

Scala

Василий Купоросов

4 декабря 2018 г.

Содержание

1. Основы	1
1.1 Немного истории	1
1.2 Классы	1
1.2.1 <code>class</code>	1
1.2.2 <code>trait</code>	1
1.2.3 <code>object</code>	2
1.3 Наследование	2
1.3.1 Наследование в <code>generic</code> 'ах	2
1.4 Оператор <code>new</code> и функциональный метод <code>apply()</code>	2
1.5 Скобки	3
1.5.1 Писать – не писать	3
1.5.2 Скобки для <code>generic</code> 'ов	3
1.6 Массивы и доступ к элементам	3
1.7 Модификаторы доступа	4
1.8 <code>Properties</code>	4
1.9 <code>Hello world!</code>	4
1.10 Пакеты, <code>import</code> 'ы и коллизии	5
1.11 Избавляемся от лишнего	5
1.12 Рекурсия	5
1.13 Метод в методе	6
1.14 <code>try</code>	6
1.15 Алгебраические типы данных	6
1.16 <code>case</code> классы	7
2. Функциональное программирование на Scala	8
2.1 Функциональные типы	8
2.2 Частичное применение	8
2.3 Каррирование	9
2.3.1 <code>Placeholder</code>	10

2.3.2	Parameter scope	10
2.4	Вариантность	10
2.5	Кортежи	11
3.	Операции над коллекциями	12
4.	Еще пачка фишек в Scala	13
4.1	Интерполяция строк	13
4.2	XML	13
4.3	Перегрузка операторов	13
4.4	Вызов операторов из Java	14
4.5	Во что превращается <code>object</code>	15
4.6	Свойства	15
4.7	<code>apply()</code> и <code>update()</code>	15
4.8	Юникод, бессмысленный и беспощадный	16
5.	ООП фишки в Scala	17
5.1	Множественное наследование	17
5.1.1	Линеваризация	17
5.1.2	Смешивание (<code>mix in</code>)	18
5.2	Инициализация	19
5.2.1	Ленивые переменные	19
5.2.2	Early definitions	19
5.3	Cake pattern	20
5.4	Try with resources	21

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.

```
1 | class Foo {  
2 |     val bar: Int = 42  
3 |     def foo(): Int = bar + 1  
4 | }
```

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

```
1 | class Bar(innerFoo: Int) {  
2 |     this(doubleFoo: Double) = // побочный конструктор  
3 |     this(doubleFoo.toInt)    // вызывает примарный  
4 |  
5 |     def foo(): Int = innerFoo // используем значение из примерного конструктора  
6 | }
```

1.2.2. trait

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

```
1 | trait Bar {  
2 |     val bar: Double  
3 |     def foo(): Int = 42  
4 | }
```

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.

```
1 | object Foo // синглтон Foo
2 | class Bar // класс Bar
3 | 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.

```
1 | class Foo
2 | trait Bar
3 | trait Baz
4 |
5 | class MyClass extends Foo with Bar with Baz
```

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

```
1 | class Foo extends Bar {
2 |   override val bar: Double = 42
3 |   override def foo(): Int = bar.toInt
4 | }
```

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).

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

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

```
1 | 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 – создается 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()`.

```

1 | class Bar
2 | object Bar {
3 |   def apply(): Bar = new Bar
4 | }
5 |
6 | val bar = Bar() // теперь можем не писать new
7 | 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.

```

1 | class Bar(foo: Int) { ... }
2 | object Bar {
3 |   def apply(doubleFoo: Double) = new Bar(doubleFoo.toInt)
4 | }
5 |
6 | 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:

```

1 | 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.

```

1 | val array: Array[Int] = Array(3)
2 | array(0) = 42 // array.update(0, 42), присваиваем первый элемент
3 | 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.

```
1 | package ru.spbau.jvm.scala
2 |
3 | class Foo {
4 |     private var foo: Int          // доступ только внутри класса
5 |     private[scala] var bar: Int    // доступ только в пакете ru.spbau.jvm.scala
6 |     private[jvm] var baz: Int     // доступ только в пакете ru.spbau.jvm
7 |     private[this] var bus: Int    // доступ только внутри экземпляра
8 |
9 |     def fun(other: Foo): Unit = {
10 |         bus = 42                // ок, доступ изнутри экземпляра
11 |         other.bus = 42          // ошибка, доступ из другого экземпляра
12 |     }
13 | }
```

Примарный конструктор тоже можно сделать защищенным или приватным

```
1 | class Foo protected()
2 | class Bar private()
3 | 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`.

```
1 | object HelloWorld {
2 |     def main(args: Array[String]): Unit = {
3 |         println("Hello world!")
4 |     }
5 | }
```

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

```
1 | class HelloWorld extends App {
2 |     println("Hello world!")
3 | }
```

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 заимпортировать его. Но при импорте Scala'вский `List` перекроется, что не очень хорошо. Поэтому можно импортировать прямо пакеты.

```
1 | import java.util
2 | 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.

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

1.11. Избавляемся от лишнего

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

```
1 | def foo(x: Boolean): Int = {
2 |   if (x) return 42
3 |   return 43
4 | }
```

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.

```
1 | def foo(x: Boolean): Int = {
2 |   if (x) 42 else 43
3 | }
```

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

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

1.12. Рекурсия

Напишем функцию, считающую факториал числа.

```
1 | def fac(n: Int): BigInt =
2 |   if (n == 1) 1
3 |   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.

```
1 | @tailrec
2 | def facTailRec(n: Int, accumulator: BigInt = 1): BigInt = n match {
3 |   case 1 => accumulator
4 |   case _ => facTailRec(n - 1, n * accumulator)
5 | }
```

Здесь мы видим сразу две фишки – 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.

```
1 def facTailRec(n: Int): BigInt = {
2   @tailrec
3   def facTailRecInner(n: Int, accumulator: BigInt = 1): BigInt = n match {
4     case 1 => accumulator
5     case _ => facTailRec(n - 1, n * accumulator)
6   }
7
8   facTailRecInner(n)
9 }
```

1.14. try

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

```
1 try {
2   fac(1000000)
3 } catch {
4   case e: StackOverflowError => facTailRec(1000000)
5   case _: IOException | _: FileNotFoundException => ... // комбинируем случаи
6   _: // матчим всё что не сматчилось ранее
7 }
```

TODO: подробнее будет позже, вместе с `match`

1.15. Алгебраические типы данных

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

```
1 sealed trait IntList
2
3 final class IntNil extends IntList
4 final class IntCons(val head: Int, val tail: IntList) extends IntList
```

Теперь можем заводить списки.

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

А потом матчить.

```
1 list match {
2   case cons: IntCons => println(cons.head) // теперь cons = list.asInstanceOf[IntCons]
3   case _: IntNil => // ничего не делаем
4 }
```


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 ключевое 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.

```
1 sealed trait IntList
2
3 case object IntNil extends IntList
4 final case class IntCons(head: Int, tail: IntList) extends IntList
5
6 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.

```
1 def printList(list: IntList): Unit = {
2   list match {
3     case IntNil =>
4     case IntCons(head, tail) =>
5       println(head)
6       printList(tail)
7   }
8 }
```

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.

```
1 || 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.

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

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

```
1 || 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.

```
1 || 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]`.

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

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

```
1 || val sum: (Int, Int) => Int = {  
2 ||   case (left, right) => right + left  
3 || }
```

2.2. Частичное применение

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

```
1 | val weirdSum: PartialFunction[(Int, Int), Int] = {  
2 |   case (0, _) | (_, 0) => 0 // если хотя бы один ноль, то сумма тоже ноль  
3 |   case (left, right) => left + right  
4 | }
```

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

```
1 | val weirdSum: PartialFunction[(Int, Int), Int] = {  
2 |   case (0, _) | (_, 0) => _  
3 | }
```

Если оба аргумента будут ненулевыми, то pattern matching не пройдет, и выпадет ошибка `MatchError`.

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

```
1 | val sum: PartialFunction[(Int, Int), Int] = {  
2 |   case (left, right) => left + right  
3 | }  
4 |  
5 | 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 тьюпинг преобразовал наши аргументы в одну пару и передал в функцию.

То же самое можно написать в виде функции.

```
1 | val sum: (Int, Int) => Int = {  
2 |   case (left, right) => left + right  
3 | }  
4 |  
5 | def sum(left: Int, right: Int): Int = (left, right) match {  
6 |   case (left, right) => left + right  
7 | }  
8 |  
9 | sum.apply(1, 2); // вызовется val sum  
10 | 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.

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

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

```
1 | 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.

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

2.4. Вариантность

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

```
1 | 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.

```
1 | 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`.

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

В Java ему соответствует такая запись.

```
1 | 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.

```
1 | val tuple: (Int, String) = (42, "str")
2 | println(tuple._1)
3 | println(tuple._2)
```

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

```
1 | val (index, text) = (42, "str")
2 | println(index)
3 | println(text)
```

Рассмотрим функцию суммы, принимающая `tuple` из двух слагаемых.

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

В Haskell-стиле правильнее было бы писать так.

```
1 | 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 тьюплов. Там можно распаковывать списки и коллекции, пользуясь похожим на тьюплы 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.

```

1 final class ListWrapper[A](val toList: List[A]) {
2   override def toString: String = toList.mkString("List(", ", ", ")")
3
4   def foreach(action: A => Unit): Unit = toList.foreach(action)
5
6   def withFilter(predicate: A => Boolean): ListWrapper[A] =
7     new ListWrapper(toList.filter(predicate))
8
9   def map[B](function: A => B): ListWrapper[B] = new ListWrapper(toList.map(function))
10
11  def ++[B >: A](tail: ListWrapper[B]): ListWrapper[B] =
12    new ListWrapper[B](toList ++ tail.toList)
13
14  def :::[B >: A](prefix: ListWrapper[B]): ListWrapper[B] =
15    new ListWrapper[B](toList ::: prefix.toList)
16
17  def ::[B >: A](head: B): ListWrapper[B] = new ListWrapper(head :: toList)
18 }
19
20 object ListWrapper {
21   def apply[T](elements: T*): ListWrapper[T] = new ListWrapper(elements.toList)
22
23   def unapplySeq[T](list: ListWrapper[T]): Option[Seq[T]] = Some(list.toList)
24
25   implicit def wrapper2list[T](list: ListWrapper[T]): List[T] = list.toList
26 }
27
28 object Modulo2 {
29   def unapply(int: Int) = int % 2 match {
30     case 0 => Some(0)
31     case _ => None
32   }
33 }
34
35 object isEven { def unapply(int: Int) = int % 2 == 0 }
36
37 val list = ListWrapper(1, 2, 3)
38 for {
39   element <- list
40   isEven() <- Some(element)
41   // if element % 2 == 0
42 } println(s"$element % 2 = 0")
43
44 val ListWrapper(first, second, tail@_*) = list
45 list ++ ListWrapper(1) // list.++(ListWrapper(1))
46 1 :: list // list.::(1)
47 (0 :: list) ::: list // list.:::(0 :: list)
48
49 import scala.collection.JavaConverters._
50
51 val scalaList: List[Int] = list
52 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()`.

```
1 final case class ComplexNumber(var x: Double = 0, var y: Double = 0)
2
3 object ComplexNumber {
4     val Zero = ComplexNumber() // ComplexNumber.apply()
5     val One = ComplexNumber(1) // ComplexNumber.apply(1)
6     val I = ComplexNumber(y = 1) // ComplexNumber.apply(y = 1)
7 }
```

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

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

На Java пришлось бы писать сложнее.

```
1 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.

```
1 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 фиш, 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 `+`.

```
1 final case class ComplexNumber(var x: Double = 0, var y: Double = 0) {
2     def +(number: ComplexNumber): ComplexNumber = {
3         val ComplexNumber(x1, y1) = number
4         ComplexNumber(x + x1, y + y1)
5     }
6 }
7
8 println(ComplexNumber.Zero + ComplexNumber(1, 1))
```

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

```
1 || 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 `+=`.

```
1 || def +=(number: ComplexNumber): this.type = { // this.type может вернуть только this
2 ||     x += number.x
3 ||     y += number.y
4 ||     this
5 || }
```

Но теперь мы можем изменять константы, что плохо.

```
1 || ComplexNumber.One += ComplexNumber(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()`.

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

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

```
1 || ComplexNumber number = ComplexNumber.apply(C
2 ||     omplexNumber.apply$default$1(), ComplexNumber.apply$default$2()
3 || );
```

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

```
1 || 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.

```
1 | ComplexNumber number = ComplexNumber$.MODULE$.apply(C
2 |     omplexNumber$.MODULE$.apply$default$1(),C
3 |     omplexNumber$.MODULE$.apply$default$2()
4 | );
5 |
6 | ComplexNumber$.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 `_set`.

```
1 | number.x_set(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.

```
1 | final case class ComplexNumber(var x: Double = 0, var y: Double = 0) {
2 |     def apply(real: Boolean): Double = if (real) x else y
3 | }
4 | . . .
5 | println(I(true)) // 0
6 | 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.

```
1 | final case class ComplexNumber(var x: Double = 0, var y: Double = 0) {
2 |     def update(real: Boolean, value: Double): Double =
3 |         if (real) x += value else y += value
4 | }
5 | . . .
6 | I(true) = 1
7 | I(false) = 1
8 | println(I) // ComplexNumber(1.0,2.0)
```

Теперь сделаем следующий вызов.

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

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

```
1 | I(false) += 1
2 | // =>
3 | I(false) = I(false) + 1
4 | // =>
5 | 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 (\rightarrow , \Rightarrow) заменить на аналогичные символы \rightarrow , \Rightarrow . А также можно писать всякие бессмысленные и беспощадные штуки.

```
1 | package object trash {
2 |   val нет: Any = null
3 |   val !!! : Any = null
4 |   object B{
5 |     def Scala(o: Any) = new {
6 |       def полей(x: Any): Unit = {}
7 |     }
8 |   }
9 | }
10 | 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.

```

1 object Linearization {
2   trait Foo {
3     def foo = '1'
4   }
5
6   trait Bar extends Foo {
7     override def foo: String = '2' + super.foo
8   }
9
10  trait Baz extends Bar {
11    override def foo: String = '3' + super.foo
12  }
13
14  trait BarBaz extends Foo {
15    override def foo: String = '4' + super.foo
16  }
17
18  def main(args: Array[String]): Unit = {
19    val foo = new Foo with BarBaz with Baz with Bar {
20      override def foo: String = super.foo
21    }
22    print(foo.foo) // 3241
23  }
24 }
```

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.

```

1 | val foo: Bar & Foo // пересечение (коммутативно)
2 |
3 | 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.

```

1 | val bar: String | Null
```

5.2. Инициализация

Какие еще проблемы могут возникнуть при наследовании? Рассмотрим пример.

```
1 | trait Initialization {
2 |   val text: String
3 |   val textLength: Int
4 | }
5 |
6 | object Initialization {
7 |
8 |   def main(args: Array[String]): Unit = {
9 |     Bar
10 |   }
11 |
12 |   class Foo extends Initialization {
13 |     val text = "foo"
14 |     val textLength = text.length
15 |   }
16 |
17 |   object Bar extends Foo {
18 |     override val text: String = "foobar"
19 |   }
20 | }
```

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 пройдет 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.

```
1 trait FooComponent {
2   val foo: Foo
3   protected trait Foo {
4     def foo: String
5   }
6 }
7
8 trait FooComponentImpl extends FooComponent {
9   override val foo: Foo = new FooImpl
10  private class FooImpl extends Foo {
11    override def foo: String = "foo"
12  }
13 }
14
15 trait BarComponent {
16   val bar: Bar
17   protected trait Bar {
18     def bar: String
19   }
20 }
21
22 trait BarComponentImpl extends BarComponent {
23   override val bar: Bar = new BarImpl
24   private class BarImpl extends Bar {
25     override def bar: String = "bar"
26   }
27 }
28
29 trait Baz {
30   this: FooComponent with BarComponent => // this можно заменить на self
31   def baz: String = foo.foo + bar.bar
32 }
33
34 def main(args: Array[String]): Unit = {
35   val baz = new Baz with FooComponentImpl with BarComponentImpl
36   println(baz.baz)
37 }
```

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.

```

1 def withClose[C <: AutoCloseable, R](init: => C) // by name parameter
2                                     (action: C => R)
3                                     (recover: PartialFunction[Throwable, R]): R = {
4   var closeable: C = null.asInstanceOf[C]
5   try {
6     closeable = init // вычисление init
7     action(closeable)
8   } catch {
9     recover
10  } finally {
11    if (closeable != null) {
12      try {
13        closeable.close()
14      } catch {
15        case _: Exception =>
16      }
17    }
18  }
19 }

```

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.

```

1 withClose(new BufferedReader(new FileReader("file.txt"))) { reader =>
2   var line = reader.readLine()
3   while (line != null) {
4     println(line)
5     line = reader.readLine()
6   }
7 } {
8   case e: FileNotFoundException =>
9     println(e.getMessage)
10  case e: IOException =>
11    for {
12      exception <- e +: e.getSuppressed
13      message = exception.getMessage
14    } println(message)
15 }

```

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