Также надо заметить, то статических полей и методов в классах и интерфейсах нет, статичность достигается как раз за счет `object`'ов. Чтобы объявить `companion object`, надо написать объект с именем нужного класса.

```
object Foo // синглтон Foo
class Bar // класс Bar
object Bar // companion object для класса Bar
```

## 1.3. Наследование

Наследоваться, как и в Java, можно от одного класса и любого количества интерфейсов, но ключевое слово для них одно – `extends`. для множественного наследования используется ключевое слово `with`. Если в множественном наследовании участвует класс, он должен стоять на первом месте.

```
class Foo
trait Bar
trait Baz

class MyClass extends Foo with Bar with Baz
```

При наследовании как у полей, так и у методов надо писать модификатор `override`.

```
class Foo extends Bar {
  override val bar: Double = 42
  override def foo(): Int = bar.toInt
}
```

### 1.3.1. Наследование в `generic`'ах

Вместо `extends` и `super` в Scala используются специальные значки. Это пришло из теории типов (как мы помним, Scala изначально академический язык).

```
class GenericFoo[F <: Foo] // <F extends Foo>
class GenericFoo[F >: Foo] // <F super Foo>
```

Символ `<:` также можно использовать для наследования вместо `extends`.

```
trait Foo <: Serializable // trait Foo extends Serializable
```

## 1.4. Оператор `new` и функциональный метод `apply()`

В отличие от Котлина, в Scala надо писать `new` при инстанцировании объекта. В Scala это пришло из Java, а туда – из C++, в котором это важно – создется ли объект в куче или на стеке.

Есть способ, как не писать `new`. Для этого заведем `companion object` для нашего класса и реализуем в нем метод `apply()`.

```
class Bar
object Bar {
  def apply(): Bar = new Bar
}

val bar = Bar() // теперь можем не писать new
val bar = Bar.apply() // аналогичный вызов
```

Метод `apply()` можно определять с разными аргументами, то есть это что-то вроде перегрузки круглых скобок (`operator ()` в C++).

Также `apply()` может заменить вторичный конструктор. Недостаток только в том, что при объявлении класса-наследника можно использовать только вызовы конструкторов.

```
class Bar(foo: Int) { ... }
object Bar {
  def apply(doubleFoo: Double) = new Bar(doubleFoo.toInt)
}

class Baz extends Bar(42.0) // ошибка
```

## 1.5. Скобки

### 1.5.1. Писать – не писать

В Scala есть конвенция, что если какой-то код можно не писать, его писать не надо. Если у класса нет тела, то фигурные скобки не пишутся, то же и с вызовами функций. Если в функцию не передаются аргументы, и она не имеет побочных эффектов, скобки принято опускать. Если же побочные эффекты есть, скобки все же пишут, чтобы подчеркнуть, что мы вызываем функцию. При объявлении метода без побочных эффектов скобки тоже опускаются.

### 1.5.2. Скобки для `generic`'ов

В Scala для `generic` параметров пишутся квадратные скобки, а не угловые. Так сделано, чтобы избежать неоднозначности при парсинге. Пример, когда угловые скобки распарсить не получается:

```
f1(f2<A, B>(x + 1))
```

## 1.6. Массивы и доступ к элементам

В качестве массивов используется класс `Array`. Поскольку квадратные скобки заняты под `generic`'и, для доступа к элементам они не используются. Зато для `Array` и для некоторых других коллекций определены волшебные методы – уже знакомый нам `apply()` и `update()`. Они позволяют обращаться к элементам через круглые скобки.

```
val array: Array[Int](3)
array(0) = 42 // array.update(0, 42), присваиваем первый элемент
println(array(0)) // array.apply(0), получаем первый элемент
```

## 1.7. Модификаторы доступа

По-умолчанию все члены класса публичные. Для всего остального есть модификаторы `private` и `protected`. Вместо `package private` модификатора используются модификаторы с усилением.

```
package ru.spbau.jvm.scala

class Foo {
  private var foo: Int         // доступ только внутри класса
  private[scala] var bar: Int // доступ только в пакете ru.spbau.jvm.scala
  private[jvm] var baz: Int    // доступ только в пакете ru.spbau.jvm
  private[this] var bus: Int  // доступ только внутри экземпляра

  def fun(other: Foo): Unit = {
    bus = 42       // ок, доступ изнутри экземпляра
    other.bus = 42 // ошибка, доступ из другого экземпляра
  }
}
```

Примарный конструктор тоже можно сделать защищенным или приватным

```
class Foo protected()
class Bar private()
class Baz private[this]()
```

Так как `companion object`'ы имеют доступ к приватным полям своих классов, единственный способ что-то от них скрыть (в т.ч. конструктор) – это `private[this]`. При этом, так как в Java такой фичи нет, в ней этот модификатор воспринимается как просто `private`.

## 1.8. Properties

Они есть. Примерно как в Котлине. **TODO:** об этом позже

## 1.9. Hello world!

Есть несколько способов написать запускающуюся программу на Scala. Самый простой и привычный – это объявить метод `main`.

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello world!")
  }
}
```

Другой способ – отнаследоваться от трейта `App` и написать всё прямо в теле класса. Но таким способом пользоваться не рекомендуется.

```
class HelloWorld extends App {
  println("Hello world!")
}
```

## 1.10. Пакеты, `import`'ы и коллизии

В Scala, как и в Котлине, есть свои коллекции, например `List`. Если мы хотим использовать `java.util.List`, надо либо явно писать весь пакет, либо заимпортить его. Но при импорте Scala'вский `List` перекроется, что не очень хорошо. Поэтому можно импортить прямо пакеты.

```
import java.util
var list: util.List[Int]
```

Но такие названия как `util` тоже могут встречаться часто и конфликтовать. Чтобы этого избежать, можно создавать синонимы для пакетов. В основном это используют именно для `java.util`, для чего-то другого так делают редко.

```
import java.{util => ju}
var list: ju.List[Int]
```

## 1.11. Избавляемся от лишнего

Рассмотрим следующий код и поймем, что в нем можно сократить.

```
def foo(x: Boolean): Int = {
  if (x) return 42
  return 43
}
```

Во-первых, последний `return` можно не писать, потому что результат последнего выражения и так возвращается по-умолчанию. Во-вторых, можно объединить две ветки условия в `if else` и избавиться от `return` совсем.

```
def foo(x: Boolean): Int = {
  if (x) 42 else 43
}
```

Теперь у нас функция состоит из одного выражения, поэтому можно убрать скобки и возвращаемый тип, так как он выведется.

```
def foo(x: Boolean) = if (x) 42 else 43
```

## 1.12. Рекурсия

Напишем функцию, считающую факториал числа.

```
def fac(n: Int): BigInt =
  if (n == 1) 1
  else n * fac(n - 1)
```

Проблема такой реализации в том, что при вызове `fac(10000)` случится переполнение стека. Хочется сделать хвостовую рекурсию, а затем развернуть ее в цикл.

```
@tailrec
def facTailRec(n: Int, accumulator: BigInt = 1): BigInt = n match {
  case 1 => accumulator
  case _ => facTailRec(n - 1, n * accumulator)
}
```

Здесь мы видим сразу две фичи – аннотацию `@tailrec` и матчер. Матчер работает примерно как `when` в Котлине. Аннотация `@tailrec` нужна только для того, чтобы удостовериться, что в методе используется хвостовая рекурсия. В противном случае код не скомпилируется. Компилятор же **всегда** разворачивает хвостовую рекурсию в цикл.

## 1.13. Метод в методе

В нашей реализации факториала есть одна концептуальная проблема. Мы можем передать туда любой аккумулятор, и это повлияет на результат. Чтобы этого избежать, можно обернуть наш метод в другой метод, таким образом спрятав его от нерадивых пользователей.

```
def facTailRec(n: Int): BigInt = {
  @tailrec
  def facTailRecInner(n: Int, accumulator: BigInt = 1): BigInt = n match {
    case 1 => accumulator
    case _ => facTailRec(n − 1, n * accumulator)
  }

  facTailRecInner(n)
}
```

## 1.14. `try`

Блок `catch` очень похож на `match` тем, что внутри мы матчим исключения, чтобы по-разному их обрабатывать. Рассмотрим пример.

```
try {
  fac(100000)
} catch {
  case e: StackOverflowError => facTailRec(100000)
  case _: IOException | _: FileException => ... // комбинируем случаи
  _: // матчим всё что не сматчилось ранее
}
```

**TODO:** подробнее будет позже, вместе с `match`

## 1.15. Алгебраические типы данных

Хотим написать список в функциональном стиле (как в Haskell).

```
sealed trait IntList

final class IntNil extends IntList
final class IntCons(val head: Int, val tail: IntList) extends IntList
```

Теперь можем заводить списки.

```
val list: IntList = new IntCons(1, new IntCons(2, new IntCons(3, new IntNil)))
```

А потом матчить.

```
list match {
  case cons: IntCons => println(cons.head) // теперь cons = list.asInstanceOf[IntCons]
  case _: IntNil => // ничего не делаем
}
```

Поскольку трейт `IntList` мы объявили `sealed`, у нас есть гарантия, что больше никаких наследников кроме этих двух нет, то есть всегда что-то заматчится. Иначе мы могли бы завести инстанс анонимного класса, и матчер бы свалился, потому что никакая ветка бы не подошла.

## 1.16. `case` классы

В нашей реализации списка тоже есть лишний код, от которого лучше избавиться. Для начала поймем, что `IntNil` бывает всего один, поэтому его можно сделать `object`'ом. А теперь добавим к наследникам сключевое слово `case`. Оно создает для класса `companion object`, определяет для него методы `apply()` и `thenApply()` (о нем позже). Сделает неизменяемые параметры примарного конструктора полями (не надо будет писать в конструкторе `val`) и сгенерирует методы `equals()`, `hashCode()` и `toString()`. В Котлине очень похожее поведение имеют `data`-классы.

```
sealed trait IntList

case object IntNil extends IntList
final case class IntCons(head: Int, tail: IntList) extends IntList

val list: IntList = IntCons(1, IntCons(2, IntCons(3, IntNil)))
```

Более того, теперь в матчере для `IntNil` не надо писать плейсхолдер, а `IntCons` теперь можно матчить по его параметрам. Теперь можем рекурсивно вывести список на экран.

```
def printList(list: IntList): Unit = {
  list match {
    case IntNil =>
    case IntCons(head, tail) =>
      println(head)
      printList(tail)
  }
}
```

# 2. Функциональное программирование на Scala

Как мы помним, Scala – симбиоз функционального и объектно-ориентированного языков. Так как в JVM не предусмотрены фичи функциональной парадигмы, всё сделано синтетически.

## 2.1. Функциональные типы

Рассмотрим следующий метод.

```
def increase(x: Int): Int = x + 1
```

В байт коде это будет обычная функция, увеличивающая свой аргумент на единицу.

Можно написать то же самое в функциональном стиле.

```
val increase: Int => Int = x => x + 1
```

Здесь `Int => Int` – это функциональный тип, а `x => x + 1` – объявление функции. Можно написать еще более функционально.

```
val increase: Int => Int = _ + 1
```

В этих примерах надо обязательно явно указывать тип, потому что иначе компилятор будет думать, что аргумент имеет тип `Any`, и у него нет оператора `+`.

Наиболее интересная версия. Тут можно без явного указания типа.

```
val increase = (_: Int) + 1
```

Запись `Int => Int` – это синтаксический сахар для `Function1[Int, Int]`. То есть с точки зрения Java, это будет просто класс функция, у которой можно вызвать метод `apply()`.

Функциональные классы и другие подобные штуки в Scala написаны вручную и существуют лишь до 22, то есть могут принимать максимум 22 аргумента. В Scala 3 их немного переделают, и это ограничение уйдет.

Рассмотрим теперь функцию с двумя аргументами.
У нее будет тип `Function2[Int, Int, Int]`.

```
val sum: (Int, Int) => Int = _ + _
```

Если хочется как-то различать аргументы, например, если хотим сложить их в другом порядке, можно применить pattern matching.

```
val sum: (Int, Int) => Int = {
  case (left, right) => right + left
}
```

## 2.2. Частичное применение

С точки зрения компилятора предыдущий пример – это частичная функция. Она позволяет матчить аргументы и проверять частные случаи.

```
val weirdSum: PartialFunction[(Int, Int), Int] = {
  case (0, _) | (_, 0) => 0 // если хотя бы один ноль, то сумма тоже ноль
  case (left, right) => left + right
}
```

Если мы в первом кейсе заменим ноль на плейсхолдер, то получим частично примененную функцию.

```
val weirdSum: PartialFunction[(Int, Int), Int] = {
  case (0, _) | (_, 0) => _
}
```

Если оба аргумента будут ненулевые, то pattern matching не пройдется, и выпадет ошибка `MatchError`.

Вернемся к нормальной реализации `sum()` и вызовем её.

```
val sum: PartialFunction[(Int, Int), Int] = {
  case (left, right) => left + right
}

sum(1, 2);
```

Внимательный читатель найдет здесь две особенности. Во-первых, мы вызвали поле `val sum` как функцию. В этом нам помог уже известный нам метод `apply()`, который можно вызывать, просто приписав скобки к переменной. Во-вторых, наша функция должна принимать tuple с двумя `Int`'ами, а мы передали просто два числа. Автоматический тюплинг преобразовал наши аргументы в одну пару и передал в функцию.

То же самое можно написать в виде функции.

```
val sum: (Int, Int) => Int = {
  case (left, right) => left + right
}

def sum(left: Int, right: Int): Int = (left, right) match {
  case (left, right) => left + right
}

sum.apply(1, 2); // вызовется val sum
sum(1, 2); // вызовется def sum
```

Здесь случилась коллизия имен, но компилятор поймет, что если мы вызываем что-то как функцию, то надо вызвать именно функцию, а если через `apply()`, то это переменная.

## 2.3. Каррирование

Каррирование – преобразование функции из `(Int, Int) => Int` в `Int => (Int => Int)`. В Scala как такового каррирования нет, но есть способ его имитировать. Это позволит нам реализовать частичное применение.

### 2.3.1. Placeholder

Самый бессмысленный и беспощадный пример – вызов функции, с заменой всех аргументов на плейсхолдеры.

```
def sum(left: Int, right: Int): Int = left + right
sum(_, _) // выражение имеет тип (Int, Int) => Int
sum(_, _)(1, 2) // и это можно даже вызвать
```

Более содержательный пример – вызов с заменой некоторых аргументов на плейсхолдеры.

```
sum(42, _) // получили выражение с типом Int => Int
```

### 2.3.2. Parameter scope

Можно объявлять функцию с несколькими скоупами параметров, тогда и при вызове придется писать аргументы в разных скобках.

```
def sum(left: Int)(right: Int): Int = left + right
sum(1)(2) // просто вызываем sum
sum(1) _ // частичное применение sum
```

Значения из предыдущих скоупов можно использовать в последующих.

```
def foo(a: Int)(b: Int = a) = ???
```

## 2.4. Вариантность

Вернемся к нашему инкременту и подумаем, какой наиболее общий тип можно приписать нашей функциональной переменной.

```
val increase: ? => ? = (x: Int) => x + 1
```

Принять мы можем любого наследника `Int`, а вернуть любого его потомка. Это называется умными словами – **контрвариантность** аргумента и **ковариантность** возвращаемого значения. Таким образом наиболее общий тип будет такой.

```
val increase: Nothing => Any = (x: Int) => x + 1
```

`Any` – это "Top type" – тип, от которого наследуются все остальные, а `Nothing` – "Bottom type" – является чем-то вроде наследника всех остальных классов.

Для generic параметров тоже можно задавать вариантность. Посмотрим на объявление `Function1`.

```
trait Function1[-T1, +R]
```

В Java эму соответствует такая запись.

```
interface Function1<? extends T1, ? super R> { }
```

## 2.5. Кортежи

Кортежи – это синтаксический сахар для `Tuple1`, `Tuple2`, ... `Tuple22`. В Scala 3 ограничение в 22 элемента тоже уйдет.

К элементам кортежа можно обращаться по номерам (`_1`, `_2`, ...). Элементы нумеруются с единицы.

```
val tuple: (Int, String) = (42, ''str'')
println(tuple._1)
println(tuple._2)
```

Но так писать крайне не рекомендуется. Вместо этого элементы лучше именовать и далее использовать как переменные.

```
val (index, text) = (42, ''str'')
println(index)
println(text)
```

Рассмотрим функцию суммы, принимающая tuple из двух слагаемых.

```
val sum: Function1[Tuple2[Int, Int], Int] = tuple => tuple._1 + tuple._2
```

В Haskell-стиле правильнее было бы писать так.

```
val sum: Function1[Tuple2[Int, Int], Int] = (left, right) = left + right
```

Но в Scala так писать пока нельзя, но скоро будет можно. Пока что приходится писать через `case`.

В Котлине нет тюплов. Там можно распаковывать списки и коллекции, пользуясь похожим на тюплы синтаксисом. Недостаток списков в том, что нельзя во время компиляции (на уровне типов) проверить количество элементов, и в рантайме наш `sum()` может сломаться.

Вообще система типов нужна, чтобы отлавливать ошибки как можно раньше. Поэтому в промышленных языках чаще используется статическая типизация.

# 3. Операции над коллекциями

Здесь была лекция, которой не было. Код оставляется читателю для самостоятельного изучения.

```
final class ListWrapper[A](val toList: List[A]) {
  override def toString: String = toList.mkString("List(", ", ", ")")

  def foreach(action: A => Unit): Unit = toList.foreach(action)

  def withFilter(predicate: A => Boolean): ListWrapper[A] =
     new ListWrapper(toList.filter(predicate))

  def map[B](function: A => B): ListWrapper[B] = new ListWrapper(toList.map(function))

  def ++[B >: A](tail: ListWrapper[B]): ListWrapper[B] =
     new ListWrapper[B](toList ++ tail.toList)

  def :::[B >: A](prefix: ListWrapper[B]): ListWrapper[B] =
     new ListWrapper[B](toList ::: prefix.toList)

  def ::[B >: A](head: B): ListWrapper[B] = new ListWrapper(head :: toList)
}

object ListWrapper {
  def apply[T](elements: T*): ListWrapper[T] = new ListWrapper(elements.toList)

  def unapplySeq[T](list: ListWrapper[T]): Option[Seq[T]] = Some(list.toList)

  implicit def wrapper2list[T](list: ListWrapper[T]): List[T] = list.toList
}

object Modulo2 {
  def unapply(int: Int) = int % 2 match {
    case 0 => Some(0)
    case _ => None
  }
}

object isEven { def unapply(int: Int) = int % 2 == 0 }

val list = ListWrapper(1, 2, 3)
for {
  element <- list
  isEven() <- Some(element)
  // if element % 2 == 0
} println(s"$element % 2 = 0")

val ListWrapper(first, second, tail@_*) = list
list ++ ListWrapper(1) // list.++(ListWrapper(1))
1 :: list // list.::(1)
(0 :: list) ::: list // list.:::(0 :: list)

import scala.collection.JavaConverters._

val scalaList: List[Int] = list
val javaList: java.util.List[Int] = scalaList.asJava
```

# 4. Еще пачка фишек в Scala

Заведем класс "комплексное число". Использование `case class` дает нам автоматические `equals()`, `hashCode()`, `toString()`, `apply()`, `thenApply()`.

```
final case class ComplexNumber(var x: Double = 0, var y: Double = 0)

object ComplexNumber {
  val Zero = ComplexNumber() // ComplexNumber.apply()
  val One = ComplexNumber(1) // ComplexNumber.apply(1)
  val I = ComplexNumber(y = 1) // ComplexNumber.apply(y = 1)
}
```

Теперь можем использовать наш класс из Scala кода. Можем сразу красиво напечатать.

```
val number = ComplexNumber(1, 1)
println(number) // ComplexNumber(1.0,1.0)
```

На Java пришлось бы писать сложнее.

```
System.out.println("ComplexNumber(" + number.x + "," + number.y + ")");
```

## 4.1. Интерполяция строк

Если хотим вручную написать красивый вывод на Scala, нам на помощь придут интерполированные строки.

```
System.out.println(s"ComplexNumber(${number.x},${number.y})")
```

В отличие от Котлина, в Scala чтобы включить такую интерполяцию, надо перед строкой написать символ `s`.

Стандартных интерполяторов в Scala несколько – `s`, `f`, `raw`. Подробнее про них здесь. Также можно определять свои.

## 4.2. `XML`

С самой первой версии Scala поддерживала написание `XML` тегов в коде программы. Зачем это и как это использовать, написано здесь. Это осложняло парсинг, доставляло проблемы при реализации новых фич, и в какой-то момент добавили новый строковый интерполятор `xml`, чтобы теги писать не прямо в коде, а в виде строковых литералов.

## 4.3. Перегрузка операторов

Введем для наших комплексных чисел оператор `+`.

```
final case class ComplexNumber(var x: Double = 0, var y: Double = 0) {
   def +(number: ComplexNumber): ComplexNumber = {
     val ComplexNumber(x1, y1) = number
     ComplexNumber(x + x1, y + y1)
   }
}

println(ComplexNumber.Zero + ComplexNumber(1, 1))
```

Такая инфиксная запись противоречит конвенциям Java. На самом деле такая запись превращается в нормальный вызов метода с названием `+`.

```
ComplexNumber.Zero.+(ComplexNumber(1, 1))
```

Инфиксная запись работает для любых методов и часто используется в коллекциях с `filter`, `map`, итд. Однако сейчас мода на инфиксную запись уходит, так как, например, часто приходится расставлять скобки.

Определим оператор `+=`.

```
def +=(number: ComplexNumber): this.type = { // this.type может вернуть только this
    x += number.x
    y += number.y
    this
}
```

Но теперь мы можем изменять константы, что плохо.

```
ComplexNumer.One += ComplecNumber(1, 1)
```

Поэтому в Scala принято всё писать максимально немутабельным.

## 4.4. Вызов операторов из Java

В Java метод не может называться `+`, поэтому при компиляции этому методу присваивается другое название. Посмотрим, как вообще вызывать те или иные фишки у Scala классов.

Начнем с `apply()`.

```
ComplexNumber number = ComplexNumber.apply(0, 0);
```

Тут всё хорошо, но у нас бывает `apply` с параметрами по-умолчанию. Для них генерируются специальные геттеры.

```
ComplexNumber number = ComplexNumber.apply(C
     omplexNumber.apply$default$1(), ComplexNumber.apply$default$2()
);
```

Теперь вспомним, что в `object ComplexNumber` у нас есть константа `Zero`. Сложим его с другим числом.

```
ComplexNumber.Zero().$plus(new ComplexNumber(1, 1));
```

Видим, что оператор превратился в метод с названием `$plus`. Оператор `+=` превратится в `$plus$eq()`.

## 4.5. Во что превращается `object`

В Scala нет ничего статического, всё, что имитирует статику, находится в соответствующих `companion`-объектах. Эти объекты компилируются в отдельные классы. То есть когда мы из Java обращались к константам, мы по-хорошему должны были писать так.

```
ComplexNumber number = ComplexNumber$.MODULE$.apply(C
     omplexNumber$.MODULE$.apply$default$1(),C
     omplexNumber$.MODULE$.apply$default$2()
);

ComplexNumber$.MODULE$.Zero;
```

Но современная Scala генерирует более удобные методы, чтобы можно было писать как в первом примере.

## 4.6. Свойства

В Scala, как и в Котлине, нет полей в понимании Java. В место них сразу генерируются геттеры и сеттеры. В Java геттер выглядит просто как метод с таким же названием как у переменной, а к названию сеттера в конце приписывается `_$eq`.

```
number.x_$eq(number.y()); // number.x = number.y
```

## 4.7. `apply()` и `update()`

С методом `apply()` мы уже встречались, когда писали его в объекте и вызывали в нем конструктор. Его также можно определять в классе. Сделаем, чтобы `apply()` возвращал вещественную либо мнимую часть числа.

```
final case class ComplexNumber(var x: Double = 0, var y: Double = 0) {
   def apply(real: Boolean): Double = if (real) x else y
}
. . .
println(I(true)) // 0
println(I(false)) // 1
```

Более интересный метод – `update()`. Если он будет принимать два параметра, то можно будет его использовать со знаком присваивания. Сделаем `apply()`, который прибавляет аргумент к одному из компонентов.

```
final case class ComplexNumber(var x: Double = 0, var y: Double = 0) {
   def update(real: Boolean, value: Double): Double =
      if (real) x += value else y += value
}
. . .
I(true) = 1
I(false) = 1
println(I) // ComplexNumber(1.0,2.0)
```

Теперь сделаем следующий вызов.

```
println(I(false)) // 1.0
I(false) += 1
println(I(false)) // 3.0
```

Второй раз напечаталось `3.0`, потому что на самом деле `+=` развернется в следующий код и произойдет два прибавления.

```
I(false) += 1
// =>
I(false) = I(false) + 1
// =>
I.update(false, I.apply(false) + 1)
```

## 4.8. Юникод, бессмысленный и беспощадный

Scala поддерживает символы юникода. Это дает возможность писать русскоязычные названия полей и методов. Также можно все стрелочки (`->`, `=>`) заменить на аналогичные символы $\rightarrow$, $\Rightarrow$. А также можно писать всякие бессмысленные и беспощадные штуки.

```
package object trash {
  val нет: Any = null
  val !!! : Any = null
  object B{
    def Scala(o: Any) = new {
      def полей(x: Any): Unit = {}
    }
  }
}
B Scala нет полей !!!
```

# 5. ООП фишки в Scala

## 5.1. Множественное наследование

Множественное наследование интерфейсов в Scala есть. Родительский класс может быть только один. Отличие от Java – дефолтные реализации. В современной Scala они такие же как в современной Java с точки зрения байт кода, но другие с точки зрения использования.

### 5.1.1. Линеаризация

Линеаризация – это процесс объединения (смешивания) интерфейсов при множественном наследовании. Чтобы понять, в каком порядке происходит линеаризация в Scala, рассмотрим следующий пример.

```
1  object Linearization {
2
3    trait Foo {
4     print(''1'')
5    }
6
7    trait Bar extends Foo {
8     print(''2'')
9    }
10
11   trait Baz extends Bar {
12    print(''3'')
13   }
14
15   trait BarBaz extends Foo {
16    print(''4'')
17   }
18
19   def main(args: Array[String]): Unit = {
20     val foo = new Foo with BarBaz with Baz with Bar
21   }
22 }
```

Данный код выведет 1423. Из этого нетрудно догадаться, что интерфейсы обходятся слева направо поиском вглубину. При этом если интерфейс уже добавлен и проинициализирован, второй раз он не добавляется.

### 5.1.2. Смешивание (mix in)

Рассмотрим, что происходит с методами при смешивании. Если добавляемые члены разных интерфейсов не пересекаются по сигнатуре, они просто все добавятся, их можно будет вызывать на сконструированном объекте. Если же они пересекаются, надо писать ключевое слово `override`. Перепишем наш пример.

```
object Linearization {
  trait Foo {
    def foo = ''1''
  }

  trait Bar extends Foo {
    override def foo: String = ''2'' + super.foo
  }

  trait Baz extends Bar {
    override def foo: String = ''3'' + super.foo
  }

  trait BarBaz extends Foo {
    override def foo: String = ''4'' + super.foo
  }

  def main(args: Array[String]): Unit = {
    val foo = new Foo with BarBaz with Baz with Bar {
      override def foo: String = super.foo
    }
    print(foo.foo) // 3241
  }
}
```

Тут всё выведется в обратном порядке, потому что мы каждый раз сначала обращаемся к методу предка. Помимо привычного синтаксиса `super.foo`, можно явно указать, у какого предка мы хотим вызвать метод: `super[Foo].foo`.

Если слово `override` не написать, соответствующий метод или свойство перекроется. При этом важно понимать, что при перекрытии порядок, в котором записаны трейты, важен (то есть with не коммутативен)

В scala 3.0 с этим поборолись. Появятся новые операторы.

```
val foo: Bar & Foo // пересечение (коммутативно)

val foo: Bar | Foo // объединение (коммутативно)
```

Пересечение позволит вызывать только те методы, которые есть и там, и там. Объединение – все методы, без перекрытий. С помощью объединения можно реализовать Nullable типы, как в Котлине.

```
val bar: String | Null
```

## 5.2. Инициализация

Какие еще проблемы могут возникнуть при наследовании? Рассмотрим пример.

```
trait Initialization {
  val text: String
  val textLength: Int
}

object Initialization {

  def main(args: Array[String]): Unit = {
    Bar
  }

  class Foo extends Initialization {
    val text = ''foo''
    val textLength = text.length
  }

  object Bar extends Foo {
    override val text: String = ''foobar''
  }
}
```

Пытаемся инстанцировать `Bar`. Получаем либо `NullPointerException`, либо `ExceptionInInitializerError`. Это происходит потому что в строчке `val textLength = text.length` текст, который должен быть проинициализирован в `Bar`, еще не проинициализировался, то есть он равен `null`. Для решения такой проблемы есть два способа.

### 5.2.1. Ленивые переменные

Если к `val` переменной или свойству приписать `lazy`, инициализация проихойдет не сразу, а при первом обращении. Это элегантно, позволяет удобно писать, но есть много сложностей. Возникает множество проблем с многопоточностью, можно получить дедлок на инициализации. С этим настолько много проблем, что их стали решать в компиляторе.

Еще одна деталь. Слово `lazy` необходимо писать при `override`, даже если в предке это свойство уже ленивое.

### 5.2.2. Early definitions

Такая штука не очень элегантная. В Scala 3.0 трейтам можно будет задавать параметры. Что-то вроде первичных конструкторов у классов. Это позволит передавать то что надо переопределить, если что-то должно пересечься. Это также решает проблему дедлоков.

## 5.3. Cake pattern

В чем суть этого шаблона. Пусть в системе есть компоненты `Foo` и `Bar`. Надо сконструировать `Baz` с их использованием. Давайте вместо наследования скажем, что наш интерфейс зависит от интерфейсов `FooComponent` и `BarComponent`, и для этих компонентов заведем реализации. Теперь при инстанцировании `Baz` необходимо явно указывать реализации компонентов, и мы гарантированно не забудем ничего проинициализировать.

```
trait FooComponent {
  val foo: Foo
  protected trait Foo {
    def foo: String
  }
}

trait FooComponentImpl extends FooComponent {
  override val foo: Foo = new FooImpl
  private class FooImpl extends Foo {
    override def foo: String = ''foo''
  }
}

trait BarComponent {
  val bar: Bar
  protected trait Bar {
    def bar: String
  }
}

trait BarComponentImpl extends BarComponent {
  override val bar: Bar = new BarImpl
  private class BarImpl extends Bar {
    override def bar: String = ''bar''
  }
}

trait Baz {
  this: FooComponent with BarComponent => // this можно заменить на self
  def baz: String = foo.foo + bar.bar
}

def main(args: Array[String]): Unit = {
  val baz = new Baz with FooComponentImpl with BarComponentImpl
  println(baz.baz)
}
```

Примерно так это было на лекции. Проще и понятнее это написано, например, здесь.

## 5.4. Try with resources

Так как Scala начала создаваться раньше Java 7, try with resources в ней нет. Чтобы правильно прочитать файл, приходится писать всякий треш, как у нас был в летучке по Java.

Попробуем немного исправить ситуацию. Хотим написать функцию, которая сама будет открывать и закрывать файлы и ловить ошибки, а задавать ее поведение мы будем, передавая функции в качестве аргументов.

```
def withClose[C <: AutoCloseable, R](init: => C) // by name parameter
                                    (action: C => R)
                                    (recover: PartialFunction[Throwable, R]): R = {
  var closeable: C = null.asInstanceOf[C]
  try {
    closeable = init // вычисление init
    action(closeable)
  } catch {
    recover
  } finally {
    if (closeable != null) {
      try {
        closeable.close()
      } catch {
        case _: Exception =>
      }
    }
  }
}
```

Что здесь происходит? Принимаем на вход ресурс (`init`), функцию, описывающую содержательные действия (`action`) и функцию обработки ошибок (`recover`).

Во-первых, заметим, что `init` имеет странный тип `=> C`. Это значит, что параметр передается "by name". Это значит, что если передать туда некоторое выражение, например `new File("file.txt")`, то оно вычислится не перед тем как попасть в функцию, а во время использования. Здесь это позволит нам обернуть открытие ресурса в блок `try-catch`.

Во-вторых, `recover` – это `PartialFunction`, что позволит внутри сопоставлять исключение с образцом, как в `catch` блоке.

Теперь научимся это использовать.

```
withClose(new BufferedReader(new FileReader("file.txt"))) { reader =>
  var line = reader.readLine()
  while (line != null) {
    println(line)
    line = reader.readLine()
  }
} {
  case e: FileNotFoundException =>
    println(e.getMessage)
  case e: IOException =>
    for {
      exception <- e +: e.getSuppressed
      message = exception.getMessage
    } println(message)
}
```

По синтаксису стало больше похоже на try with resources, и не так нагроможденно.

# 6. Implicit

Implicit – это одна из самых мощных возможностей языка. Это набор фич, который работает еще с другими фичами. Появилась эта штука еще до Scala, но распространилась именно здесь. На нее ругаются, потому что код становится менее читаемым, но иногда получается писать прикольные вещи. Implicit надо использовать аккуратно, понимать, к чему ведет.

## 6.1. Отступление про пакеты и импорты

Прежде чем начать, сделаем несколько замечаний по поводу пакетов и импортов в Scala.

Во-первых, задание пакетов можно делить на несколько выражений. Код ниже будет эквивалентен выражению `package ru.spbau.jvm.scala.a`.

```
1 package ru.spbau.jvm
2 package scala
3 package a
```

Во-вторых, импорты можно писать в любом месте кода. Соответственно, их можно изолировать в различные области видимости (тело функции, тело цикла, просто блок `{...}`, и т.п.)

Наконец, импорты можно именовать. Пусть есть класс `Token`, который лежит в пакете `a`. Пусть также есть другой класс, который тоже называется Token и лежит в пакете `b`. Чтобы было удобно работать с этими классами в одном коде, можно их заимпортировать с разными имнами.

```
1 import a.{Token => AToken}
2 import b.{Token => BToken}
3
4 val atoken = new AToken()
5 val btoken = new BToken()
```

С импортами разобрались, можно переходить к implicit. Неявными бывают параметры, классы и методы. Поговорим обо всём по порядку.

## 6.2. Implicit функции и неявные преобразования

Неявные функции вызываются автоматически в тех местах, где без них не обойтись.

Для примера вернемся к вышеупомянутым токенам. Эти классы немного разные и лежат в разных пакетах, но нам хочется работать с ними так, будто это одно и то же.

Например, мы умеем печатать `a.Token`.

```
1 package a
2
3 case class Token(text: String)
4
5 object Token {
6   def printToken(token: Token): Unit = println(token)
7 }
8
```

Но напечатать `b.Token` пока что не получится.

```
import a.{Token => AToken}
import b.{Token => BToken}
import a.Token // заимпортили object Token

def main() {
   val atoken = new AToken("a.Token")
   Token.printToken(atoken) // OK, напечатали
   val btoken = new BToken("b.Token")
   Token.printToken(btoken) // Ошибка, тип не совпадает.
}
```

Напишем для `b.Token` неявную функцию преобразования к `a.Token`.

```
package b

case class Token(text: String)

object Token {
   implicit def bToA(token: Token): a.Token = new a.Token(token.text)
}
```

Теперь в строчку, которая раньше не компилировалась, неявно подставится вызов функции `b.Token.bToA()`, и всё заработает.

```
def main() {
   val btoken = BToken("b.Token")
   Token.printToken(btoken) // заменится на Token.printToken(b.Token.bToA(btoken))
}
```

В `IDEA 2018.3` можно нажать View → Show Implicit Hints, и редактор будет прямо в коде показывать серым цветом неявные вызовы.

Важно помнить, что компилятор может сделать не более одного неявного преобразования. То есть если есть преобразования `AToken => BToken` и `BToken => CToken`, то неявно преобразовать `AToken => CToken` не получится.

## 6.3. Implicit классы и extension методы

Как мы помним, в Котлине есть методы-расширения (extension methods). Их можно использовать, чтобы расширять возможности работы с библиотечными классами, которые мы не можем изменять напрямую, например `String`, `Int`, и т.п.

В Scala похожая фишка реализована с помощью неявных классов. Напишем extension метод, который печатает `Token`.

```
implicit class TokenExt(private val token: Token) {
   def print(): Unit = println(token)
}
```

Вот так бы выглядело использование, если бы не было слова `implicit`.

```
val token = new Token("token")
val tokenExt = new TokenExt(token)
tokenExt.print()
```

С `implicit` же можно гораздо проще.

```
val token = new Token(”token”)
token.print()
```

При этом надо понимать, что в любом случае будет создаваться лишний объект, а потом удаляться. Хотим, чтобы вместо этого вызывалась статическая функция. Для этого надо отнаследовать `TokenExt` от `AnyVal`. Это приведет к определенным ограничениям на класс `TokenExt`, например, внутри него нельзя будет хранить состояние (`val`, `var`). Подробнее про `AnyVal` читайте в документации.

### 6.3.1. package object

Возникает вопрос – где определять расширения? Определять можно где угодно, но обычно их пишут внутри companion object или package object.

В Котлине можно писать код прямо в файле, вне всяких классов. В Scala вместо этого можно писать package object. Всё что объявлено внутри такого объекта, доступно во всем пакете.

```
package ru.spbau.jvm.scala

package object lecture07 { ... }
```

Даже если есть свой класс, для него тоже хорошо писать расширения, потому что это позволит не слишком расширять публичный интерфейс. Мы иногда хотим писать статические методы, которые обслуживают наш класс. Вместо этого extension методы более удобные. Но на эту тему тоже есть жесткие споры.

## 6.4. Implicit параметры и переменные

Как мы помним, у метода может быть несколько parameter scope’ов.

```
def foo(a: Int)(b: Double) = ???
```

В текущей версии Scala максимум один scope может быть помечен словом `implicit`. Все параметры в этом scope будут implicit. В Scala 3.0 можно будет писать сколько угодно implicit scope’ов.

```
def foo(i: Int)(implicit n: Int, k: Long) = ??? // n, k будут implicit
```

Вернемся к нашим токенам. В качестве параметра можно принять, например, функцию, которая преобразует `T => a.Token`. Тогда внутри сможем печатать любой тип.

```
def printTToken[T](token: T)(implicit tToA: T => AToken) = {
   Token.printToken(token) // превратится в Token.printToken(tToA(token))
}
```

Переменные и свойства тоже могут быть implicit. Компилятор будет пытаться их подставить вместо недостающих параметров при вызове функции. Если для подстановки не найдется ни одного подходящего implicit’а или найдется больше одного подходящего, будет ошибка компиляции.

Далее на паре был пример использования всего этого, но на него меня не хватило `):`

# 7. Продвинутый pattern matching и экстракторы

## 7.1. Регулярные выражения

Создать регулярное выражение можно прямо из строки с помощью extention метода `r`. Вспомним, что в Scala, если писать строку в тройных кавычках, то не надо будет экранировать слэши, и писать регулярки будет чуть менее больно. Напишем регулярку для десятичной дроби.

```
val decimal = """(-)?(\d*)(\d+)?""".r
```

Теперь мы можем применить нашу регулярку в pattrn matching'е.

```
"-123.45" match {
  case decimal(null, int, frac) => println(s"$int + 0.$frac")
  case decimal(_, int, frac) => println(s"-$int + 0.$frac")
  case _ => println("error")
}
```

Работает это следующим образом. У `decimal` создался экстрактор, который в случае успешного разбора разбивает строку на блоки, соответствующие скобочкам в регулярке. Если какой-то блок оказался пустым (например, у числа нет знака), соответствующий компонент будет равен `null`.

Также можно просто завести переменные, содержащие нужные куски. Синтаксис похож на кортежи в Котлине.

```
val decimal(_, two, _) = "-2.0"
println(two) // 2
```

Наконец, можно найти все совпадения в строке и проитерироваться по ним с помощью for comprehension.

```
for (Decimal(sign, i, f) <- Decimal.findAllIn("1.0 and -2.0, 33")) {
   print(s"$sign$i$f ") // null1.0 -2.0 null33null
}
```

## 7.2. Инфиксные паттерны

В скале есть type alias'ы (псевдонимы для типов), они очень мощные. Рассказывать нам про них, конечно, не стали.

```
type ++[LeftT, RightT] = (LefT, RightT)

val tuple: (Int, Int) = (1, 2)
// эквивалентно
val tuple: Int ++ Int = (1, 2)
```

Что здесь произошло? Мы сказали, что теперь тип `++` является синонимом типа `( , )`.

Теперь можем для своего синонима завести отдельный экстрактор. Для этого разобьем пару с помощью экстрактора `tuple` и вернем получившиеся компоненты.

```
object ++ {
    def unapply[LeftT, RightT](pair: LeftT ++ RightT): Option[LeftT ++ RightT] = {
      val (first, second) = pair
      Some(first, second)
    }
}
```

Теперь, когда у нас есть свой экстрактор, мы можем использовать его в pattern matching'е. Причем теперь можно писать паттерн инфиксно.

```
(42, 42) match {
    case left ++ right => println(left + right) // аналогично case ++(left, right) =>
  }
```

Но при задании переменных инфиксная запись пока не работает. Чтобы она работала, придется задать extension оператор ++ для любого типа.

```
implicit class Ext[LeftT](private val left: LeftT) extends AnyVal {
    def ++[RightT](right: RightT): (LeftT, RightT) = (left, right)
}

val intAndString = 42 ++ ''42'' // аналогично val intAndString = (42, ''42'')
```

## 7.3. Что возвращает `unapply()`

Мы привыкли, что экстрактор возвращает `Optional`, содержащий компоненты в случае успеха и пустой в противном случае. На самом же деле он может вернуть любой класс, у которого есть методы `isEmpty()`, проверяющий, что содержимое есть, и `get()`, который возвращает то, что мы потом будем сопоставлять.

Но для полного счастья нам не хватает методов, позволяющих вытаскивать значения из нашей обертки. Они называются как же как и в `tuple`'ах: `_1`, `_2`, `_3`, ...

```
class Extractable(string: String) {
  def _1: Int = string.length
  def _2: String = string
  def isEmpty: Boolean = false
  def get: Extractable = this
}

object Extractable {
  def unapply(string: String): Extractable = new Extractable(string)
  def main(args: Array[String]): Unit = args match {
    case Array(Extractable(length, _)) => println(length)
    case _ =>
  }
}
```

# 8. Type-level programming

## 8.1. Path dependent types

Минутка алгоритмов. Пусть у нас есть граф. У него есть внутренние классы – вершины и ребра. Хранятся эти вершины и ребра в списках, в которые можно добавлять элементы с помощью оператора +=.

```
class Graph {

  case class Vertex()

  type Edge = (Vertex, Vertex)

  private val vertices = mutable.ArrayBuffer.empty[Vertex]
  private val edges = mutable.ArrayBuffer.empty[Edge]

  def +=(vertex: Vertex): Unit = {
    vertices += vertex
  }

  def +=(edge: Edge): Unit = {
    edges += edge
  }
}
```

Попробуем воспользоваться этим классом.

```
// создаем граф
val firstGraph = new Graph
// создаем две вершины
val fromVertex = firstGraph.Vertex()
val toVertex = firstGraph.Vertex()
// добавляем вершины в граф
firstGraph += fromVertex
firstGraph += toVertex
// добавляем ребро в граф
firstGraph += (fromVertex, toVertex)
```

Всё хорошо. Теперь создадим второй граф и попробуем в него добавить те же самые вершины и ребра.

```
val secondGraph = new Graph
secondGraph += fromVertex // ошибка
secondGraph += (fromVertex, toVertex) // ошибка
```

У нас ничего не получится, потому что в Scala типы внутренних классов зависят от конкретного экземпляра внешнего класса, используемого при инстанцировании.

Если по-русски, то у `toVertex` и `fromVertex` будет тип `firstGraph.Vertex`, а во второй граф можно добавлять только вершины типа `secondGraph.Vertex`. Аналогично с ребрами.

Что же делать, если хотим функцию, принимающую любые вершины или ребра? Для этого надо указать тип аргумента не `Graph.Edge`, а `Graph#Edge`. Решетка показывает, что нам не важно, какому именно графу принадлежит ребро. Теперь можем написать функцию, выводящую ребро в консоль.

```scala
def printEdge(vertex: Graph#Edge): Unit = {
    val (from, to) = vertex
    println(s"$from -> $to")
}
```

## 8.2. Гетерогенные списки

Гетерогенный список позволяет хранить элементы разных типов, при этом информация о типе каждого элемента тоже сохраняется.

Задать такой список можно рекурсивно, с помощью case классов.

```scala
sealed trait HList

object HList {
  case class HCons[+H, +T <: HList](head: H, tail: T) extends HList
  case object HNil extends HList
}

val list = HCons(1, HCons("hello!", HNil))
val str: String = list.tail.head
```

Пока всё просто. Теперь научимся объявлять список более красиво, реализовав оператор `::`

```scala
implicit class HListExt[R <: HList](private val list: R) extends AnyVal {
  def ::[H](head: H) = HCons(head, list)
}

val list = 1 :: "asd" :: HNil
```

Кажется, что здесь произошла магия, ведь мы сделали расширение для списка, а слева от первого `::` стоит `Int`. На самом деле оператор `::` является правоассоциативным, и у него справа будет стоять объект, от которого он вызывается, а слева – аргумент. То есть наша запись развернется в следующую.

```scala
val list = HNil.::("asd").::(1)
```

Едем дальше. Хотим склеивать списки. И тут нам придется выйти на новый уровень отношений со Scala – type level.

Начнем с определения самого оператора. Он будет принимать в качестве implicit аргумента trait, который будет нам всё склеивать, и вызывать у него `apply`, в котором всё и будет происходить.

```scala
def :::[L <: HList, Result <: HList](left: L)
                                    (implicit appendable: Appendable[L, R, Result]): Result =
  appendable(left, list)
```

А теперь приступим к реализации самого `Appendable`.

```scala
trait Appendable[L <: HList, R <: HList, Result <: HList] {
  def apply(left: L, right: R): Result
}
```

В качестве типовых параметров `Appendable` принимает тип левого списка, тип правого списка и тип результата. Вспоминаем, что запись `L <: HList` означает, что тип `L` является наследником типа `HList` или им самим.

А теперь – самое интересное. Мы, как в Haskell, по индукции, определим операцию склеивания.

```
object Appendable {
  import HList._

  implicit def nilAppendable[R <: HList]: Appendable[HNil.type, R, R] =
    new Appendable[HNil.type, R, R] {
      override def apply(left: HNil.type, right: R) = right
    }

  implicit def appendable[L <: HList, R <: HList, H, Result <: HList]
                         (implicit appendable: Appendable[L, R, Result]):
                         Appendable[HCons[H, L], R, HCons[H, Result]] =
    new Appendable[HCons[H, L], R, HCons[H, Result]] {
      override def apply(left: HCons[H, L], right: R) =
        HCons(left.head, appendable(left.tail, right))
    }
}
```

Не торопитесь падать в обморок. Сейчас во всём разберемся. Для начала поймем, что происходит в `apply()`. Он принимает на вход два списка и должен вернуть третий. Так как `Appendable` является trait'ом, и `apply()` реализации не имеет, компилятор пытается найти функцию, которая бы вернула подходящую реализацию. Здесь представлено две такие функции.

Функция `nilAppendable()` возвращает тип `Appendable[HNil.type, R, R]`, то есть такую реализацию, в которой метод `apply()` в качестве левого списка принимает `HNil` и возвращает правый список. Теперь мы можем склеивать пустой список с любым другим. Это будет база нашей индукции.

Функция `appendable()` возвращает такую реализацию `Appendable`, которая принимает первым параметром **не**пустой список. Метод `apply()` в этой реализации будет возвращать список, состоящий из `left.head` и склеенных `left.tail` и `right`. Но чтобы склеить `left.tail` и `right`, потребуется уже другая реализация `Appendable`. Эту реализацию наша функция получает в качестве implicit аргумента и рекурсивно выбирает функцию, которая ее вернет.

Напоследок заменим анонимные классы на лямбды, чтобы выглядело всё не так страшно.

```
implicit def nilAppendable[R <: HList]: Appendable[HNil.type, R, R] =
  (left: HNil.type, right: R) => right

implicit def appendable[L <: HList, R <: HList, H, Result <: HList]
                       (implicit appendable: Appendable[L, R, Result]):
                       Appendable[HCons[H, L], R, HCons[H, Result]] =
  (left: HCons[H, L], right: R) => HCons(left.head, appendable(left.tail, right))
```

Прелесть всего этого ужаса заключается в том, что все вычисления происходят еще на этапе компиляции, когда мы знаем типы всех элементов списка. И так как для разных типов создаются разные реализации `Appendeble`, они-то и позволяют нам помнить на этапе выполнения, какой тип имеет каждый элемент.

## 8.3. Абстрактные типы

Эту тему совсем чуть-чуть затронули на последней лекции. Краткое описание здесь.

## 8.4. Продвинутые ограничения в generic типах

Это тоже немного было на последней лекции. Читаем здесь.