Бесплатная электронная книга

# УЧУСЬ

# Scala Language

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#scala

1
1: Scala 2
2
2
Examples 3
Hello World, «» 3
Hello World, 4
4
4
Hello World 5
Scala REPL 6
Scala Quicksheet 7
2: Implicits 9
9
9
Examples 9
9
10
11
«» 12
REPL 12
3: JSON 14
Examples 14
JSON 14
SBT 14
14
JSON 14
JSON 14
DSL 14
- Case 15
15

JSON Circe ..... 16
JSON play-json ..... 16
JSON json4s ..... 19
**4: Quasiquotes** ..... **22**
Examples ..... 22
..... 22
**5: Scala.js** ..... **23**
..... 23
Examples ..... 23
console.log Scala.js ..... 23
..... 23
..... 23
..... 23
DOM ..... 23
SBT ..... 24
Sbt ..... 24
..... 24
: ..... 24
JavaScript: ..... 24
**6: Scaladoc** ..... **25**
..... 25
..... 25
Examples ..... 26
Scaladoc ..... 26
**7: scalaz** ..... **27**
..... 27
Examples ..... 27
ApplyUsage ..... 27
FunctorUsage ..... 27
ArrowUsage ..... 28
**8: Streams** ..... **29**
.....

29

Examples....29

....29

....29

....30

**9: Var, Val Def....31**

....31

Examples....31

Var, Val Def....31

....**31**

....**32**

....**32**

....**33**

Lazy val....33

**«»....34**

Def....35

....35

**10: ....37**

....37

....37

....37

Examples....37

....37

....37

....38

**11: ....40**

....40

....40

....40

Examples....40

- ....40

, ..... 41
12: ..... 43
..... 43
..... 43
..... 43
Examples ..... 43
..... 43
«Do-While» ..... 43
13: ..... 45
Examples ..... 45
Cake Pattern ..... 45
14: ..... 46
Examples ..... 46
..... 46
..... 46
..... 46
..... 47
15: ..... 49
..... 49
..... 49
..... 49
Examples ..... 49
..... 49
..... 50
..... 50
16: ..... 52
..... 52
..... 52
Examples ..... 52
..... 52
..... 52
..... 53

53
for 54
Desugaring 55
17: ( SAM) 56
56
Examples 56
Lambda 56
18: 57
Examples 57
if 57
19: 59
59
Examples 59
Hello String 59
f 59
59
60
61
61
20: 63
63
Examples 63
63
, 63
63
64
64
Currying 65
66
21: 68
68
Examples 68

68

Null 68

70

70

70

22: 72

72

Examples 72

72

: {} vs () 73

Singleton & Companion 74

74

74

75

76

77

77

78

23: 80

80

Examples 80

80

80

82

82

83

84

24: 85

Examples 85

85

, n x 86

86

Cheatsheet 87
88
88
88
88
89
89
Scala 89
90
91
92
93
25: Parser 95
95
Examples 95
95
26: 97
97
Examples 97
97
98
27: 99
99
Examples 99
99
99
, 100
28: 101
101
101
101

Examples....101
-....101
....102
....103
29: ....105
Examples....105
....105
30: Scala....107
Examples....107
Linux dpkg....107
Ubuntu ....107
Mac OSX Macports....108
31: ()....109
....109
....109
Examples....109
....109
....109
32: XML....111
Examples....111
XML....111
33: ....112
Examples....112
....112
....112
....113
....113
getOrElse....113
....113
Java....114
, ....114
try-catch....115

115
34: 116
116
116
Examples 116
( ) 116
116
117
117
117
117
35: Scala 119
Examples 119
119
119
120
36: 122
Examples 122
122
37: 123
123
Examples 123
123
123
124
38: 125
125
Examples 125
125
125
39: 128
Examples 128

Infix 128
128
40: 130
130
Examples 130
Scala Enumeration 130
131
case allValues-macro 132
41: 134
Examples 134
UUF Hive Apache Spark 134
42: 135
Examples 135
135
43: 137
137
137
Examples 137
val vs. var 137
val var 137
138
! 138
« ?» 139
result 139
139
140
140
140
44: 141
Examples 141
141
Gradle Scala 141

142
146
45: 148
Examples 148
148
148
149
150
150
46: 152
152
Examples 152
152
153
47: 154
Examples 154
154
154
155
(scala.util.control.TailCalls) 155
48: 157
157
157
Examples 157
157
49: 159
159
Examples 159
159
50: 161
161

Examples....161
....161
....161
51: Java....162
Examples....162
Scala Java....162
....162
Scala Java....163
Scala - scala-java8-compat....164
52: ....166
....166
....166
Examples....166
....166
....167
Seq....168
( )....169
case....170
....170
....170
....171
(@)....171
....172
, tablewitch lookupswitch....173
....174
....175
53: ScalaCheck....176
....176
Examples....176
Scalacheck scalatest ....176
54: ScalaTest....179
Examples....179

Hello World Spec Test ..... 179
Cheatsheet ..... 179
ScalaTest SBT ..... 180
55: (Generics) ..... 181
Examples ..... 181
..... 181
..... 181
..... 182
..... 182
..... 182
56: ..... 183
..... 183
Examples ..... 183
..... 183
..... 184
..... 185
57: ..... 187
..... 187
: ..... 187
Examples ..... 187
..... 187
..... 188
..... 188
..... 188
PartialFunctions ..... 189
58: ..... 190
..... 190
Examples ..... 190
..... 190
( ) ..... 191
..... 191

59: ..... 193
Examples ..... 193
..... 193
..... 193
..... 193
..... 194
..... 194
- ..... 195
60: ..... 197
Examples ..... 197
..... 197
`collect` ..... 197
..... 198
..... 199
..... 199
61: ..... 201
..... 201
Examples ..... 201
..... 201
..... 202
..... 202
..... 204
62: ..... 206
..... 206
Examples ..... 206
..... 206
..... 207
Unapply - ..... 207
Extractor Infix ..... 209
..... 209
..... 209
..... 211

# Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: scala-language

It is an unofficial and free Scala Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Scala Language.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# глава 1: Начало работы с языком Scala

## замечания

Scala - это современный язык программирования с несколькими парадигмами, предназначенный для выражения общих шаблонов программирования в краткой, элегантной и безопасной форме. Он плавно интегрирует функции объектно-ориентированного и функционального языков.

Для большинства приведенных примеров требуется рабочая установка Scala. Это страница установки Scala , и это пример «Как настроить Scala» . scalafiddle.net - хороший ресурс для выполнения небольших примеров кода через Интернет.

## Версии

| Версия | Дата выхода |
|---|---|
| 2.10.1 | 2013-03-13 |
| 2.10.2 | 2013-06-06 |
| 2.10.3 | 2013-10-01 |
| 2.10.4 | 2014-03-24 |
| 2.10.5 | 2015-03-05 |
| 2.10.6 | 2015-09-18 |
| 2.11.0 | 2014-04-21 |
| 2.11.1 | 2014-05-21 |
| 2.11.2 | 2014-07-24 |
| 2.11.4 | 2014-10-30 |
| 2.11.5 | 2014-01-14 |
| 2.11.6 | 2015-03-05 |
| 2.11.7 | 2015-06-23 |
| 2.11.8 | 2016-03-08 |
| 2.11.11 | 2017-04-19 |

| Версия | Дата выхода |
| --- | --- |
| 2.12.0 | 2016-11-03 |
| 2.12.1 | 2016-12-06 |
| 2.12.2 | 2017-04-19 |

# Examples

## Hello World, определяя «основной» метод

Поместите этот код в файл `HelloWorld.scala` :

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

Демо-версия

Чтобы скомпилировать его в байт-код, исполняемый JVM:

```
$ scalac HelloWorld.scala
```

Чтобы запустить его:

```
$ scala Hello
```

Когда среда выполнения Scala загружает программу, она ищет объект с именем `Hello` с `main` методом. `main` метод - это точка входа в программу и выполняется.

Обратите внимание, что в отличие от Java, у Scala нет требования об именовании объектов или классов после файла, в котором они находятся. Вместо этого параметр `Hello` переданный в команде `scala Hello` ссылается на объект для поиска, содержащий `main` метод, который должен быть выполнен. В одном файле `.scala` вполне возможно иметь несколько объектов с основными методами.

Массив `args` будет содержать аргументы командной строки, заданные программе, если таковые имеются. Например, мы можем изменить программу следующим образом:

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello World!")
    for {
      arg <- args
    } println(s"Arg=$arg")
```

```
  }
}
```

Скомпилируйте его:

```
$ scalac HelloWorld.scala
```

И затем выполните его:

```
$ scala HelloWorld 1 2 3
Hello World!
Arg=1
Arg=2
Arg=3
```

### Hello World, расширяя приложение

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

Демо-версия

Расширяя черту `App` , вы можете избежать определения явного `main` метода. Весь объект `HelloWorld` рассматривается как «основной метод».

2.11.0

## Отложенная инициализация

В официальной документации `App` использует функцию « *Задержка инициализации»* . Это означает, что поля объекта инициализируются *после* вызова основного метода.

2.11.0

## Отложенная инициализация

В официальной документации `App` использует функцию « *Задержка инициализации»* . Это означает, что поля объекта инициализируются *после* вызова основного метода.

`DelayedInit` теперь **устарел** для общего использования, но по- *прежнему поддерживается* `App` как особый случай. Поддержка будет продолжаться до тех пор, пока не будет решена и не будет реализована функция замены.

Чтобы получить доступ к аргументам командной строки при расширении `App` , используйте `this.args` :

```
object HelloWorld extends App {
  println("Hello World!")
  for {
    arg <- this.args
  } println(s"Arg=$arg")
}
```

При использовании `App` тело объекта будет выполняться как `main` метод, нет необходимости переопределять `main` .

## Hello World как сценарий

Scala может использоваться как язык сценариев. Чтобы продемонстрировать, создайте `HelloWorld.scala` со следующим содержимым:

```
println("Hello")
```

Выполните его с помощью интерпретатора командной строки ( `$` - приглашение командной строки):

```
$ scala HelloWorld.scala
Hello
```

Если вы опустите `.scala` (например, если вы просто набрали `scala HelloWorld` ), бегун будет искать скомпилированный файл `.class` с байт-кодом вместо компиляции и последующего выполнения скрипта.

> **Примечание.** Если scala используется как язык сценариев, пакет не может быть определен.

В операционных системах, использующих `bash` или аналогичные терминалы оболочки, скрипты Scala могут быть выполнены с использованием «преамбулы оболочки». Создайте файл с именем `HelloWorld.sh` и поместите в него содержимое:

```
#!/bin/sh
exec scala "$0" "$@"
!#
println("Hello")
```

Части между `#!` и `!#` является «преамбулой оболочки» и интерпретируется как сценарий bash. Остальное - Скала.

После того как вы сохранили вышеуказанный файл, вы должны предоставить ему «исполняемые» разрешения. В оболочке вы можете сделать это:

```
$ chmod a+x HelloWorld.sh
```

(Обратите внимание, что это дает разрешение всем: прочитайте о chmod, чтобы узнать, как установить его для более конкретных наборов пользователей.)

Теперь вы можете выполнить скрипт следующим образом:

```
$ ./HelloWorld.sh
```

## Использование Scala REPL

Когда вы выполняете `scala` в терминале без дополнительных параметров, он открывает интерпретатор REPL (Read-Eval-Print Loop):

```
nford:~ $ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66).
Type in expressions for evaluation. Or try :help.

scala>
```

REPL позволяет выполнять Scala в режиме листа: контекст выполнения сохраняется, и вы можете вручную опробовать команды, не создавая целую программу. Например, набрав `val poem = "As halcyons we shall be"` будет выглядеть так:

```
scala> val poem = "As halcyons we shall be"
poem: String = As halcyons we shall be
```

Теперь мы можем распечатать наш `val` :

```
scala> print(poem)
As halcyons we shall be
```

Обратите внимание, что `val` неизменен и не может быть перезаписан:

```
scala> poem = "Brooding on the open sea"
<console>:12: error: reassignment to val
       poem = "Brooding on the open sea"
```

Но в REPL вы *можете* переопределить `val` (что вызовет ошибку в обычной программе Scala, если это было сделано в той же области):

```
scala> val poem = "Brooding on the open sea"
poem: String = Brooding on the open sea
```

На оставшуюся часть вашего сеанса REPL эта вновь определенная переменная будет затенять ранее определенную переменную. REPLs полезны для быстрого просмотра того, как работают объекты или другой код. Доступны все возможности Scala: вы можете

определять функции, классы, методы и т. Д.

## Scala Quicksheet

| Описание | Код |
|---|---|
| Назначить неизменяемую стоимость int | `val x = 3` |
| Назначить изменяемую стоимость int | `var x = 3` |
| Назначить неизменяемое значение с явным типом | `val x: Int = 27` |
| Присвоить лениво оцениваемое значение | `lazy val y = print("Sleeping in.")` |
| Привязать функцию к имени | `val f = (x: Int) => x * x` |
| Привязать функцию к имени с явным типом | `val f: Int => Int = (x: Int) => x * x` |
| Определить метод | `def f(x: Int) = x * x` |
| Определить метод с явным набором текста | `def f(x: Int): Int = x * x` |
| Определить класс | `class Hopper(someParam: Int) { ... }` |
| Определить объект | `object Hopper(someParam: Int) { ... }` |
| Определить черту | `trait Grace { ... }` |
| Получить первый элемент последовательности | `Seq(1,2,3).head` |
| Если переключатель | `val result = if(x > 0) "Positive!"` |
| Получить все элементы последовательности, кроме первых | `Seq(1,2,3).tail` |
| Прокрутите список | `for { x <- Seq(1,2,3) } print(x)` |
| Вложенные петли | `for {`<br>`x <- Seq(1,2,3)`<br>`y <- Seq(4,5,6)`<br>`} print(x + ":" + y)` |
| Для каждого элемента списка выполните функцию | `List(1,2,3).foreach { println }` |
| Печатать до стандартного | `print("Ada Lovelace")` |
| Сортировка списка по алфавиту | `List('b','c','a').sorted` |

Прочитайте Начало работы с языком Scala онлайн: https://riptutorial.com/ru/scala/topic/216/начало-работы-с-языком-scala

# глава 2: Implicits

## Синтаксис

- неявный val x: T = ???

## замечания

Неявные классы позволяют добавлять пользовательские методы к существующим типам без необходимости изменять их код, тем самым обогащая типы без необходимости управления кодом.

Использование неявных типов для обогащения существующего класса часто называют шаблоном «обогатить мою библиотеку».

**Ограничения на неявные классы**

1. Неявные классы могут существовать только в пределах другого класса, объекта или свойства.
2. Неявные классы могут иметь только один неявный первичный конструктор.
3. В пределах той же области действия, которая имеет то же имя, что и неявный класс, не может быть другого объекта, класса, признака или определения члена класса.

## Examples

### Неявное преобразование

Неявное преобразование позволяет компилятору автоматически преобразовывать объект одного типа в другой тип. Это позволяет коду обрабатывать объект как объект другого типа.

```
case class Foo(i: Int)

// without the implicit
Foo(40) + 2    // compilation-error (type mismatch)

// defines how to turn a Foo into an Int
implicit def fooToInt(foo: Foo): Int = foo.i

// now the Foo is converted to Int automatically when needed
Foo(40) + 2    // 42
```

Преобразование одностороннее: в этом случае вы не можете преобразовать `42` в `Foo(42)` . Для этого необходимо определить второе неявное преобразование:

```
implicit def intToFoo(i: Int): Foo = Foo(i)
```

Обратите внимание, что это механизм, посредством которого, например, можно добавить значение float к целочисленному значению.

> Неявные преобразования должны использоваться экономно, потому что они запутывают то, что происходит. Лучше всего использовать явное преобразование посредством вызова метода, если нет очевидной выгоды от использования неявного преобразования.

Значительное влияние неявных преобразований не оказывает.

Scala автоматически импортирует различные неявные преобразования в `scala.Predef` , включая все преобразования с Java на Scala и обратно. Они включены по умолчанию в любую компиляцию файлов.

## Неявные параметры

Неявные параметры могут быть полезны, если параметр типа должен быть определен один раз в области и затем применяться ко всем функциям, использующим значение этого типа.

Обычный вызов функции выглядит примерно так:

```
// import the duration methods
import scala.concurrent.duration._

// a normal method:
def doLongRunningTask(timeout: FiniteDuration): Long = timeout.toMillis

val timeout = 1.second
// timeout: scala.concurrent.duration.FiniteDuration = 1 second

// to call it
doLongRunningTask(timeout) // 1000
```

Теперь давайте скажем, что у нас есть некоторые методы, у которых есть время ожидания, и мы хотим вызвать все эти методы, используя один и тот же тайм-аут. Мы можем определить таймаут как неявную переменную.

```
// import the duration methods
import scala.concurrent.duration._

// dummy methods that use the implicit parameter
def doLongRunningTaskA()(implicit timeout: FiniteDuration): Long = timeout.toMillis
def doLongRunningTaskB()(implicit timeout: FiniteDuration): Long = timeout.toMillis

// we define the value timeout as implicit
implicit val timeout: FiniteDuration = 1.second

// we can now call the functions without passing the timeout parameter
```

```
doLongRunningTaskA() // 1000
doLongRunningTaskB() // 1000
```

Способ, которым это работает, заключается в том, что компилятор scalac ищет значение в области, которое **помечено как неявное** и **тип которого соответствует** одному из неявных параметров. Если он найдет один, он применит его как неявный параметр.

> Обратите внимание, что это не сработает, если вы определяете два или даже больше импликации одного и того же типа в области.

Чтобы настроить сообщение об ошибке, используйте аннотацию `implicitNotFound` для типа:

```
@annotation.implicitNotFound(msg = "Select the proper implicit value for type M[${A}]!")
case class M[A](v: A) {}

def usage[O](implicit x: M[O]): O = x.v

//Does not work because no implicit value is present for type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
implicit val first: M[Int] = M(1)
usage[Int]     //Works when `second` is not in scope
implicit val second: M[Int] = M(2)
//Does not work because more than one implicit values are present for the type `M[Int]`
//usage[Int]   //Select the proper implicit value for type M[Int]!
```

Тайм-аут является обычным прецедентом для этого, или, например, в Акке, ActorSystem (в большинстве случаев) всегда одинакова, поэтому она обычно передается неявно. Другим вариантом использования будет дизайн библиотеки, чаще всего с библиотеками FP, которые полагаются на классы (например, скалаз , кошки или восторг ).

> Обычно считается неправильной практикой использовать неявные параметры с такими базовыми типами, как *Int* , *Long* , *String* и т. Д., Так как это создаст путаницу и сделает код менее читаемым.

## Неявные классы

Неявные классы позволяют добавлять новые методы к ранее определенным классам.

Класс `String` не имеет метода `withoutVowels` . Это можно добавить так:

```
object StringUtil {
  implicit class StringEnhancer(str: String) {
    def withoutVowels: String = str.replaceAll("[aeiou]", "")
  }
}
```

Неявный класс имеет единственный конструктор ( `str` ) с типом, который вы хотели бы расширить ( `String` ), и содержит метод, который вы хотите «добавить» к типу (без `withoutVowels` ). Новые методы теперь могут использоваться непосредственно для расширенного типа (когда расширенный тип находится в неявной области):

```
import StringUtil.StringEnhancer // Brings StringEnhancer into implicit scope

println("Hello world".withoutVowels) // Hll wrld
```

Под капотом неявные классы определяют неявное преобразование из расширенного типа в неявный класс, например:

```
implicit def toStringEnhancer(str: String): StringEnhancer = new StringEnhancer(str)
```

Неявные классы часто определяются как классы значений, чтобы избежать создания объектов времени выполнения и, таким образом, удалить служебные данные во время выполнения:

```
implicit class StringEnhancer(val str: String) extends AnyVal {
    /* conversions code here */
}
```

С приведенным выше улучшенным определением новый экземпляр `StringEnhancer` не нужно создавать каждый раз, когда `withoutVowels` метод `withoutVowels` .

## Разрешение неявных параметров с использованием «неявно»

Предположим, что список неявных параметров содержит более одного неявного параметра:

```
case class Example(p1:String, p2:String)(implicit ctx1:SomeCtx1, ctx2:SomeCtx2)
```

Теперь, предполагая, что один из неявных экземпляров недоступен ( `SomeCtx1` ), в то время как все другие неявные экземпляры необходимы в области видимости, для создания экземпляра класса должен быть предоставлен экземпляр `SomeCtx1` .

Это можно сделать, сохранив друг друга в неявном экземпляре с использованием ключевого слова `implicitly` :

```
Example("something","somethingElse")(new SomeCtx1(), implicitly[SomeCtx2])
```

## Имплициты в REPL

Чтобы просмотреть все `implicits` в области видимости во время сеанса REPL:

```
scala> :implicits
```

Также включать неявные преобразования, определенные в `Predef.scala` :

```
scala> :implicits -v
```

Если у вас есть выражение и вы хотите увидеть влияние всех правил перезаписи, которые применяются к нему (включая implicits):

```
scala> reflect.runtime.universe.reify(expr) // No quotes. reify is a macro operating directly
on code.
```

(Пример:

```
scala> import reflect.runtime.universe._
scala> reify(Array("Alice", "Bob", "Eve").mkString(", "))
resX: Expr[String] = Expr[String](Predef.refArrayOps(Array.apply("Alice", "Bob",
"Eve")(Predef.implicitly)).mkString(", "))
```

)

Прочитайте Implicits онлайн: https://riptutorial.com/ru/scala/topic/1732/implicits

# глава 3: JSON

## Examples

### JSON с распылителем

spray-json обеспечивает простой способ работы с JSON. Используя неявные форматы, все происходит «за кулисами»:

# Сделать библиотеку доступной с SBT

Чтобы управлять `spray-json` с зависимостями управляемой библиотеки SBT :

```
libraryDependencies += "io.spray" %% "spray-json" % "1.3.2"
```

Обратите внимание, что последний параметр, номер версии ( `1.3.2` ), может отличаться в разных проектах.

Библиотека `spray-json` размещена на сайте repo.spray.io .

## Импорт библиотеки

```
import spray.json._
import DefaultJsonProtocol._
```

По умолчанию протокол JSON `DefaultJsonProtocol` содержит форматы для всех основных типов. Чтобы обеспечить функциональность JSON для пользовательских типов, либо используйте конструкторы удобства для форматов, либо форматы записи явно.

# Читать JSON

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = """{ "foo": "bar" }""".parseJson // JsValue = {"foo":"bar"}

res.convertTo[Map[String, String]] // Map(foo -> bar)
```

# Написать JSON

```
val values = List("a", "b", "c")
values.toJson.prettyPrint // ["a", "b", "c"]
```

# DSL

DSL не поддерживается.

# Чтение-запись в класс Case

В следующем примере показано, как сериализовать объект класса case в формате JSON.

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = jsonFormat2(Address)
implicit val personFormat = jsonFormat2(Person)

// serialize a Person
Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = fred.toJson.prettyPrint
```

Это приводит к следующему JSON:

```
{
  "name": "Fred",
  "address": {
    "street": "Awesome Street 9",
    "city": "SuperCity"
  }
}
```

То, что JSON может, в свою очередь, десериализоваться обратно в объект:

```
val personRead = fredJsonString.parseJson.convertTo[Person]
//Person(Fred,Address(Awesome Street 9,SuperCity))
```

# Пользовательский формат

Напишите пользовательский `JsonFormat` если требуется специальная сериализация типа. Например, если имена полей в Scala различны, чем в JSON. Или, если различные конкретные типы создаются на основе ввода.

```
implicit object BetterPersonFormat extends JsonFormat[Person] {
    // deserialization code
    override def read(json: JsValue): Person = {
        val fields = json.asJsObject("Person object expected").fields
        Person(
            name = fields("name").convertTo[String],
            address = fields("home").convertTo[Address]
```

```
        )
    }

    // serialization code
    override def write(person: Person): JsValue = JsObject(
        "name" -> person.name.toJson,
        "home" -> person.address.toJson
    )
}
```

## JSON с Circe

Circe предоставляет кодеки, производные от компиляции, для en / decode json в классах классов. Простой пример выглядит следующим образом:

```
import io.circe._
import io.circe.generic.auto._
import io.circe.parser._
import io.circe.syntax._

case class User(id: Long, name: String)

val user = User(1, "John Doe")

// {"id":1,"name":"John Doe"}
val json = user.asJson.noSpaces

// Right(User(1L, "John Doe"))
val res: Either[Error, User] = decode[User](json)
```

## JSON с play-json

play-json использует неявные форматы в качестве других json-фреймворков

SBT-зависимость: `libraryDependencies += ""com.typesafe.play" %% "play-json" % "2.4.8"`

```
import play.api.libs.json._
import play.api.libs.functional.syntax._ // if you need DSL
```

`DefaultFormat` содержит форматы defaul для чтения / записи всех основных типов. Чтобы обеспечить функциональность JSON для ваших собственных типов, вы можете явно использовать конструкторы удобства для форматов или форматов записи.

### Читать json

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = Json.parse("""{ "foo": "bar" }""") // JsValue = {"foo":"bar"}

res.as[Map[String, String]]                  // Map(foo -> bar)
res.validate[Map[String, String]]            //JsSuccess(Map(foo -> bar),)
```

### Написать json

```
val values = List("a", "b", "c")
Json.stringify(Json.toJson(values))          // ["a", "b", "c"]
```

### DSL

```
val json = parse("""{ "foo": [{"foo": "bar"}]}""")
(json \ "foo").get                    //Simple path: [{"foo":"bar"}]
(json \\ "foo")                       //Recursive path:List([{"foo":"bar"}], "bar")
(json \ "foo")(0).get                  //Index lookup (for JsArrays): {"foo":"bar"}
```

*Как всегда предпочитайте сопоставление шаблонов с `JsSuccess` / `JsError` и старайтесь избегать `.get` , `array(i)` .*

### Чтение и запись в класс case

```
case class Address(street: String, city: String)
case class Person(name: String, address: Address)

// create the formats and provide them implicitly
implicit val addressFormat = Json.format[Address]
implicit val personFormat = Json.format[Person]

// serialize a Person
val fred = Person("Fred", Address("Awesome Street 9", "SuperCity"))
val fredJsonString = Json.stringify(Json.toJson(Json.toJson(fred)))

val personRead = Json.parse(fredJsonString).as[Person] //Person(Fred,Address(Awesome Street
9,SuperCity))
```

### Собственный формат

Вы можете написать свой собственный JsonFormat, если вам нужна специальная сериализация вашего типа (например, называть поля по-разному в scala и Json или создавать экземпляры разных конкретных типов на основе ввода)

```
case class Address(street: String, city: String)

// create the formats and provide them implicitly
implicit object AddressFormatCustom extends Format[Address] {
  def reads(json: JsValue): JsResult[Address] = for {
    street <- (json \ "Street").validate[String]
    city <- (json \ "City").validate[String]
  } yield Address(street, city)

  def writes(x: Address): JsValue = Json.obj(
    "Street" -> x.street,
    "City" -> x.city
  )
}
// serialize an address
val address = Address("Awesome Street 9", "SuperCity")
val addressJsonString = Json.stringify(Json.toJson(Json.toJson(address)))
//{"Street":"Awesome Street 9","City":"SuperCity"}

val addressRead = Json.parse(addressJsonString).as[Address]
```

```
//Address(Awesome Street 9,SuperCity)
```

### альтернатива

Если json точно не соответствует вашим полям класса case ( `isAlive` в случае класса vs `is_alive` в json):

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Boolean)

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").read[Boolean]
    ) (User.apply _)
}
```

### Json с дополнительными полями

```
case class User(username: String, friends: Int, enemies: Int, isAlive: Option[Boolean])

object User {

  import play.api.libs.functional.syntax._
  import play.api.libs.json._

  implicit val userReads: Reads[User] = (
      (JsPath \ "username").read[String] and
      (JsPath \ "friends").read[Int] and
      (JsPath \ "enemies").read[Int] and
      (JsPath \ "is_alive").readNullable[Boolean]
    ) (User.apply _)
}
```

### Чтение временных меток от json

Представьте, что у вас есть объект Json с полем отметки Unix:

```
{
  "field": "example field",
  "date": 1459014762000
}
```

решение:

```
case class JsonExampleV1(field: String, date: DateTime)
object JsonExampleV1{
  implicit val r: Reads[JsonExampleV1] = (
    (__ \ "field").read[String] and
```

```
      (__ \ "date").read[DateTime](Reads.DefaultJodaDateReads)
    )(JsonExampleV1.apply _)
}
```

### Чтение пользовательских классов

Теперь, если вы завернете идентификаторы объектов для безопасности типов, вам это понравится. См. Следующий объект json:

```
{
  "id": 91,
  "data": "Some data"
}
```

и соответствующие классы случаев:

```
case class MyIdentifier(id: Long)

case class JsonExampleV2(id: MyIdentifier, data: String)
```

Теперь вам просто нужно прочитать примитивный тип (Long) и сопоставить с вашим idenfier:

```
object JsonExampleV2 {
  implicit val r: Reads[JsonExampleV2] = (
      (__ \ "id").read[Long].map(MyIdentifier) and
    (__ \ "data").read[String]
    )(JsonExampleV2.apply _)
}
```

код на странице https://github.com/pedrorijo91/scala-play-json-examples

## JSON с json4s

json4s использует неявные форматы в качестве других json-фреймворков.

Зависимость SBT:

```
libraryDependencies += "org.json4s" %% "json4s-native" % "3.4.0"
//or
libraryDependencies += "org.json4s" %% "json4s-jackson" % "3.4.0"
```

### импорт

```
import org.json4s.JsonDSL._
import org.json4s._
import org.json4s.native.JsonMethods._

implicit val formats = DefaultFormats
```

`DefaultFormats`

содержит форматы по умолчанию для чтения / записи всех основных типов.

### Читать json

```
// generates an intermediate JSON representation (abstract syntax tree)
val res = parse("""{ "foo": "bar" }""")           // JValue = {"foo":"bar"}
res.extract[Map[String, String]]                  // Map(foo -> bar)
```

### Написать json

```
val values = List("a", "b", "c")
compact(render(values))         // ["a", "b", "c"]
```

### DSL

```
json \ "foo"       //Simple path: JArray(List(JObject(List((foo,JString(bar))))))
json \\ "foo"      //Recursive path: ~List([{"foo":"bar"}], "bar")
(json \ "foo")(0)  //Index lookup (for JsArrays): JObject(List((foo,JString(bar))))
("foo" -> "bar") ~ ("field" -> "value")    // {"foo":"bar","field":"value"}
```

### Чтение и запись в класс case

```
import org.json4s.native.Serialization.{read, write}

case class Address(street: String, city: String)
val addressString = write(Address("Awesome stree", "Super city"))
// {"street":"Awesome stree","city":"Super city"}

read[Address](addressString) // Address(Awesome stree,Super city)
//or
parse(addressString).extract[Address]
```

### Чтение и запись гетерогенных списков

Для сериализации и десериализации гетерогенного (или полиморфного) списка необходимо указать конкретные подсказки типа.

```
trait Location
case class Street(name: String) extends Location
case class City(name: String, zipcode: String) extends Location
case class Address(street: Street, city: City) extends Location
case class Locations (locations : List[Location])

implicit val formats = Serialization.formats(ShortTypeHints(List(classOf[Street],
classOf[City], classOf[Address])))

val locationsString = write(Locations(Street("Lavelle Street"):: City("Super city","74658")))

read[Locations](locationsString)
```

### Собственный формат

```
class AddressSerializer extends CustomSerializer[Address](format => (
  {
    case JObject(JField("Street", JString(s)) :: JField("City", JString(c)) :: Nil) =>
      new Address(s, c)
  },
  {
    case x: Address => ("Street" -> x.street) ~ ("City" -> x.city)
  }
  ))

implicit val formats = DefaultFormats + new AddressSerializer
val str = write[Address](Address("Awesome Stree", "Super City"))
// {"Street":"Awesome Stree","City":"Super City"}
read[Address](str)
// Address(Awesome Stree,Super City)
```

Прочитайте JSON онлайн: https://riptutorial.com/ru/scala/topic/2348/json

# глава 4: Quasiquotes

## Examples

### Создание дерева синтаксиса с квазикварталами

Используйте квазивоты для создания `Tree` в макросе.

```
object macro {
  def addCreationDate(): java.util.Date = macro impl.addCreationDate
}

object impl {
  def addCreationDate(c: Context)(): c.Expr[java.util.Date] = {
    import c.universe._

    val date = q"new java.util.Date()" // this is the quasiquote
    c.Expr[java.util.Date](date)
  }
}
```

Он может быть произвольно сложным, но он будет проверен для правильного синтаксиса scala.

Прочитайте Quasiquotes онлайн: https://riptutorial.com/ru/scala/topic/4032/quasiquotes

# глава 5: Scala.js

## Вступление

`Scala.js` - это порт от `Scala` который компилируется на `JavaScript` , который в конце будет работать за пределами `JVM` . Он имеет преимущества, такие как сильная типизация, оптимизация кода во время компиляции, полная совместимость с библиотеками JavaScript.

## Examples

### console.log в Scala.js

```
println("Hello Scala.js") // In ES6: console.log("Hello Scala.js");
```

### Функции жирной стрелки

```
val lastNames = people.map(p => p.lastName)
// Or shorter:
val lastNames = people.map(_.lastName)
```

### Простой класс

```
class Person(val firstName: String, val lastName: String) {
  def fullName(): String =
    s"$firstName $lastName"
}
```

### Коллекции

```
val personMap = Map(
  10 -> new Person("Roger", "Moore"),
  20 -> new Person("James", "Bond")
)
val names = for {
  (key, person) <- personMap
  if key > 15
} yield s"$key = ${person.firstName}"
```

### Манипулирование DOM

```
import org.scalajs.dom
import dom.document

def appendP(target: dom.Node, text: String) = {
  val pNode = document.createElement("p")
```

```
  val textNode = document.createTextNode(text)
  pNode.appendChild(textNode)
  target.appendChild(pNode)
}
```

### Использование с SBT

## Sbt зависимость

```
libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "0.9.1" // (Triple %%%)
```

## Бег

```
sbt run
```

## Работа с непрерывной компиляцией:

```
sbt ~run
```

## Скомпилируйте один файл JavaScript:

```
sbt fastOptJS
```

Прочитайте Scala.js онлайн: https://riptutorial.com/ru/scala/topic/9426/scala-js

# глава 6: Scaladoc

## Синтаксис

- Выбирает методы, поля, классы или пакеты.
- Начинается с `/**`
- В каждой строке есть начальная `*` процедура с комментариями
- Заканчивается на `*/`

## параметры

| параметр | подробности |
|---|---|
| **Специально для класса** | _ |
| `@constructor detail` | Объясняет главный конструктор класса |
| **Метод специфический** | _ |
| `@return detail` | Подробности о том, что возвращается методу. |
| **Теги метода, конструктора и / или класса** | _ |
| `@param x detail` | Сведения о параметре значения `x` для метода или конструктора. |
| `@tparam x detail` | Подробная информация о параметре типа `x` для метода или конструктора. |
| `@throws detail` | Какие исключения могут быть брошены. |
| **использование** | _ |
| `@see detail` | Ссылки на другие источники информации. |
| `@note detail` | Добавляет примечание для условий до или после сообщения или любых других заметных ограничений или ожиданий. |
| `@example detail` | Предоставляет примерный код или соответствующую документацию по примерам. |

| параметр | подробности |
| --- | --- |
| `@usecase detail` | Предоставляет упрощенное определение метода, когда полное определение метода слишком сложное или шумное. |
| **Другой** | _ |
| `@author detail` | Предоставляет информацию об авторе о следующем. |
| `@version detail` | Предоставляет версию, которой принадлежит эта часть. |
| `@deprecated detail` | Помечает следующий объект как устаревший. |

# Examples

## Простой метод Scaladoc к методу

```
/**
  * Explain briefly what method does here
  * @param x Explain briefly what should be x and how this affects the method.
  * @param y Explain briefly what should be y and how this affects the method.
  * @return Explain what is returned from execution.
  */
def method(x: Int, y: String): Option[Double] = {
  // Method content
}
```

Прочитайте Scaladoc онлайн: https://riptutorial.com/ru/scala/topic/4518/scaladoc

# глава 7: scalaz

## Вступление

Scalaz - библиотека Scala для функционального программирования.

Он обеспечивает чисто функциональные структуры данных, дополняющие те из стандартной библиотеки Scala. Он определяет набор классов базового типа (например, `Functor` , `Monad` ) и соответствующие экземпляры для большого количества структур данных.

## Examples

### ApplyUsage

```
import scalaz._
import Scalaz._

scala> Apply[Option].apply2(some(1), some(2))((a, b) => a + b)
res0: Option[Int] = Some(3)

scala> val intToString: Int => String = _.toString

scala> Apply[Option].ap(1.some)(some(intToString))
res1: Option[String] = Some(1)

scala> Apply[Option].ap(none)(some(intToString))
res2: Option[String] = None

scala> val double: Int => Int = _ * 2

scala> Apply[List].ap(List(1, 2, 3))(List(double))
res3: List[Int] = List(2, 4, 6)

scala> :kind Apply
scalaz.Apply's kind is X[F[A]]
```

### FunctorUsage

```
import scalaz._
import Scalaz._
scala> val len: String => Int = _.length
len: String => Int = $$Lambda$1164/969820333@7e758f40

scala> Functor[Option].map(Some("foo"))(len)
res0: Option[Int] = Some(3)

scala> Functor[Option].map(None)(len)
res1: Option[Int] = None

scala> Functor[List].map(List("qwer", "adsfg"))(len)
res2: List[Int] = List(4, 5)
```

```
scala> :kind Functor
scalaz.Functor's kind is X[F[A]]
```

## ArrowUsage

```
import scalaz._
import Scalaz._
scala> val plus1 = (_: Int) + 1
plus1: Int => Int = $$Lambda$1167/111311964 9@6a6bfd97

scala> val plus2 = (_: Int) + 2
plus2: Int => Int = $$Lambda$1168/924329548@6bbe050f

scala> val rev = (_: String).reverse
rev: String => String = $$Lambda$1227/1278001332@72685b74

scala> plus1.first apply (1, "abc")
res0: (Int, String) = (2,abc)

scala> plus1.second apply ("abc", 2)
res1: (String, Int) = (abc,3)

scala> rev.second apply (1, "abc")
res2: (Int, String) = (1,cba)

scala> plus1 *** rev apply(7, "abc")
res3: (Int, String) = (8,cba)

scala> plus1 &&& plus2 apply 7
res4: (Int, Int) = (8,9)

scala> plus1.product apply (1, 2)
res5: (Int, Int) = (2,3)

scala> :kind Arrow
scalaz.Arrow's kind is X[F[A1,A2]]
```

Прочитайте scalaz онлайн: https://riptutorial.com/ru/scala/topic/9893/scalaz

# глава 8: Streams

## замечания

Потоки лениво оцениваются, то есть они могут использоваться для реализации генераторов, которые будут предоставлять или «генерировать» новый элемент указанного типа по запросу, а не до этого. Это гарантирует выполнение только необходимых вычислений.

## Examples

### Использование потока для генерации случайной последовательности

`genRandom` создает поток случайных чисел, который имеет один из четырех шансов прекратить каждый раз, когда он вызывается.

```
def genRandom: Stream[String] = {
  val random = scala.util.Random.nextFloat()
  println(s"Random value is: $random")
  if (random < 0.25) {
    Stream.empty[String]
  } else {
    ("%.3f : A random number" format random) #:: genRandom
  }
}

lazy val randos = genRandom  // getRandom is lazily evaluated as randos is iterated through

for {
  x <- randos
} println(x) // The number of times this prints is effectively randomized.
```

Обратите внимание на конструкцию `#::` , которая *лениво рекурсирует* : поскольку она *добавляет* текущее случайное число к потоку, оно не оценивает оставшуюся часть потока до тех пор, пока оно не будет выполнено.

### Бесконечные потоки через рекурсию

Потоки могут быть построены так, чтобы ссылаться на них и, таким образом, становиться бесконечно рекурсивными.

```
// factorial
val fact: Stream[BigInt] = 1 #:: fact.zipWithIndex.map{case (p,x)=>p*(x+1)}
fact.take(10)  // (1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880)
fact(24)       // 620448401733239439360000

// the Fibonacci series
val fib: Stream[BigInt] = 0 #:: fib.scan(1:BigInt)(_+_)
```

```
fib.take(10)  // (0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
fib(124)      // 367267407055057792558994 43

// random Ints between 10 and 99 (inclusive)
def rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(10)  // (20, 95, 14, 44, 42, 78, 85, 24, 99, 85)
```

В этом контексте интересна разница между Var, Val и Def . Как `def` каждый элемент пересчитывается каждый раз, когда он ссылается. В качестве значения `val` каждый элемент сохраняется и повторно используется после его вычисления. Это можно продемонстрировать, создавая побочный эффект при каждом вычислении.

```
// def with extra output per calculation
def fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!!!!!!!!!!! 120
fact(4)  // !!!!!!!!!! 24
fact(7)  // !!!!!!!!!!!!!!!!!!!!!!!!!!!! 5040

// now as val
val fact: Stream[Int] = 1 #:: fact.zipWithIndex.map{case (p,x)=>print("!");p*(x+1)}
fact(5)  // !!!!! 120
fact(4)  // 24
fact(7)  // !! 5040
```

Это также объясняет, почему случайное число `Stream` не работает как `val` .

```
val rndInt: Stream[Int] = (util.Random.nextInt(90)+10) #:: rndInt
rndInt.take(5)  // (79, 79, 79, 79, 79)
```

## Бесконечный поток самореферментa

```
// Generate stream that references itself in its evaluation
lazy val primes: Stream[Int] =
  2 #:: Stream.from(3, 2)
    .filter { i => primes.takeWhile(p => p * p <= i).forall(i % _ != 0) }
    .takeWhile(_ > 0) // prevent overflowing

// Get list of 10 primes
assert(primes.take(10).toList == List(2, 3, 5, 7, 11, 13, 17, 19, 23, 29))

// Previously calculated values were memoized, as shown by toString
assert(primes.toString == "Stream(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ?)")
```

Прочитайте Streams онлайн: https://riptutorial.com/ru/scala/topic/3702/streams

# глава 9: Var, Val и Def

## замечания

Поскольку `val` семантически статичны, они инициализируются «на месте», где бы они ни появлялись в коде. Это может вызвать неожиданное и нежелательное поведение при использовании в абстрактных классах и чертах.

Например, предположим, что мы хотели бы сделать признак, называемый `PlusOne` который определяет операцию приращения на обернутом `Int` . Поскольку `Int` s неизменяемы, значение плюс один известно при инициализации и никогда не будет изменено впоследствии, поэтому семантически это значение `val` . Однако определение этого способа приведет к неожиданному результату.

```
trait PlusOne {
    val i:Int

    val incr = i + 1
}

class IntWrapper(val i: Int) extends PlusOne
```

Независимо от того, какое значение `i IntWrapper` , вызов `.incr` в возвращаемом объекте всегда будет возвращаться 1. Это происходит потому, что val `incr` инициализируется *в признаке* , перед расширяющим классом, и в это время у `i` есть только значение по умолчанию `0` . (В других условиях он может быть заполнен `Nil` , `null` или аналогичным значением по умолчанию).

Таким образом, общее правило состоит в том, чтобы избежать использования `val` для любого значения, которое зависит от абстрактного поля. Вместо этого используйте `lazy val` , который не оценивается, пока он не понадобится, или `def` , который оценивает каждый раз, когда он вызывается. Обратите внимание, однако, что если `lazy val` вынужден оценивать по `val` до завершения инициализации, произойдет такая же ошибка.

Здесь можно найти скрипку (написанную в Scala-Js, но такое же поведение) .

## Examples

### Var, Val и Def

# вар

Параметр `var` является ссылочной переменной, подобной переменным в таких языках, как

Java. Различные объекты могут быть свободно назначены для `var` , если данный объект имеет тот же тип, что и `var` :

```
scala> var x = 1
x: Int = 1

scala> x = 2
x: Int = 2

scala> x = "foo bar"
<console>:12: error: type mismatch;
 found   : String("foo bar")
 required: Int
       x = "foo bar"
       ^
```

Обратите внимание, что в примере выше тип `var` был выведен компилятором при первом присвоении значения.

# вал

`val` является постоянной ссылкой. Таким образом, новый объект не может быть назначен `val` , которое уже было назначено.

```
scala> val y = 1
y: Int = 1

scala> y = 2
<console>:12: error: reassignment to val
       y = 2
         ^
```

Однако объект, на который указывает `val` , *не* является постоянным. Этот объект может быть изменен:

```
scala> val arr = new Array[Int](2)
arr: Array[Int] = Array(0, 0)

scala> arr(0) = 1

scala> arr
res1: Array[Int] = Array(1, 0)
```

# Защита

`def` определяет метод. Метод не может быть повторно назначен.

```
scala> def z = 1
z: Int
```

```
scala> z = 2
<console>:12: error: value z_= is not a member of object $iw
       z = 2
       ^
```

В приведенных выше примерах `val y` и `def z` возвращают одно и то же значение. Тем не менее, `def` оценивается, *когда он вызывается* , тогда как `val` или `var` оценивается, *когда он назначен* . Это может привести к различному поведению, когда определение имеет побочные эффекты:

```
scala> val a = {println("Hi"); 1}
Hi
a: Int = 1

scala> def b = {println("Hi"); 1}
b: Int

scala> a + 1
res2: Int = 2

scala> b + 1
Hi
res3: Int = 2
```

# функции

Поскольку функции являются значениями, они могут быть назначены `val` / `var` / `def` s. Все остальное работает так же, как описано выше:

```
scala> val x = (x: Int) => s"value=$x"
x: Int => String = <function1>

scala> var y = (x: Int) => s"value=$x"
y: Int => String = <function1>

scala> def z = (x: Int) => s"value=$x"
z: Int => String

scala> x(1)
res0: String = value=1

scala> y(2)
res1: String = value=2

scala> z(3)
res2: String = value=3
```

## Lazy val

`lazy val` - это языковая функция, при которой инициализация `val` задерживается до тех пор, пока она не будет доступна в первый раз. После этого момента он действует как

обычный `val` .

Чтобы использовать его, добавьте `lazy` ключевое слово до `val` . Например, используя REPL:

```
scala> lazy val foo = {
     |   println("Initializing")
     |   "my foo value"
     | }
foo: String = <lazy>

scala> val bar = {
     |   println("Initializing bar")
     |   "my bar value"
     | }
Initializing bar
bar: String = my bar value

scala> foo
Initializing
res3: String = my foo value

scala> bar
res4: String = my bar value

scala> foo
res5: String = my foo value
```

Этот пример демонстрирует порядок выполнения. Когда объявляется `lazy val` , все, что сохраняется в значение `foo` - это ленивый вызов функции, который еще не был оценен. Когда установлен правильный `val` , мы видим, что вызов `println` выполняется, и значение присваивается `bar` . Когда мы evalute `foo` в первый раз видим, что `println` выполняется, но не тогда, когда он оценивается во второй раз. Аналогично, когда `bar` оценивается, мы не видим `println` execute - только при его объявлении.

# Когда использовать «ленивый»

1. **Инициализация является дорогостоящим вычислительным методом, а использование `val` является редкостью.**

   ```
   lazy val tiresomeValue = {(1 to 1000000).filter(x => x % 113 == 0).sum}
   if (scala.util.Random.nextInt > 1000) {
     println(tiresomeValue)
   }
   ```

   `tiresomeValue` занимает много времени, и она не всегда используется. Делать это `lazy val` спасает ненужные вычисления.

2. **Разрешение циклических зависимостей**

   Давайте рассмотрим пример с двумя объектами, которые должны быть объявлены

одновременно во время создания экземпляра:

```
object comicBook {
  def main(args:Array[String]): Unit = {
    gotham.hero.talk()
    gotham.villain.talk()
  }
}

class Superhero(val name: String) {
  lazy val toLockUp = gotham.villain
  def talk(): Unit = {
    println(s"I won't let you win ${toLockUp.name}!")
  }
}

class Supervillain(val name: String) {
  lazy val toKill = gotham.hero
  def talk(): Unit = {
    println(s"Let me loosen up Gotham a little bit ${toKill.name}!")
  }
}

object gotham {
  val hero: Superhero = new Superhero("Batman")
  val villain: Supervillain = new Supervillain("Joker")
}
```

Без ключевого слова `lazy` соответствующие объекты не могут быть членами объекта. Выполнение такой программы приведет к `java.lang.NullPointerException` . Используя `lazy` , ссылка может быть назначена до ее инициализации, не опасаясь иметь неинициализированное значение.

## Перегрузка Def

Вы можете перегрузить `def` если подпись отличается:

```
def printValue(x: Int) {
  println(s"My value is an integer equal to $x")
}

def printValue(x: String) {
  println(s"My value is a string equal to '$x'")
}

printValue(1)  // prints "My value is an integer equal to 1"
printValue("1") // prints "My value is a string equal to '1'"
```

Это работает так же, как внутри классов, черт, объектов или нет.

## Именованные параметры

При вызове `def` параметры могут быть назначены явно по имени. Это означает, что они не должны быть правильно заказаны. Например, определите `printUs()` как:

```
// print out the three arguments in order.
def printUs(one: String, two: String, three: String) =
   println(s"$one, $two, $three")
```

Теперь это можно назвать таким образом (среди прочих):

```
printUs("one", "two", "three")
printUs(one="one", two="two", three="three")
printUs("one", two="two", three="three")
printUs(three="three", one="one", two="two")
```

Это приводит к `one, two, three` что во всех случаях печатается `one, two, three` .

Если не все аргументы названы, первые аргументы сопоставляются по порядку. Никакой позиционный (неименованный) аргумент не может соответствовать названному:

```
printUs("one", two="two", three="three") // prints 'one, two, three'
printUs(two="two", three="three", "one") // fails to compile: 'positional after named
argument'
```

Прочитайте Var, Val и Def онлайн: https://riptutorial.com/ru/scala/topic/3155/var--val-и-def

# глава 10: Аннотации

## Синтаксис

- @AnAnnotation def someMethod = {...}
- Класс @AnAnnotation someClass {...}
- @AnnotatioWithArgs (annotation_args) def someMethod = {...}

## параметры

| параметр | подробности |
|---|---|
| @ | Указывает, что следующий токен представляет собой аннотацию. |
| SomeAnnotation | Название аннотации |
| constructor_args | (необязательно) Аргументы, переданные аннотации. Если нет, скобки не требуются. |

## замечания

Scala-lang предоставляет список стандартных аннотаций и их эквивалентов Java .

## Examples

### Использование аннотации

Эта аннотация показывает, что следующий метод `deprecated` .

```
@deprecated
def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

Это также можно записать эквивалентно:

```
@deprecated def anUnusedLegacyMethod(someArg: Any) =  {
  ...
}
```

### Аннотирование основного конструктора

```
/**
 * @param num Numerator
 * @param denom Denominator
 * @throws ArithmeticException in case `denom` is `0`
 */
class Division @throws[ArithmeticException](/*no annotation parameters*/) protected (num: Int,
denom: Int) {
    private[this] val wrongValue = num / denom

    /** Integer number
     *  @param num Value */
    protected[Division] def this(num: Int) {
      this(num, 1)
    }
}
object Division {
  def apply(num: Int) = new Division(num)
  def apply(num: Int, denom: Int) = new Division(num, denom)
}
```

Модификатор видимости (в этом случае `protected` ) должен появиться после аннотаций в той же строке. Если аннотация принимает необязательные параметры (так как в этом случае `@throws` принимает необязательную причину), вы должны указать пустой список параметров для аннотации: `()` перед параметрами конструктора.

Примечание. Можно указать несколько аннотаций, даже с одного и того же типа ( повторяющиеся аннотации ).

Аналогично классу case без вспомогательного заводского метода (и причина, указанная для аннотации):

```
case class Division @throws[ArithmeticException]("denom is 0") (num: Int, denom: Int) {
    private[this] val wrongValue = num / denom
}
```

## Создание собственных аннотаций

Вы можете создать свои собственные аннотации Scala, создав классы, полученные из scala.annotation.StaticAnnotation или scala.annotation.ClassfileAnnotation

```
package animals
// Create Annotation `Mammal`
class Mammal(indigenous:String) extends scala.annotation.StaticAnnotation

// Annotate class Platypus as a `Mammal`
@Mammal(indigenous = "North America")
class Platypus{}
```

Затем аннотации могут быть опрошены с использованием API отражения.

```
scala>import scala.reflect.runtime.{universe ⇒ u}
```

```
scala>val platypusType = u.typeOf[Platypus]
platypusType: reflect.runtime.universe.Type = animals.reflection.Platypus

scala>val platypusSymbol = platypusType.typeSymbol.asClass
platypusSymbol: reflect.runtime.universe.ClassSymbol = class Platypus

scala>platypusSymbol.annotations
List[reflect.runtime.universe.Annotation] = List(animals.reflection.Mammal("North America"))
```

Прочитайте Аннотации онлайн: https://riptutorial.com/ru/scala/topic/3783/аннотации

# глава 11: Библиотека непрерывности

## Вступление

Стиль продолжения передачи - это форма потока управления, которая включает в себя передачу функций остальной части вычисления в качестве аргумента «продолжение». Следующая функция вызывает это продолжение, чтобы продолжить выполнение программы. Один из способов думать о продолжении - это закрытие. Библиотека продолжения Scala приносит разграниченные продолжения в виде `shift` / `reset` примитивов к языку.

библиотека продолжений: https://github.com/scala/scala-continuations

## Синтаксис

- reset {...} // Продолжение продолжается до конца блока сброса
- shift {...} // Создаем продолжение, указывающее после вызова, передавая его закрытию
- A @cpsParam [B, C] // Вычисление, требующее функции A => B для создания значения C
- @cps [A] // Псевдоним для @cpsParam [A, A]
- @suspendable // Псевдоним для @cpsParam [Unit, Unit]

## замечания

`shift` и `reset` - это примитивные структуры потока управления, такие как `Int.+` - примитивная операция, а `Long` - примитивный тип. Они более примитивны, чем либо в том, что разграниченные продолжения могут фактически использоваться для построения почти всех структур управления потоком. Они не очень полезны «из коробки», но они действительно блестят, когда они используются в библиотеках для создания богатых API.

Продолжения и монады также тесно связаны. В продолжение монады можно продолжить , и монады являются продолжениями, потому что их операция `flatMap` берет продолжение как параметр.

## Examples

### Обратные вызовы - это продолжение

```
// Takes a callback and executes it with the read value
def readFile(path: String)(callback: Try[String] => Unit): Unit = ???
```

```
readFile(path) { _.flatMap { file1 =>
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}
```

Аргумент функции `readFile` является продолжением, поскольку `readFile` вызывает его, чтобы продолжить выполнение программы после того, как она выполнила свою работу.

Чтобы обуздать то, что может легко стать адским обратным вызовом, мы используем библиотеку продолжений.

```
reset { // Reset is a delimiter for continuations.
  for { // Since the callback hell is relegated to continuation library machinery.
        // a for-comprehension can be used
    file1 <- shift(readFile(path1)) // shift has type (((A => B) => C) => A)
    // We use it as (((Try[String] => Unit) => Unit) => Try[String])
    // It takes all the code that occurs after it is called, up to the end of reset, and
    // makes it into a closure of type (A => B).
    // The reason this works is that shift is actually faking its return type.
    // It only pretends to return A.
    // It actually passes that closure into its function parameter (readFile(path1) here),
    // And that function calls what it thinks is a normal callback with an A.
    // And through compiler magic shift "injects" that A into its own callsite.
    // So if readFile calls its callback with parameter Success("OK"),
    // the shift is replaced with that value and the code is executed until the end of reset,
    // and the return value of that is what the callback in readFile returns.
    // If readFile called its callback twice, then the shift would run this code twice too.
    // Since readFile returns Unit though, the type of the entire reset expression is Unit
    //
    // Think of shift as shifting all the code after it into a closure,
    // and reset as resetting all those shifts and ending the closures.
    file2 <- shift(readFile(path2))
  } processFiles(file1, file2)
}

// After compilation, shift and reset are transformed back into closures
// The for comprehension first desugars to:
reset {
  shift(readFile(path1)).flatMap { file1 => shift(readFile(path2)).foreach { file2 =>
processFiles(file1, file2) } }
}
// And then the callbacks are restored via CPS transformation
readFile(path1) { _.flatMap { file1 => // We see how shift moves the code after it into a
closure
  readFile(path2) { _.foreach { file2 =>
    processFiles(file1, file2)
  }}
}}  // And we see how reset closes all those closures
// And it looks just like the old version!
```

## Создание функций, обеспечивающих непрерывность

Если `shift` вызывается за пределами разделительного блока `reset` , его можно использовать для создания функций, которые сами создают продолжения внутри блока `reset` . Важно отметить, что тип `shift` не просто `(((A => B) => C) => A)` , он фактически `(((A`

`=> B) => C) => (A @cpsParam[B, C]))` . Эти аннотации означают, что необходимы преобразования CPS. Функции, которые вызывают `shift` без `reset` имеют тип возвращаемого типа «заражен» этой аннотацией.

Внутри блока `reset` значение `A @cpsParam[B, C]` похоже имеет значение `A` , хотя на самом деле это просто притворство. Продолжение, необходимое для завершения вычисления, имеет тип `A => B` , поэтому код, следующий за методом, возвращающим этот тип, должен возвращать `B C` - это «реальный» тип возврата, а после преобразования CPS вызов функции имеет тип `C`

Теперь, пример, взятый из Scaladoc библиотеки

```
val sessions = new HashMap[UUID, Int=>Unit]
def ask(prompt: String): Int @suspendable = // alias for @cpsParam[Unit, Unit]. @cps[Unit] is
also an alias. (@cps[A] = @cpsParam[A,A])
  shift {
    k: (Int => Unit) => {
      println(prompt)
      val id = uuidGen
      sessions += id -> k
    }
  }

def go(): Unit = reset {
  println("Welcome!")
  val first = ask("Please give me a number") // Uses CPS just like shift
  val second = ask("Please enter another number")
  printf("The sum of your numbers is: %d\n", first + second)
}
```

Здесь `ask` будет хранить продолжение в карте, а затем некоторый другой код может получить этот «сеанс» и передать результат запроса пользователю. Таким образом, `go` может фактически использовать асинхронную библиотеку, в то время как ее код выглядит как нормальный императивный код.

Прочитайте Библиотека непрерывности онлайн: https://riptutorial.com/ru/scala/topic/8312/библиотека-непрерывности

# глава 12: В то время как петли

## Синтаксис

- while (boolean_expression) {block_expression}
- do {block_expression} while (boolean_expression)

## параметры

| параметр | подробности |
|---|---|
| `boolean_expression` | Любое выражение, которое будет оцениваться как `true` или `false` . |
| `block_expression` | Любое выражение или набор выражений, которые будут оцениваться, если `boolean_expression` `true` . |

## замечания

Основное различие между циклами `while` и `do-while` заключается в том, выполняются ли они `block_expression` прежде чем они `block_expression` проверку, должны ли они зацикливаться.

Поскольку циклы `while` и `do-while` полагаются на выражение для вычисления `false` для завершения, они часто требуют, чтобы изменяемое состояние было объявлено вне цикла, а затем модифицировано внутри цикла.

## Examples

### В то время как петли

```
var line = 0
var maximum_lines = 5

while (line < maximum_lines) {
  line = line + 1
  println("Line number: " + line)
}
```

### Циклы «Do-While»

```
var line = 0
var maximum_lines = 5
```

```
do {
  line = line + 1
  println("Line number: " + line)
} while (line < maximum_lines)
```

`do` / в `while` цикл редко используется в функциональном программировании, но могут быть использованы для работы вокруг отсутствие поддержки `break` / `continue` конструкцию, как показано на других языках:

```
if(initial_condition) do if(filter) {
  ...
} while(continuation_condition)
```

Прочитайте В то время как петли онлайн: https://riptutorial.com/ru/scala/topic/650/в-то-время-как-петли

# глава 13: Внедрение зависимости

## Examples

### Cake Pattern с внутренним классом реализации.

```
//create a component that will be injected
trait TimeUtil {
    lazy val timeUtil = new TimeUtilImpl()

    class TimeUtilImpl{
        def now() = new DateTime()
    }
}

//main controller is depended on time util
trait MainController {
    _ : TimeUtil => //inject time util into main controller

    lazy val mainController = new MainControllerImpl()

    class MainControllerImpl {
        def printCurrentTime() = println(timeUtil.now()) //timeUtil is injected from TimeUtil
trait
    }
}

object MainApp extends App {
    object app extends MainController
       with TimeUtil //wire all components

    app.mainController.printCurrentTime()
}
```

В приведенном выше примере я продемонстрировал, как вводить TimeUtil в MainController.

Наиболее важным синтаксисом является самообновление ( `_: TimeUtil =>` ), которое должно вводить `TimeUtil` в `MainController` . `MainController` словами, `MainController` зависит от `TimeUtil` .

Я использую внутренний класс (например, `TimeUtilImpl` ) в каждом компоненте, потому что, на мой взгляд, его легче тестировать, поскольку мы можем издеваться над внутренним классом. И также проще отслеживать, откуда вызывается метод, когда проект становится более сложным.

Наконец, я соединяю все компоненты вместе. Если вы знакомы с Guice, это эквивалентно `Binding`

Прочитайте Внедрение зависимости онлайн: https://riptutorial.com/ru/scala/topic/5909/внедрение-зависимости

# глава 14: Вывод типа

## Examples

### Локальный вывод типа

Scala имеет мощный механизм ввода-вывода, встроенный в язык. Этот механизм называется «Local Type Inference»:

```
val i = 1 + 2                 // the type of i is Int
val s = "I am a String"       // the type of s is String
def squared(x : Int) = x*x    // the return type of squared is Int
```

Компилятор может вывести тип переменных из выражения инициализации. Аналогично, возвращаемый тип методов можно опустить, поскольку они эквивалентны типу, возвращаемому телом метода. Вышеприведенные примеры эквивалентны приведенным ниже объявлениям явного типа:

```
val i: Int = 1 + 2
val s: String = "I am a String"
def squared(x : Int): Int = x*x
```

### Тип вывода и дженерики

Компилятор Scala также может выводить параметры типа при вызове полиморфных методов или когда генерируются генерические классы:

```
case class InferedPair[A, B](a: A, b: B)

val pairFirstInst = InferedPair("Husband", "Wife")  //type is InferedPair[String, String]

// Equivalent, with type explicitly defined
val pairSecondInst: InferedPair[String, String]
                      = InferedPair[String, String]("Husband", "Wife")
```

Вышеупомянутая форма вывода типа похожа на оператора Diamond , представленную на Java 7.

### Ограничения на вывод

Существуют сценарии, в которых тип-вывод Scala не работает. Например, компилятор не может вывести тип параметров метода:

```
def add(a, b) = a + b  // Does not compile
def add(a: Int, b: Int) = a + b // Compiles
def add(a: Int, b: Int): Int = a + b // Equivalent expression, compiles
```

Компилятор не может вывести возвращаемый тип рекурсивных методов:

```
// Does not compile
def factorial(n: Int) = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
// Compiles
def factorial(n: Int): Int = if (n == 0 || n == 1) 1 else n * factorial(n - 1)
```

## Предотвращение вывода

Основано на этом сообщении в блоге .

Предположим, у вас есть такой метод:

```
  def get[T]: Option[T] = ???
```

Когда вы пытаетесь вызвать его без указания общего параметра, `Nothing` не выводится, что не очень полезно для реальной реализации (и его результат не пригодится). При следующем растворе `NotNothing` контекст связан можно предотвратить с помощью метода без указания ожидаемого типа (в данном примере `RuntimeClass` также исключаются за `ClassTags` не `Nothing` , но `RuntimeClass` выводятся):

```
@implicitNotFound("Nothing was inferred")
sealed trait NotNothing[-T]

object NotNothing {
  implicit object notNothing extends NotNothing[Any]
  //We do not want Nothing to be inferred, so make an ambigous implicit
  implicit object `\n The error is because the type parameter was resolved to Nothing` extends
NotNothing[Nothing]
  //For classtags, RuntimeClass can also be inferred, so making that ambigous too
  implicit object `\n The error is because the type parameter was resolved to RuntimeClass`
extends NotNothing[RuntimeClass]
}

object ObjectStore {
  //Using context bounds
  def get[T: NotNothing]: Option[T] = {
    ???
  }

  def newArray[T](length: Int = 10)(implicit ct: ClassTag[T], evNotNothing: NotNothing[T]):
Option[Array[T]] = ???
}
```

Пример использования:

```
object X {
  //Fails to compile
  //val nothingInferred = ObjectStore.get

  val anOption = ObjectStore.get[String]
  val optionalArray = ObjectStore.newArray[AnyRef]()
```

```
  //Fails to compile
  //val runtimeClassInferred = ObjectStore.newArray()
}
```

Прочитайте Вывод типа онлайн: https://riptutorial.com/ru/scala/topic/4918/вывод-типа

# глава 15: Динамический вызов

## Вступление

Scala позволяет использовать динамический вызов при вызове методов или доступе к полям на объекте. Вместо того, чтобы встроить этот язык глубоко в язык, это достигается путем переписывания правил, аналогичных правилам неявных преобразований, которые активируются маркерным признаком [ `scala.Dynamic` ] [Dynamic scaladoc]. Это позволяет эмулировать способность динамически добавлять свойства к объектам, присутствующим в динамических языках, и многое другое. [Dynamic scaladoc]: http://www.scala-lang.org/api/2.12.x/scala/Dynamic.html

## Синтаксис

- класс Foo расширяет динамический
- foo.field
- foo.field = значение
- foo.method (арг)
- foo.method (namedArg = x, y)

## замечания

Чтобы объявить подтипы `Dynamic` , `dynamics` характеристик языка необходимо активировать, импортируя `scala.language.dynamics` или `-language:dynamics` compiler. Пользователи этого `Dynamic` которые не определяют свои собственные подтипы, не должны включать это.

## Examples

### Доступ к полям

Это:

```
class Foo extends Dynamic {
  // Expressions are only rewritten to use Dynamic if they are not already valid
  // Therefore foo.realField will not use select/updateDynamic
  var realField: Int = 5
  // Called for expressions of the type foo.field
  def selectDynamic(fieldName: String) = ???
  def updateDynamic(fieldName: String)(value: Int) = ???
}
```

обеспечивает простой доступ к полям:

```
val foo: Foo = ???
foo.realField // Does NOT use Dynamic; accesses the actual field
foo.realField = 10 // Actual field access here too
foo.unrealField // Becomes foo.selectDynamic(unrealField)
foo.field = 10  // Becomes foo.updateDynamic("field")(10)
foo.field = "10" // Does not compile; "10" is not an Int.
foo.x() // Does not compile; Foo does not define applyDynamic, which is used for methods.
foo.x.apply() // DOES compile, as Nothing is a subtype of () => Any
// Remember, the compiler is still doing static type checks, it just has one more way to
// "recover" and rewrite otherwise invalid code now.
```

## Вызов метода

Это:

```
class Villain(val minions: Map[String, Minion]) extends Dynamic {
  def applyDynamic(name: String)(jobs: Task*) = jobs.foreach(minions(name).do)
  def applyDynamicNamed(name: String)(jobs: (String, Task)*) = jobs.foreach {
    // If a parameter does not have a name, and is simply given, the name passed as ""
    case ("", task) => minions(name).do(task)
    case (subsys, task) => minions(name).subsystems(subsys).do(task)
  }
}
```

позволяет использовать вызовы методов с и без параметров:

```
val gru: Villain = ???
gru.blu() // Becomes gru.applyDynamic("blu")()
// Becomes gru.applyDynamicNamed("stu")(("fooer", ???), ("boomer", ???), ("", ???),
//          ("computer breaker", ???), ("fooer", ???))
// Note how the `???` without a name is given the name ""
// Note how both occurrences of `fooer` are passed to the method
gru.stu(fooer = ???, boomer = ???, ???, `computer breaker` = ???, fooer = ???)
gru.ERR("a") // Somehow, scalac thinks "a" is not a Task, though it clearly is (it isn't)
```

## Взаимодействие между полевым доступом и методом обновления

Немного контрастивно (но и единственный разумный способ заставить его работать):

```
val dyn: Dynamic = ???
dyn.x(y) = z
```

эквивалентно:

```
dyn.selectDynamic("x").update(y, z)
```

в то время как

```
dyn.x(y)
```

все еще

```
dyn.applyDynamic("x")(y)
```

Важно знать об этом, иначе он может незаметно прокрасться и вызвать странные ошибки.

Прочитайте Динамический вызов онлайн: https://riptutorial.com/ru/scala/topic/8296/динамический-вызов

# глава 16: Для выражений

## Синтаксис

- для тела {clauses}
- для {clauses} результата тела
- для (статей) тела
- для (статей) дают тело

## параметры

| параметр | подробности |
|---|---|
| за | Требуемое ключевое слово для использования цикла / понимания for |
| статьи | Итерация и фильтры, над которыми работает. |
| Уступать | Используйте это, если вы хотите создать или «уронить» коллекцию. Используя `yield` вызовет тип возврата `for` быть коллекция вместо `Unit` . |
| тело | Тело выражения for, выполняемое на каждой итерации. |

## Examples

### Базовый для цикла

```
for (x <- 1 to 10)
  println("Iteration number " + x)
```

Это демонстрирует итерацию переменной `x` , от `1` до `10` и выполнение чего-то с этим значением. Тип возврата `for` понимания - `Unit` .

### Основы для понимания

Это демонстрирует фильтр для цикла и использование `yield` для создания «понимания последовательности»:

```
for ( x <- 1 to 10 if x % 2 == 0)
  yield x
```

Выход для этого:

```
scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)
```

Для понимания полезно, когда вам нужно создать новую коллекцию на основе итерации и ее фильтров.

### Вложенный для цикла

Это показывает, как вы можете перебирать несколько переменных:

```
for {
  x <- 1 to 2
  y <- 'a' to 'd'
} println("(" + x + "," + y + ")")
```

(Обратите внимание , что `to` здесь является методом инфиксного оператора , который возвращает диапазон включительно . См определения здесь .)

Это создает выход:

```
(1,a)
(1,b)
(1,c)
(1,d)
(2,a)
(2,b)
(2,c)
(2,d)
```

Обратите внимание, что это эквивалентное выражение, используя скобки вместо скобок:

```
for (
  x <- 1 to 2
  y <- 'a' to 'd'
) println("(" + x + "," + y + ")")
```

Чтобы получить все комбинации в один вектор, мы можем `yield` результат и установить его в `val` :

```
val a = for {
  x <- 1 to 2
  y <- 'a' to 'd'
} yield "(%s,%s)".format(x, y)
// a: scala.collection.immutable.IndexedSeq[String] = Vector((1,a), (1,b), (1,c), (1,d),
(2,a), (2,b), (2,c), (2,d))
```

### Монадический для понимания

Если у вас есть несколько объектов монадических типов, мы можем добиться сочетания значений, используя «для понимания»:

```
for {
   x <- Option(1)
   y <- Option("b")
   z <- List(3, 4)
} {
    // Now we can use the x, y, z variables
    println(x, y, z)
    x  // the last expression is *not* the output of the block in this case!
}

// This prints
// (1, "b", 3)
// (1, "b", 4)
```

Тип возврата этого блока - `Unit` .

Если объекты имеют один и тот *же* монадический тип `M` (например, `Option` ), то использование `yield` вернет объект типа `M` вместо `Unit` .

```
val a = for {
   x <- Option(1)
   y <- Option("b")
} yield {
    // Now we can use the x, y variables
    println(x, y)
    // whatever is at the end of the block is the output
    (7 * x, y)
}

// This prints:
// (1, "b")
// The val `a` is set:
// a: Option[(Int, String)] = Some((7,b))
```

Обратите внимание, что ключевое слово `yield` *нельзя* использовать в исходном примере, где есть сочетание монадических типов ( `Option` и `List` ). Попытка сделать это приведет к ошибке несоответствия типа компиляции.

## Итерация через коллекции с использованием цикла for

Это демонстрирует, как печатать каждый элемент карты

```
val map = Map(1 -> "a", 2 -> "b")
for (number <- map) println(number) // prints (1,a), (2,b)
for ((key, value) <- map) println(value) // prints a, b
```

Это демонстрирует, как печатать каждый элемент списка

```
val list = List(1,2,3)
for(number <- list) println(number) // prints 1, 2, 3
```

## Desugaring для понимания

`for` понимания в Scala просто синтаксический сахар . Эти понимания реализуются с использованием `withFilter` , `foreach` , `flatMap` и `map` их типов объектов. По этой причине только типы, которые имеют эти методы, могут быть использованы `for` понимания.

А `for` понимания следующей формы с образцами `pN` , генераторами `gN` и условиями `cN` :

```
for(p0 <- x0 if g0; p1 <- g1 if c1) { ??? }
```

... будет `withFilter` для вложенных вызовов, используя `withFilter` и `foreach` :

```
g0.withFilter({ case p0 => c0  case _ => false }).foreach({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).foreach({
    case p1 => ???
  })
})
```

В то время `for` выражение `for` / `yield` имеет следующий вид:

```
for(p0 <- g0 if c0; p1 <- g1 if c1) yield ???
```

... будет `withFilter` для вложенных вызовов, используя `withFilter` и либо `flatMap` либо `map` :

```
g0.withFilter({ case p0 => c0  case _ => false }).flatMap({
  case p0 => g1.withFilter({ case p1 => c1  case _ => false }).map({
    case p1 => ???
  })
})
```

(Обратите внимание, что `map` используется во внутреннем понимании, а `flatMap` используется во всех внешних `flatMap` .)

А `for` понимания может применяться к любому типу, реализующему методы, требуемые де-сагированным представлением. Никаких ограничений на типы возврата этих методов не существует, если они являются составными.

Прочитайте Для выражений онлайн: https://riptutorial.com/ru/scala/topic/669/для-выражений

# глава 17: Единые абстрактные типы методов (типы SAM)

## замечания

Единые абстрактные методы - это типы, введенные в Java 8 , которые имеют ровно один абстрактный член.

## Examples

### Синтаксис Lambda

**ПРИМЕЧАНИЕ. Это доступно только в Scala 2.12+ (и в последних версиях 2.11.x с `-Xexperimental -Xfuture` компилятора)**

Тип SAM может быть реализован с использованием лямбда:

2.11.8

```
trait Runnable {
  def run(): Unit
}

val t: Runnable = () => println("foo")
```

Тип может необязательно иметь другие не-абстрактные элементы:

2.11.8

```
trait Runnable {
  def run(): Unit
  def concrete: Int = 42
}

val t: Runnable = () => println("foo")
```

Прочитайте Единые абстрактные типы методов (типы SAM) онлайн: https://riptutorial.com/ru/scala/topic/3664/единые-абстрактные-типы-методов--типы-sam-

# глава 18: Если выражения

## Examples

### Основные выражения if

В Scala (в отличие от Java и большинства других языков) **выражение** `if` вместо *выражения* . Независимо от того, синтаксис идентичен:

```
if(x < 1984) {
   println("Good times")
} else if(x == 1984) {
   println("The Orwellian Future begins")
} else {
   println("Poor guy...")
}
```

Импликация `if` являющегося выражением, заключается в том, что вы можете присвоить результат вычисления выражения переменной:

```
val result = if(x > 0) "Greater than 0" else "Less than or equals 0"
\\ result: String = Greater than 0
```

Выше мы видим, что выражение `if` оценивается, и `result` устанавливается на это результирующее значение.

Тип возвращаемого выражения `if` является **супертипом** всех ветвей логики. Это означает, что для этого примера возвращаемый тип является `String` . Поскольку не все , `if` выражения возвращают значение (например, `if` заявление , которое не имеет `else` филиальную логики), то возможно , что тип возвращаемого значения `Any` :

```
val result = if(x > 0) "Greater than 0"
// result: Any = Greater than 0
```

Если никакое значение не может быть возвращено (например, если в логических ветвях используются только побочные эффекты, такие как `println` ), тогда тип возврата будет `Unit` :

```
val result = if(x > 0) println("Greater than 0")
// result: Unit = ()
```

`if` выражения в Scala похожи на то, как работает тернарный оператор в Java . Из-за этого сходства в Scala такого оператора нет, он был бы лишним.

> Фигурные скобки могут быть опущены в `if` выражение , если содержание представляет собой одну строку.

Прочитайте Если выражения онлайн: https://riptutorial.com/ru/scala/topic/4171/если-выражения

# глава 19: Интерполяция строк

## замечания

Эта функция существует в Scala 2.10.0 и выше.

## Examples

### Интерпретация Hello String

Интерполятор **s** позволяет использовать переменные в строке.

```
val name = "Brian"
println(s"Hello $name")
```

печатает «Hello Brian» на консоль при запуске.

### Форматированная интерполяция строк с помощью интерполятора f

```
val num = 42d
```

Распечатайте два десятичных знака для `num` используя *f*

```
println(f"$num%2.2f")
42.00
```

Печать `num` использованием научной нотации с помощью *электронной*

```
println(f"$num%e")
4.200000e+01
```

Печать `num` в шестнадцатеричном с

```
println(f"$num%a")
0x1.5p5
```

Другие строки формата можно найти на странице https://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#detail

### Использование выражения в строковых литералах

Вы можете использовать фигурные скобки для интерполяции выражений в строковые литералы:

```
def f(x: String) = x + x
val a = "A"

s"${a}"    // "A"
s"${f(a)}" // "AA"
```

---

Без фигурных скобок scala будет только интерполировать *идентификатор* после `$` (в этом случае `f` ). Поскольку нет никакого неявного преобразования из `f` в `String` это исключение в этом примере:

```
s"$f(a)"  // compile-time error (missing argument list for method f)
```

## Пользовательские строковые интерполяторы

В дополнение к встроенным можно определить пользовательские строковые интерполяторы.

```
my"foo${bar}baz"
```

Распространяется компилятором на:

```
new scala.StringContext("foo", "baz").my(bar)
```

`scala.StringContext` не имеет `my` метода, поэтому он может быть обеспечен неявным преобразованием. Обычай Интерполятор с таким же поведением , как и встроенный `s` интерпол затем будет осуществляться следующим образом :

```
implicit class MyInterpolator(sc: StringContext) {
  def my(subs: Any*): String = {
    val pit = sc.parts.iterator
    val sit = subs.iterator
    // Note parts.length == subs.length + 1
    val sb = new java.lang.StringBuilder(pit.next())
    while(sit.hasNext) {
      sb.append(sit.next().toString)
      sb.append(pit.next())
    }
    sb.toString
  }
}
```

И интерполяция `my"foo${bar}baz"` будет обессоливать:

```
new MyInterpolation(new StringContext("foo", "baz")).my(bar)
```

Обратите внимание, что нет никаких ограничений на аргументы или возвращаемый тип функции интерполяции. Это приводит нас к темному пути, где синтаксис интерполяции может быть использован творчески для создания произвольных объектов, как показано в

следующем примере:

```
case class Let(name: Char, value: Int)

implicit class LetInterpolator(sc: StringContext) {
  def let(value: Int): Let = Let(sc.parts(0).charAt(0), value)
}

let"a=${4}" // Let(a, 4)
let"b=${"foo"}" // error: type mismatch
let"c=" // error: not enough arguments for method let: (value: Int)Let
```

## Строковые интерполяторы как экстракторы

Также возможно использовать функцию интерполяции строк Scala для создания сложных экстракторов (шаблонов шаблонов), которые, возможно, наиболее широко используются в API квазикодов макросов Scala.

Учитывая, что `n"p0${i0}p1"` desugars для `new StringContext("p0", "p1").n(i0)` , возможно, неудивительно, что функция экстрактора включена, обеспечивая неявное преобразование из `StringContext` в класс с свойство `n` типа, определяющего метод `unapply` или `unapplySeq` .

В качестве примера рассмотрим следующий экстрактор, который извлекает сегменты пути, создавая регулярное выражение из частей `StringContext` . Затем мы можем делегировать большую часть тяжелого подъема методу `unapplySeq` предоставленному результатом scala.util.matching.Regex :

```
implicit class PathExtractor(sc: StringContext) {
  object path {
    def unapplySeq(str: String): Option[Seq[String]] =
      sc.parts.map(Regex.quote).mkString("^", "([^/]+)", "$").r.unapplySeq(str)
  }
}

"/documentation/scala/1629/string-interpolation" match {
  case path"/documentation/${topic}/${id}/${_}" => println(s"$topic, $id")
  case _ => ???
}
```

Обратите внимание, что объект `path` также может определить метод `apply` , чтобы вести себя как обычный интерполятор.

## Интерполяция сырой строки

Вы можете использовать **необработанный** интерполятор, если вы хотите, чтобы строка была напечатана как есть и без какого-либо выхода из литералов.

```
println(raw"Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

С использованием **исходного** интерполятора вы должны увидеть следующее, напечатанное на консоли:

```
Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le Monde
```

Без **необработанного** интерполятора `\n` и `\t` были бы экранированы.

```
println("Hello World In English And French\nEnglish:\tHello World\nFrench:\t\tBonjour Le
Monde")
```

Печать:

```
Hello World In English And French
English:        Hello World
French:         Bonjour Le Monde
```

Прочитайте Интерполяция строк онлайн: https://riptutorial.com/ru/scala/topic/1629/интерполяция-строк

# глава 20: Карринг

## Синтаксис

- aFunction (10) _ // Использование '_' Сообщает компилятору, что все параметры в остальных группах параметров будут зависеть.
- nArityFunction.curried // Преобразует функцию n-arity в эквивалентную версию в карри
- anotherFunction (x) (_: String) (z) // Выполнение произвольного параметра. Он явно требует своего типа.

## Examples

### Конфигурируемый множитель в виде каррической функции

```
def multiply(factor: Int)(numberToBeMultiplied: Int): Int = factor * numberToBeMultiplied

val multiplyBy3 = multiply(3)_     // resulting function signature Int => Int
val multiplyBy10 = multiply(10)_ // resulting function signature Int => Int

val sixFromCurriedCall = multiplyBy3(2) //6
val sixFromFullCall = multiply(3)(2)    //6

val fortyFromCurriedCall = multiplyBy10(4) //40
val fortyFromFullCall = multiply(10)(4)    //40
```

### Несколько групп параметров разных типов, параметры каррирования произвольных позиций

```
def numberOrCharacterSwitch(toggleNumber: Boolean)(number: Int)(character: Char): String =
  if (toggleNumber) number.toString else character.toString

// need to explicitly specify the type of the parameter to be curried
// resulting function signature Boolean => String
val switchBetween3AndE = numberOrCharacterSwitch(_: Boolean)(3)('E')

switchBetween3AndE(true) // "3"
switchBetween3AndE(false) // "E"
```

### Вычисление функции с помощью одной группы параметров

```
def minus(left: Int, right: Int) = left - right

val numberMinus5 = minus(_: Int, 5)
val fiveMinusNumber = minus(5, _: Int)

numberMinus5(7)    //  2
fiveMinusNumber(7) // -2
```

## Карринг

Определим функцию из двух аргументов:

```
def add: (Int, Int) => Int = (x,y) => x + y
val three = add(1,2)
```

Currying `add` преобразует его в функцию, которая принимает **один** `Int` и возвращает **функцию** (от **одного** `Int` до `Int` )

```
val addCurried: (Int) => (Int => Int) = add2.curried
//               ^~~ take *one* Int
//                         ^~~~ return a *function* from Int to Int

val add1: Int => Int = addCurried(1)
val three: Int = add1(2)
val allInOneGo: Int = addCurried(1)(2)
```

Вы можете применить это понятие к любой функции, которая принимает несколько аргументов. Вычисляя функцию, которая принимает несколько аргументов, преобразует ее в ряд приложений функций, которые принимают **один** аргумент:

```
def add3: (Int, Int, Int) => Int = (a,b,c) => a + b + c + d
def add3Curr: Int => (Int => (Int => Int)) = add3.curried

val x = add3Curr(1)(2)(42)
```

## Карринг

Карри, согласно Википедии ,

> является методом перевода оценки функции, которая принимает несколько аргументов при оценке последовательности функций.

Конкретно, в терминах типов scala, в контексте функции, которая принимает два аргумента (имеет смысл 2), это преобразование

```
val f: (A, B) => C // a function that takes two arguments of type `A` and `B` respectively
                   // and returns a value of type `C`
```

в

```
val curriedF: A => B => C // a function that take an argument of type `A`
                          // and returns *a function*
                          // that takes an argument of type `B` and returns a `C`
```

Итак, для функций arity-2 мы можем написать функцию curry как:

```
def curry[A, B, C](f: (A, B) => C): A => B => C = {
  (a: A) => (b: B) => f(a, b)
}
```

Использование:

```
val f: (String, Int) => Double = {(_, _) => 1.0}
val curriedF: String => Int => Double = curry(f)
f("a", 1)        // => 1.0
curriedF("a")(1) // => 1.0
```

Scala дает нам несколько функций языка, которые помогают в этом:

1. Вы можете писать валютные функции как методы. так что `curriedF` можно записать как:

```
def curriedFAsAMethod(str: String)(int: Int): Double = 1.0
val curriedF = curriedFAsAMethod _
```

2. Вы можете un-curry (т.е. перейти от `A => B => C` к `(A, B) => C` ) с помощью стандартного библиотечного метода: `Function.uncurried`

```
val f: (String, Int) => Double = Function.uncurried(curriedF)
f("a", 1) // => 1.0
```

## Когда использовать Currying

**Currying** - *это метод перевода оценки функции, которая принимает несколько аргументов в оценку последовательности функций, каждая из которых имеет один аргумент* .

Это обычно полезно, например, когда:

1. различные аргументы функции вычисляются **в разное время** . *(Пример 1)*
2. различные аргументы функции вычисляются **разными уровнями приложения** . *( Пример 2)*

**Пример 1**

Предположим, что общий годовой доход - это функция, состоящая из дохода и бонуса:

```
val totalYearlyIncome:(Int,Int) => Int =  (income, bonus) => income + bonus
```

Версия с установленной версией 2-arity:

```
val totalYearlyIncomeCurried: Int => Int => Int = totalYearlyIncome.curried
```

Обратите внимание, что в приведенном выше определении тип также можно просмотреть /

записать как:

```
Int => (Int => Int)
```

Предположим, что годовая часть дохода известна заранее:

```
val partialTotalYearlyIncome: Int => Int = totalYearlyIncomeCurried(10000)
```

И в какой-то момент вниз по линии бонус известен:

```
partialTotalYearlyIncome(100)
```

**Пример 2.**

Предположим, что производство автомобилей связано с применением автомобильных колес и кузова автомобиля:

```
val carManufacturing:(String,String) => String = (wheels, body) => wheels + body
```

Эти части применяются разными фабриками:

```
class CarWheelsFactory {
  def applyCarWheels(carManufacturing:(String,String) => String): String => String =
          carManufacturing.curried("applied wheels..")
}

class CarBodyFactory {
  def applyCarBody(partialCarWithWheels: String => String): String =
partialCarWithWheels("applied car body..")
}
```

Обратите внимание, что `CarWheelsFactory` над `CarWheelsFactory` выполняет функцию изготовления автомобилей и применяет только колеса.

Затем процесс производства автомобилей примет следующую форму:

```
val carWheelsFactory = new CarWheelsFactory()
val carBodyFactory   = new CarBodyFactory()

val carManufacturing:(String,String) => String = (wheels, body) => wheels + body

val partialCarWheelsApplied: String => String  =
carWheelsFactory.applyCarWheels(carManufacturing)
val carCompleted = carBodyFactory.applyCarBody(partialCarWheelsApplied)
```

## Реальное использование Карри.

У нас есть список кредитных карт, и мы хотели бы рассчитать страховые взносы для всех тех карт, которые компания кредитной карты должна выплатить. Сами премии зависят от

общего количества кредитных карт, поэтому компания соответствующим образом корректирует их.

У нас уже есть функция, которая рассчитывает премию за одну кредитную карту и учитывает общие карты, выпущенные компанией:

```
case class CreditCard(creditInfo: CreditCardInfo, issuer: Person, account: Account)

object CreditCard {
  def getPremium(totalCards: Int, creditCard: CreditCard): Double = { ... }
}
```

Теперь разумным подходом к этой проблеме будет сопоставление каждой кредитной карты с премией и ее сокращение до суммы. Что-то вроде этого:

```
val creditCards: List[CreditCard] = getCreditCards()
val allPremiums = creditCards.map(CreditCard.getPremium).sum //type mismatch; found : (Int,
CreditCard) ⇒ Double required: CreditCard ⇒ ?
```

Однако компилятору это не понравится, потому что `CreditCard.getPremium` требует два параметра. Частичное приложение для спасения! Мы можем частично применить общее количество кредитных карт и использовать эту функцию для сопоставления кредитных карт с их премиями. Все, что нам нужно сделать, это каррировать функцию `getPremium` , изменив ее, чтобы использовать несколько списков параметров, и нам хорошо идти.

Результат должен выглядеть примерно так:

```
object CreditCard {
  def getPremium(totalCards: Int)(creditCard: CreditCard): Double = { ... }
}

val creditCards: List[CreditCard] = getCreditCards()

val getPremiumWithTotal = CreditCard.getPremium(creditCards.length)_

val allPremiums = creditCards.map(getPremiumWithTotal).sum
```

Прочитайте Карринг онлайн: https://riptutorial.com/ru/scala/topic/1636/карринг

# глава 21: Класс опций

## Синтаксис

- class Some [+ T] (значение: T) расширяет опцию [T]
- объект None extends Option [Nothing]
- Опция [T] (значение: T)

  Конструктор создает либо `Some(value)` либо `None` если это необходимо для предоставленного значения.

## Examples

### Варианты в виде коллекций

`Option` s имеет некоторые полезные функции более высокого порядка, которые можно легко понять, просматривая параметры как *коллекции с нулевым или одним элементом* - где `None` ведет себя как пустая коллекция, а `Some(x)` ведет себя как коллекция с одним элементом `x` .

```
val option: Option[String] = ???

option.map(_.trim) // None if option is None, Some(s.trim) if Some(s)
option.foreach(println) // prints the string if it exists, does nothing otherwise
option.forall(_.length > 4) // true if None or if Some(s) and s.length > 4
option.exists(_.length > 4) // true if Some(s) and s.length > 4
option.toList // returns an actual list
```

### Использование опции вместо Null

В Java (и других языках) использование `null` является обычным способом указания того, что для ссылочной переменной не существует значения. В Scala использование `Option` предпочтительнее использования `null` . `Option` обнуляет значения, которые *могут* быть `null` .

`None` является подклассом `Option` обменивает нулевую ссылку. `Some` них являются подклассом `Option` обертывающей ненулевую ссылку.

Облегчение ссылки легко:

```
val nothing = Option(null) // None
val something = Option("Aren't options cool?") // Some("Aren't options cool?")
```

Это типичный код при вызове библиотеки Java, который может возвращать нулевую

ссылку:

```
val resource = Option(JavaLib.getResource())
// if null, then resource = None
// else resource = Some(resource)
```

Если `getResource()` возвращает `null` значение, `resource` будет объектом `None` . В противном случае это будет объект `Some(resource)` . Предпочтительным способом обработки `Option` является использование функций более высокого порядка, доступных в типе `Option` . Например, если вы хотите проверить, не является ли ваше значение `None` (аналогично проверке, если `value == null` ), вы должны использовать функцию `isDefined` :

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isDefined) {  // resource is `Some(_)` type
  val r: Resource = resource.get
  r.connect()
}
```

Аналогично, для проверки `null` ссылки вы можете сделать это:

```
val resource: Option[Resource] = Option(JavaLib.getResource())
if (resource.isEmpty) { // resource is `None` type.
  System.out.println("Resource is empty! Cannot connect.")
}
```

Предпочтительно, чтобы вы лечить условное выполнение на обернутом значении в `Option` ( без использования «» исключительного `Option.get` метода) путем обработки `Option` как монада и используя `foreach` :

```
val resource: Option[Resource] = Option(JavaLib.getResource())
resource foreach (r => r.connect())
// if r is defined, then r.connect() is run
// if r is empty, then it does nothing
```

Если требуется экземпляр `Resource` (по сравнению с экземпляром `Option[Resource]` ), вы все равно можете использовать `Option` для защиты от нулевых значений. Здесь метод `getOrElse` предоставляет значение по умолчанию:

```
lazy val defaultResource = new Resource()
val resource: Resource = Option(JavaLib.getResource()).getOrElse(defaultResource)
```

Java - код не будет легко обращаться в Scala `Option` , поэтому при передаче значений в Java коде это хорошая форма разворачивать в `Option` , передавая `null` или толкового по умолчанию в соответствующих случаях:

```
val resource: Option[Resource] = ???
JavaLib.sendResource(resource.orNull)
JavaLib.sendResource(resource.getOrElse(defaultResource)) //
```

### основы

`Option` представляет собой структуру данных, которая содержит либо одно значение, либо вообще не имеет значения. `Option` можно рассматривать как коллекции нулевого или одного элемента.

Вариант - абстрактный класс с двумя детьми: `Some` и `None` .

`Some` содержат одно значение, а `None` содержит значения.

`Option` полезна в выражениях, которые в противном случае использовали бы `null` для представления отсутствия конкретного значения. Это защищает от `NullPointerException` и позволяет создавать множество выражений, которые могут не возвращать значение с использованием комбинаторов, таких как `Map` , `FlatMap` и т. Д.

## Пример с картой

```
val countries = Map(
  "USA" -> "Washington",
  "UK" -> "London",
  "Germany" -> "Berlin",
  "Netherlands" -> "Amsterdam",
  "Japan" -> "Tokyo"
)

println(countries.get("USA")) // Some(Washington)
println(countries.get("France")) // None
println(countries.get("USA").get) // Washington
println(countries.get("France").get) // Error: NoSuchElementException
println(countries.get("USA").getOrElse("Nope")) // Washington
println(countries.get("France").getOrElse("Nope")) // Nope
```

`Option[A]` **запечатана** и, следовательно, не может быть расширена. Поэтому семантика стабильна и на нее можно положиться.

### Варианты для понимания

`Option` s имеет метод `flatMap` . Это означает, что они могут использоваться для понимания. Таким образом, мы можем поднять регулярные функции для работы с `Option` s без необходимости их переопределения.

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = Option(2)

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = Some(3)
```

Когда одно из значений равно `None` конечный результат вычисления также будет `None` .

```
val firstOption: Option[Int] = Option(1)
val secondOption: Option[Int] = None

val myResult = for {
  firstValue <- firstOption
  secondValue <- secondOption
} yield firstValue + secondValue
// myResult: Option[Int] = None
```

Примечание. Этот шаблон распространяется в более общем плане на понятия, называемые `Monad` s. (Дополнительная информация должна быть доступна на страницах, относящихся к пониманию и `Monad` s)

В общем случае невозможно смешивать разные монады в понимании. Но поскольку `Option` можно легко преобразовать в `Iterable` , мы можем легко смешать `Option` s и `Iterable` s, вызвав метод `.toIterable` .

```
val option: Option[Int] = Option(1)
val iterable: Iterable[Int] = Iterable(2, 3, 4, 5)

// does NOT compile since we cannot mix Monads in a for comprehension
// val myResult = for {
//   optionValue <- option
//   iterableValue <- iterable
//} yield optionValue + iterableValue

// It does compile when adding a .toIterable on the option
val myResult = for {
  optionValue <- option.toIterable
  iterableValue <- iterable
} yield optionValue + iterableValue
// myResult: Iterable[Int] = List(2, 3, 4, 5)
```

Небольшая заметка: если бы мы определили наш подход к пониманию, то другой путь вокруг понимания будет скомпилирован, поскольку наш вариант будет преобразован неявно. По этой причине полезно всегда добавлять эту `.toIterable` (или соответствующую функцию в зависимости от того, какую коллекцию вы используете) для согласованности.

Прочитайте Класс опций онлайн: https://riptutorial.com/ru/scala/topic/2293/класс-опций

# глава 22: Классы и объекты

## Синтаксис

- `class MyClass{} // curly braces are optional here as class body is empty`
- `class MyClassWithMethod {def method: MyClass = ???}`
- `new MyClass() //Instantiate`
- `object MyObject // Singleton object`
- `class MyClassWithGenericParameters[V1, V2](vl: V1, i: Int, v2: V2)`
- `class MyClassWithImplicitFieldCreation[V1](val v1: V1, val i: Int)`
- `new MyClassWithGenericParameters(2.3, 4, 5)` или с другим типом: `new MyClassWithGenericParameters[Double, Any](2.3, 4, 5)`
- `class MyClassWithProtectedConstructor protected[my.pack.age](s: String)`

## Examples

### Выполнить экземпляр класса

Класс в Scala является «планом» экземпляра класса. Экземпляр содержит состояние и поведение, определенные этим классом. Чтобы объявить класс:

```
class MyClass{}  // curly braces are optional here as class body is empty
```

Экземпляр может быть создан с использованием `new` ключевого слова:

```
var instance = new MyClass()
```

или же:

```
var instance = new MyClass
```

Круглые скобки необязательны в Scala для создания объектов из класса с конструктором без аргументов. Если конструктор класса принимает аргументы:

```
class MyClass(arg : Int)       // Class definition
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   // not allowed
```

Здесь `MyClass` требует один аргумент `Int` , который может использоваться только внутри класса. `arg` не может быть доступен за пределами `MyClass` если он не объявлен как поле:

```
class MyClass(arg : Int){
    val prop = arg  // Class field declaration
}

var obj = new MyClass(2)
```

```
obj.prop     // legal statement
```

В качестве альтернативы он может быть объявлен публичным в конструкторе:

```
class MyClass(val arg : Int)   // Class definition with arg declared public
var instance = new MyClass(2)  // Instance instantiation
instance.arg                   //arg is now visible to clients
```

## Создание экземпляра класса без параметров: {} vs ()

Допустим, у нас есть класс MyClass без аргумента конструктора:

```
class MyClass
```

В Scala мы можем создать экземпляр, используя синтаксис ниже:

```
val obj = new MyClass()
```

Или мы можем просто написать:

```
val obj = new MyClass
```

Но, если не обратить внимание, в некоторых случаях необязательная скобка может привести к неожиданному поведению. Предположим, мы хотим создать задачу, которая должна выполняться в отдельном потоке. Ниже приведен пример кода:

```
val newThread = new Thread { new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    }

newThread.start   // prints no output
```

Мы можем подумать, что этот примерный код, если он будет выполнен, будет печатать исполняемую `Performing task.` , но, к нашему удивлению, он ничего не напечатает. Давайте посмотрим, что здесь происходит. Если вы посмотрите ближе, мы использовали фигурные скобки `{}` , сразу после `new Thread` . Он создал анонимный класс, который расширяет `Thread` :

```
val newThread = new Thread {
  //creating anonymous class extending Thread
}
```

А затем в теле этого анонимного класса мы определили нашу задачу (снова создавая аннонимный класс, реализующий интерфейс `Runnable` ). Таким образом, мы могли подумать, что мы использовали `public Thread(Runnable target)` но на самом деле (игнорируя

необязательный `()` ), мы использовали `public Thread()` не имеющий ничего определенного в теле метода `run()` . Чтобы исправить проблему, нам нужно использовать скобки вместо фигурных скобок.

```
val newThread = new Thread ( new Runnable {
        override def run(): Unit = {
            // perform task
            println("Performing task.")
        }
      }
    )
```

Другими словами, здесь `{}` и `()` не являются *взаимозаменяемыми* .

### Объекты Singleton & Companion

## Одиночные объекты

Scala поддерживает статические члены, но не так же, как Java. Scala предоставляет альтернативу этому *объекту Singleton* . Объекты Singleton аналогичны нормальному классу, за исключением того, что они не могут быть созданы с использованием `new` ключевого слова. Ниже приведен пример одноэлементного класса:

```
object Factorial {
    private val cache = Map[Int, Int]()
    def getCache = cache
}
```

Обратите внимание, что мы использовали ключевое слово `object` для определения объекта singleton (вместо «class» или «trait»). Поскольку объекты singleton не могут быть созданы, они не могут иметь параметры. Доступ к объекту singleton выглядит следующим образом:

```
Factorial.getCache() //returns the cache
```

Обратите внимание, что это похоже на доступ к статическому методу в классе Java.

## Сопутствующие объекты

В объектах Scala Singleton можно использовать имя соответствующего класса. В таком сценарии одноэлементный объект упоминается как объект- *компаньон* . Например, под классом `Factorial` определено, и под ним определяется объект-компаньон (также называемый `Factorial` ). По соглашению сопутствующие объекты определяются в том же файле, что и их класс-компаньон.

```
class Factorial(num : Int) {
```

```
  def fact(num : Int) : Int = if (num <= 1) 1 else (num * fact(num - 1))

  def calculate() : Int = {
    if (!Factorial.cache.contains(num)) {    // num does not exists in cache
      val output = fact(num) // calculate factorial
      Factorial.cache += (num -> output)     // add new value in cache
    }

    Factorial.cache(num)
  }
}

object Factorial {
  private val cache = scala.collection.mutable.Map[Int, Int]()
}

val factfive = new Factorial(5)
factfive.calculate  // Calculates the factorial of 5 and stores it
factfive.calculate  // uses cache this time
val factfiveagain = new Factorial(5)
factfiveagain.calculate  // Also uses cache
```

В этом примере мы используем закрытый `cache` для хранения факториала числа, чтобы сэкономить время вычисления для повторных номеров.

Здесь `object Factorial` является сопутствующим объектом, а `class Factorial` - соответствующим классом компаньона. Сопутствующие объекты и классы могут обращаться к `private` членам друг друга. В приведенном выше примере `Factorial` class обращается к частному `cache` своего компаньона.

Обратите внимание, что новый экземпляр класса будет по-прежнему использовать один и тот же объект-компаньон, поэтому любая модификация переменных-членов этого объекта будет перенесена.

## Объекты

Если классы больше похожи на чертежи, объекты статичны (т.е. уже созданы):

```
object Dog {
    def bark: String = "Raf"
}

Dog.bark() // yields "Raf"
```

Они часто используются в качестве компаньона для класса, они позволяют вам писать:

```
class Dog(val name: String) {

}

object Dog {
    def apply(name: String): Dog = new Dog(name)
}
```

```
val dog = Dog("Barky") // Object
val dog = new Dog("Barky") // Class
```

## Проверка типа экземпляра

**Проверка типа** : `variable.isInstanceOf[Type]`

При сопоставлении с образцом (не так полезно в этой форме):

```
variable match {
  case _: Type => true
  case _ => false
}
```

Оба `isInstanceOf` и сопоставление образцов проверяют только тип объекта, а не его общий параметр (без переопределения типа), за исключением массивов:

```
val list: List[Any] = List(1, 2, 3)             //> list  : List[Any] = List(1, 2, 3)

val upcasting = list.isInstanceOf[Seq[Int]]     //> upcasting  : Boolean = true

val shouldBeFalse = list.isInstanceOf[List[String]]
                                                //> shouldBeFalse  : Boolean = true
```

Но

```
val chSeqArray: Array[CharSequence] = Array("a") //> chSeqArray  : Array[CharSequence] =
Array(a)
val correctlyReified = chSeqArray.isInstanceOf[Array[String]]
                                               //> correctlyReified  : Boolean = false


val stringIsACharSequence: CharSequence = ""    //> stringIsACharSequence  : CharSequence = ""

val sArray = Array("a")                         //> sArray  : Array[String] = Array(a)
val correctlyReified = sArray.isInstanceOf[Array[String]]
                                                //> correctlyReified  : Boolean = true

//val arraysAreInvariantInScala: Array[CharSequence] = sArray
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
//Note: String <: CharSequence, but class Array is invariant in type T.
//You may wish to investigate a wildcard type such as `_ <: CharSequence`. (SLS 3.2.10)
//Workaround:
val arraysAreInvariantInScala: Array[_ <: CharSequence] = sArray
                                                //> arraysAreInvariantInScala  : Array[_ <:
CharSequence] = Array(a)


val arraysAreCovariantOnJVM = sArray.isInstanceOf[Array[CharSequence]]
                                                //> arraysAreCovariantOnJVM  : Boolean = true
```

**Тип литья** : `variable.asInstanceOf[Type]`

С учетом соответствия шаблону :

```
variable match {
  case _: Type => true
}
```

Примеры:

```
  val x = 3                                       //> x  : Int = 3
  x match {
    case _: Int => true//better: do something
    case _ => false
  }                                               //> res0: Boolean = true

  x match {
    case _: java.lang.Integer => true//better: do something
    case _ => false
  }                                               //> res1: Boolean = true

  x.isInstanceOf[Int]                             //> res2: Boolean = true

  //x.isInstanceOf[java.lang.Integer]//fruitless type test: a value of type Int cannot also be
a Integer

  trait Valuable { def value: Int}
  case class V(val value: Int) extends Valuable

  val y: Valuable = V(3)                          //> y  : Valuable = V(3)
  y.isInstanceOf[V]                               //> res3: Boolean = true
  y.asInstanceOf[V]                               //> res4: V = V(3)
```

Примечание. Это касается только поведения на JVM, на других платформах (JS, native) тип casting / check может вести себя по-разному.

### Конструкторы

## Первичный конструктор

В Scala основной конструктор является телом класса. За именем класса следует список параметров, которые являются аргументами конструктора. (Как и в любой функции, пустой список параметров может быть опущен.)

```
class Foo(x: Int, y: String) {
    val xy: String = y * x
    /* now xy is a public member of the class */
}

class Bar {
    ...
}
```

Параметры конструкции экземпляра недоступны вне его тела конструктора, если они не

отмечены как член экземпляра ключевым словом `val` :

```
class Baz(val z: String)
// Baz has no other members or methods, so the body may be omitted

val foo = new Foo(4, "ab")
val baz = new Baz("I am a baz")
foo.x // will not compile: x is not a member of Foo
foo.xy // returns "abababab": xy is a member of Foo
baz.z // returns "I am a baz": z is a member of Baz
val bar0 = new Bar
val bar1 = new Bar() // Constructor parentheses are optional here
```

Любые операции, которые должны выполняться при создании экземпляра объекта, записываются непосредственно в тело класса:

```
class DatabaseConnection
    (host: String, port: Int, username: String, password: String) {
    /* first connect to the DB, or throw an exception */
    private val driver = new AwesomeDB.Driver()
    driver.connect(host, port, username, password)
    def isConnected: Boolean = driver.isConnected
    ...
}
```

Обратите внимание, что считается хорошей практикой максимально возможное количество побочных эффектов в конструкторе; вместо вышеуказанного кода следует рассмотреть возможность `connect` и `disconnect` методов, чтобы потребительский код отвечал за планирование IO.

## Вспомогательные конструкторы

Класс может иметь дополнительные конструкторы, называемые «вспомогательными конструкторами». Они определяются определениями конструктора в форме `def this(...) = e` , где `e` должен вызывать другой конструктор:

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}

// usage:
new Person("Grace Hopper").fullName // returns Grace Hopper
new Person("Grace", "Hopper").fullName // returns Grace Hopper
```

Это означает, что каждый конструктор может иметь другой модификатор: только некоторые из них могут быть доступны публично:

```
class Person private(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}
```

```
new Person("Ada Lovelace") // won't compile
new Person("Ada", "Lovelace") // compiles
```

Таким образом, вы можете контролировать, как потребительский код может создавать экземпляр класса.

Прочитайте Классы и объекты онлайн: https://riptutorial.com/ru/scala/topic/2047/классы-и-объекты

# глава 23: Классы классов

## Синтаксис

- case class Foo () // Классы классов без параметров должны иметь пустой список
- класс case Foo (a1: A1, ..., aN: AN) // Создаем класс case с полями a1 ... aN
- case object Bar // Создание класса case singleton

## Examples

### Равномерность класса

Одна функция, предоставляемая бесплатно по классам классов, - это метод автогенерации `equals` значений, который проверяет равенство значений всех отдельных полей-членов вместо проверки ссылочного равенства объектов.

С обычными классами:

```
class Foo(val i: Int)
val a = new Foo(3)
val b = new Foo(3)
println(a == b)// "false" because they are different objects
```

С классами классов:

```
case class Foo(i: Int)
val a = Foo(3)
val b = Foo(3)
println(a == b)// "true" because their members have the same value
```

### Сгенерированные артефакты кода

Модификатор `case` заставляет компилятор Scala автоматически генерировать общий шаблонный код для класса. Внедрение этого кода вручную является утомительным и источником ошибок. Следующее определение класса case:

```
case class Person(name: String, age: Int)
```

... будет автоматически генерироваться следующий код:

```
class Person(val name: String, val age: Int)
  extends Product with Serializable
{
  def copy(name: String = this.name, age: Int = this.age): Person =
    new Person(name, age)
```

```
  def productArity: Int = 2

  def productElement(i: Int): Any = i match {
    case 0 => name
    case 1 => age
    case _ => throw new IndexOutOfBoundsException(i.toString)
  }

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productPrefix: String = "Person"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[Person]

  override def hashCode(): Int = scala.runtime.ScalaRunTime._hashCode(this)

  override def equals(obj: Any): Boolean = this.eq(obj) || obj match {
    case that: Person => this.name == that.name && this.age == that.age
    case _ => false
  }

  override def toString: String =
    scala.runtime.ScalaRunTime._toString(this)
}
```

Модификатор `case` также создает объект-компаньон:

```
object Person extends AbstractFunction2[String, Int, Person] with Serializable {
  def apply(name: String, age: Int): Person = new Person(name, age)

  def unapply(p: Person): Option[(String, Int)] =
    if(p == null) None else Some((p.name, p.age))
}
```

При применении к `object` модификатор `case` имеет похожие (хотя и менее драматические) эффекты. Здесь основными преимуществами являются реализация `toString` и значение `hashCode` , которое согласовано между процессами. Обратите внимание, что объекты case ( правильно) используют ссылочное равенство:

```
object Foo extends Product with Serializable {
  def productArity: Int = 0

  def productIterator: Iterator[Any] =
    scala.runtime.ScalaRunTime.typedProductIterator(this)

  def productElement(i: Int): Any =
    throw new IndexOutOfBoundsException(i.toString)

  def productPrefix: String = "Foo"

  def canEqual(obj: Any): Boolean = obj.isInstanceOf[this.type]

  override def hashCode(): Int = 70822 // "Foo".hashCode()

  override def toString: String = "Foo"
```

```
}
```

По-прежнему можно вручную реализовать методы, которые в противном случае предоставлялись модификатором `case` как в самом классе, так и в его сопутствующем объекте.

## Основы работы в классе

По сравнению с обычными классами - нотация классов случаев дает несколько преимуществ:

- Все аргументы конструктора являются `public` и могут быть доступны для инициализированных объектов (как правило, это не так, как показано здесь):

```
case class Dog1(age: Int)
val x = Dog1(18)
println(x.age) // 18 (success!)

class Dog2(age: Int)
val x = new Dog2(18)
println(x.age) // Error: "value age is not a member of Dog2"
```

- Он предоставляет реализацию для следующих методов: `toString` , `equals` , `hashCode` (на основе свойств), `copy` , `apply` и не `unapply` :

```
case class Dog(age: Int)
val d1 = Dog(10)
val d2 = d1.copy(age = 15)
```

- Он обеспечивает удобный механизм для сопоставления шаблонов:

```
sealed trait Animal // `sealed` modifier allows inheritance within current build-unit
only
case class Dog(age: Int) extends Animal
case class Cat(owner: String) extends Animal
val x: Animal = Dog(18)
x match {
    case Dog(x) => println(s"It's a $x years old dog.")
    case Cat(x) => println(s"This cat belongs to $x.")
}
```

## Классы классов и иммунитет

Компилятор Scala префикс каждого аргумента в списке параметров по умолчанию с `val` . Это означает, что по умолчанию классы case неизменяемы. Каждому параметру присваивается метод доступа, но методы мутатора отсутствуют. Например:

```
case class Foo(i: Int)
```

```
val fooInstance = Foo(1)
val j = fooInstance.i       // get
fooInstance.i = 2           // compile-time exception (mutation: reassignment to val)
```

Объявление параметра в классе case, поскольку `var` переопределяет поведение по умолчанию и делает класс case изменчивым:

```
case class Bar(var i: Int)

val barInstance = Bar(1)
val j = barInstance.i       // get
barInstance.i = 2           // set
```

Другой экземпляр, когда класс case является «изменчивым», - это когда значение в классе case изменено:

```
import scala.collection._

case class Bar(m: mutable.Map[Int, Int])

val barInstance = Bar(mutable.Map(1 -> 2))
barInstance.m.update(1, 3)                  // mutate m
barInstance                                 // Bar(Map(1 -> 3)
```

Обратите внимание, что «мутация», которая происходит здесь, находится на карте, на которую указывает `m` , а не на `m` . Таким образом, если какой-либо другой объект имеет `m` как член, он также увидит изменение. Обратите внимание, как в следующем примере изменение `instanceA` также изменяет `instanceB` :

```
import scala.collection.mutable

case class Bar(m: mutable.Map[Int, Int])

val m = mutable.Map(1 ->2)
val barInstanceA = Bar(m)
val barInstanceB = Bar(m)
barInstanceA.m.update(1,3)
barInstanceA  // Bar = Bar(Map(1 -> 3))
barInstanceB  // Bar = Bar(Map(1 -> 3))
m  // scala.collection.mutable.Map[Int,Int] = Map(1 -> 3)
```

## Создание копии объекта с определенными изменениями

Классы классов предоставляют метод `copy` который создает новый объект, который имеет те же поля, что и старый, с определенными изменениями.

Мы можем использовать эту функцию для создания нового объекта из предыдущего, который имеет одни и те же характеристики. Этот простой класс case демонстрирует эту функцию:

```
case class Person(firstName: String, lastName: String, grade: String, subject: String)
val putu = Person("Putu", "Kevin", "A1", "Math")
val mark = putu.copy(firstName = "Ketut", lastName = "Mark")
// mark: People = People(Ketut,Mark,A1,Math)
```

В этом примере мы видим, что оба объекта имеют схожие характеристики ( `grade = A1` , `subject = Math` ), за исключением тех случаев, когда они были указаны в копии ( `firstName` и `lastName` ).

## Классы с одним элементом для безопасности типов

Для достижения безопасности типа иногда мы хотим избежать использования примитивных типов в нашем домене. Например, представьте `Person` с `name` . Как правило, мы будем кодировать `name` как `String` . Однако, это не было бы трудно смешивать `String` , представляющую `Person` «сек `name` с `String` , представляющим сообщение об ошибке:

```
def logError(message: ErrorMessage): Unit = ???
case class Person(name: String)
val maybeName: Either[String, String] = ??? // Left is error, Right is name
maybeName.foreach(logError) // But that won't stop me from logging the name as an error!
```

Чтобы избежать таких ошибок, вы можете кодировать данные следующим образом:

```
case class PersonName(value: String)
case class ErrorMessage(value: String)
case class Person(name: PersonName)
```

и теперь наш код не будет компилироваться, если мы смешаем `PersonName` с `ErrorMessage` или даже обычную `String` .

```
val maybeName: Either[ErrorMessage, PersonName] = ???
maybeName.foreach(reportError) // ERROR: tried to pass PersonName; ErrorMessage expected
maybeName.swap.foreach(reportError) // OK
```

Но это накладывает небольшие накладные расходы во время выполнения, поскольку теперь нам нужно вставить / удалить `String` s в / из своих контейнеров `PersonName` . Чтобы этого избежать, можно создавать `ErrorMessage` значений `PersonName` и `ErrorMessage` :

```
case class PersonName(val value: String) extends AnyVal
case class ErrorMessage(val value: String) extends AnyVal
```

Прочитайте Классы классов онлайн: https://riptutorial.com/ru/scala/topic/1022/классы-классов

# глава 24: Коллекции

## Examples

### Сортировка списка

Предполагая следующий список, мы можем сортировать различные способы.

```
val names = List("Kathryn", "Allie", "Beth", "Serin", "Alana")
```

Поведение по умолчанию `sorted()` заключается в использовании `math.Ordering` , что для строк результатов в lexographic рода:

```
names.sorted
// results in: List(Alana, Allie, Beth, Kathryn, Serin)
```

`sortWith` позволяет вам предоставить собственный заказ, используя функцию сравнения:

```
names.sortWith(_.length < _.length)
// results in: List(Beth, Allie, Serin, Alana, Kathryn)
```

`sortBy` позволяет вам предоставить функцию преобразования:

```
//A set of vowels to use
val vowels = Set('a', 'e', 'i', 'o', 'u')

//A function that counts the vowels in a name
def countVowels(name: String) = name.count(l => vowels.contains(l.toLower))

//Sorts by the number of vowels
names.sortBy(countVowels)
//result is: List(Kathryn, Beth, Serin, Allie, Alana)
```

Вы всегда можете изменить список или отсортированный список, используя `reverse:

```
names.sorted.reverse
//results in: List(Serin, Kathryn, Beth, Allie, Alana)
```

Списки также могут быть отсортированы с использованием Java-метода `java.util.Arrays.sort` и его оболочки Scala `scala.util.Sorting.quickSort`

```
java.util.Arrays.sort(data)
scala.util.Sorting.quickSort(data)
```

Эти методы могут повысить производительность при сортировке больших коллекций, если можно избежать конверсий коллекции и распаковки / бокса. Для более подробного

обсуждения различий в производительности читайте о сортировке Scala Collection, sortWith и sortBy Performance .

## Создайте список, содержащий n копий x

Чтобы создать коллекцию `n` копий некоторого объекта `x` , используйте метод fill . В этом примере создается `List` , но он может работать с другими коллекциями, для которых `fill` имеет смысл:

```
// List.fill(n)(x)
scala > List.fill(3)("Hello World")
res0: List[String] = List(Hello World, Hello World, Hello World)
```

## Список и векторный чарт

> В настоящее время лучше использовать `Vector` вместо `List` потому что реализации имеют лучшую производительность. Характеристики производительности можно найти здесь . `Vector` можно использовать везде, где используется `List` .

### Создание списка

```
List[Int]()          // Declares an empty list of type Int
List.empty[Int]      // Uses `empty` method to declare empty list of type Int
Nil                  // A list of type Nothing that explicitly has nothing in it

List(1, 2, 3)        // Declare a list with some elements
1 :: 2 :: 3 :: Nil   // Chaining element prepending to an empty list, in a LISP-style
```

### Возьмите элемент

```
List(1, 2, 3).headOption // Some(1)
List(1, 2, 3).head       // 1

List(1, 2, 3).lastOption // Some(3)
List(1, 2, 3).last       // 3, complexity is O(n)

List(1, 2, 3)(1)         // 2, complexity is O(n)
List(1, 2, 3)(3)         // java.lang.IndexOutOfBoundsException: 4
```

### Предварительные элементы

```
0 :: List(1, 2, 3)       // List(0, 1, 2, 3)
```

### Добавить элементы

```
List(1, 2, 3) :+ 4       // List(1, 2, 3, 4), complexity is O(n)
```

### Присоединиться (объединить) списки

```
List(1, 2) ::: List(3, 4) // List(1, 2, 3, 4)
List.concat(List(1,2), List(3, 4)) // List(1, 2, 3, 4)
List(1, 2) ++ List(3, 4)  // List(1, 2, 3, 4)
```

### Общие операции

```
List(1, 2, 3).find(_ == 3)                    // Some(3)
List(1, 2, 3).map(_ * 2)                      // List(2, 4, 6)
List(1, 2, 3).filter(_ % 2 == 1)              // List(1, 3)
List(1, 2, 3).fold(0)((acc, i) => acc + i * i) // 1 * 1 + 2 * 2 + 3 * 3 = 14
List(1, 2, 3).foldLeft("Foo")(_ + _.toString)  // "Foo123"
List(1, 2, 3).foldRight("Foo")(_ + _.toString) // "123Foo"
```

## Коллекция карт Cheatsheet

> Обратите внимание, что это связано с созданием набора типов `Map` , который отличается от метода `map` .

### Создание карты

```
Map[String, Int]()
val m1: Map[String, Int] = Map()
val m2: String Map Int = Map()
```

Карту можно рассматривать как набор `tuples` для большинства операций, где первым элементом является ключ, а второй - значение.

```
val l = List(("a", 1), ("b", 2), ("c", 3))
val m = l.toMap                               // Map(a -> 1, b -> 2, c -> 3)
```

### Получить элемент

```
val m = Map("a" -> 1, "b" -> 2, "c" -> 3)

m.get("a")  // Some(1)
m.get("d")  // None
m("a")      // 1
m("d")      // java.util.NoSuchElementException: key not found: d

m.keys      // Set(a, b, c)
m.values    // MapLike(1, 2, 3)
```

### Добавить элемент (ы)

```
Map("a" -> 1, "b" -> 2) + ("c" -> 3)              // Map(a -> 1, b -> 2, c -> 3)
Map("a" -> 1, "b" -> 2) + ("a" -> 3)              // Map(a -> 3, b -> 2)
Map("a" -> 1, "b" -> 2) ++ Map("b" -> 3, "c" -> 4) // Map(a -> 1, b -> 3, c -> 4)
```

### Общие операции

В операциях, где происходит итерация по карте ( `map` , `find` , `forEach` и т. Д.), Элементами

коллекции являются `tuples` . Параметр функции может либо использовать аксессоры кортежа ( `_1` , `_2` ), либо частичную функцию с блоком корпуса:

```
m.find(_._1 == "a")  // Some((a,1))
m.map {
  case (key, value) => (value, key)
}                    // Map(1 -> a, 2 -> b, 3 -> c)
m.filter(_._2 == 2)  // Map(b -> 2)
m.foldLeft(0){
  case (acc, (key, value: Int)) => acc + value
}                    // 6
```

### Карта и фильтр по коллекции

# карта

«Сопоставление» по коллекции использует функцию `map` для преобразования каждого элемента этой коллекции аналогичным образом. Общий синтаксис:

```
val someFunction: (A) => (B) = ???
collection.map(someFunction)
```

Вы можете предоставить анонимную функцию:

```
collection.map((x: T) => /*Do something with x*/)
```

## Умножение целых чисел на два

```
// Initialize
val list = List(1,2,3)
// list: List[Int] = List(1, 2, 3)

// Apply map
list.map((item: Int) => item*2)
// res0: List[Int] = List(2, 4, 6)

// Or in a more concise way
list.map(_*2)
// res1: List[Int] = List(2, 4, 6)
```

# Фильтр

`filter` используется, когда вы хотите исключить или «отфильтровать» определенные элементы коллекции. Как и в случае с `map` , общий синтаксис принимает функцию, но эта функция должна возвращать `Boolean` :

```
val someFunction: (a) => Boolean = ???
collection.filter(someFunction)
```

Вы можете предоставить анонимную функцию напрямую:

```
collection.filter((x: T) => /*Do something that returns a Boolean*/)
```

## Проверка номеров пар

```
val list = 1 to 10 toList
// list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

// Filter out all elements that aren't evenly divisible by 2
list.filter((item: Int) => item % 2==0)
// res0: List[Int] = List(2, 4, 6, 8, 10)
```

# Дополнительные примеры карт и фильтров

```
case class Person(firstName: String,
                  lastName: String,
                  title: String)

// Make a sequence of people
val people = Seq(
  Person("Millie", "Fletcher", "Mrs"),
  Person("Jim", "White", "Mr"),
  Person("Jenny", "Ball", "Miss") )


// Make labels using map
val labels = people.map( person =>
  s"${person.title}. ${person.lastName}"
)

// Filter the elements beginning with J
val beginningWithJ = people.filter(_.firstName.startsWith("J"))

// Extract first names and concatenate to a string
val firstNames = people.map(_.firstName).reduce( (a, b) => a + "," + b )
```

### Введение в коллекции Scala

Рамка Scala Collections, по словам ее авторов , предназначена для простой в использовании, кратким, безопасным, быстрым и универсальным.

Структура состоит из черт Scala, которые предназначены для создания блоков для создания коллекций. Для получения дополнительной информации об этих строительных

блоках ознакомьтесь с официальным обзором коллекций Scala .

Эти встроенные коллекции разделяются на неизменяемые и изменяемые пакеты. По умолчанию используются неизменяемые версии. Построение `List()` (без импорта чего-либо) приведет к созданию *неизменяемого* списка.

Одной из самых мощных функций платформы является последовательный и простой в использовании интерфейс в коллекциях с единомышленниками. Например, суммирование всех элементов в коллекции одинаково для списков, наборов, векторов, секций и массивов:

```
val numList = List[Int](1, 2, 3, 4, 5)
numList.reduce((n1, n2) => n1 + n2)  // 15

val numSet = Set[Int](1, 2, 3, 4, 5)
numSet.reduce((n1, n2) => n1 + n2)   // 15

val numArray = Array[Int](1, 2, 3, 4, 5)
numArray.reduce((n1, n2) => n1 + n2) // 15
```

Эти единомышленники наследуют свойство `Traversable` .

> В настоящее время лучше использовать `Vector` вместо `List` потому что реализации имеют лучшую производительность. Характеристики производительности можно найти здесь . `Vector` можно использовать везде, где используется `List` .

## Трассируемые типы

Классы коллекции, которые имеют свойство `Traversable` trait `foreach` и наследуют множество методов для выполнения общих операций с коллекциями, которые все одинаково функционируют. Наиболее распространенные операции перечислены здесь:

- Map - `map` , `flatMap` и `collect` новые коллекции, применяя функцию к каждому элементу в исходной коллекции.

```
List(1, 2, 3).map(num => num * 2) // double every number = List(2, 4, 6)

// split list of letters into individual strings and put them into the same list
List("a b c", "d e").flatMap(letters => letters.split(" ")) // = List("a", "b", "c", "d", "e")
```

- Конверсии - `toList` , `toArray` и многие другие операции преобразования меняют текущую коллекцию на более конкретный вид коллекции. Обычно это методы с добавлением «to» и более конкретный тип (т.е. «toList» преобразуется в `List` ).

```
val array: Array[Int] = List[Int](1, 2, 3).toArray // convert list of ints to array of ints
```

- Информация о размере - `isEmpty` , `nonEmpty` , `size` и `hasDefiniteSize` - это все метаданные

о наборе. Это позволяет использовать условные операции в коллекции или для определения размера коллекции, включая ее бесконечность или дискретность.

```
List().isEmpty // true
List(1).nonEmpty // true
```

- Поиск элемента - `head` , в `last` , `find` , и их `Option` варианты используются для получения первого или последнего элемента, или найти определенный элемент в коллекции.

```
val list = List(1, 2, 3)
list.head // = 1
list.last // = 3
```

- Операции по извлечению подсетей - `filter` , `tail` , `slice` , `drop` и другие операции позволяют выбирать элементы коллекции для дальнейшей работы.

```
List(-2, -1, 0, 1, 2).filter(num => num > 0) // = List(1, 2)
```

- Операции подразделения - `partition` , `splitAt` , `span` и `groupBy` разделяют текущую коллекцию на разные части.

```
// split numbers into < 0 and >= 0
List(-2, -1, 0, 1, 2).partition(num => num < 0) // = (List(-2, -1), List(0, 1, 2))
```

- Тесты элементов - `exists` , `forall` и `count` - это операции, используемые для проверки этой коллекции, чтобы увидеть, удовлетворяет ли она предикату.

```
List(1, 2, 3, 4).forall(num => num > 0) // = true, all numbers are positive
List(-3, -2, -1, 1).forall(num => num < 0) // = false, not all numbers are negative
```

- Складки - `foldLeft` ( `/:` ), `foldRight` ( `:\` ), `reduceLeft` и `reduceRight` используется для применения бинарных функций для последовательных элементов в коллекции. Перейдите сюда, чтобы свернуть примеры и перейдите сюда, чтобы уменьшить примеры .

### складка

Метод `fold` выполняет итерацию по коллекции, используя начальное значение аккумулятора и применяя функцию, которая использует каждый элемент для успешного обновления накопителя:

```
val nums = List(1,2,3,4,5)
var initialValue:Int = 0;
var sum = nums.fold(initialValue){
  (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 15 because 0+1+2+3+4+5 = 15
```

В приведенном выше примере анонимная функция была отправлена в `fold()` . Вы также можете использовать именованную функцию, которая принимает два аргумента. Принимая это в моем, приведенный выше пример можно переписать следующим образом:

```
def sum(x: Int, y: Int) = x+ y
val nums = List(1, 2, 3, 4, 5)
var initialValue: Int = 0
val sum = nums.fold(initialValue)(sum)
println(sum) // prints 15 because 0 + 1 + 2 + 3 + 4 + 5 = 15
```

Изменение начального значения повлияет на результат:

```
initialValue = 2;
sum = nums.fold(initialValue){
 (accumulator,currentElementBeingIterated) => accumulator + currentElementBeingIterated
}
println(sum) //prints 17 because 2+1+2+3+4+5 = 17
```

`fold` метод имеет два варианта - `foldLeft` и `foldRight` .

`foldLeft()` выполняет `foldLeft()` слева направо (от первого элемента коллекции до последнего в этом порядке). `foldRight()` выполняет итерацию справа налево (от последнего элемента до первого элемента). `fold()` выполняет `foldLeft()` слева направо, например `foldLeft()` . Фактически, `fold()` фактически вызывает `foldLeft()` внутри.

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1 = foldLeft(z)(op)
```

`fold()` , `foldLeft()` и `foldRight()` возвращают значение, имеющее тот же тип, с начальным значением, которое требуется. Однако, в отличие от `foldLeft()` и `foldRight()` , начальное значение, данное `fold()` может быть только одного типа или супертипа типа коллекции.

В этом примере порядок не имеет значения, поэтому вы можете изменить `fold()` на `foldLeft()` или `foldRight()` и результат останется таким же. Использование функции, чувствительной к порядку, изменит результаты.

Если есть сомнения, предпочитайте `foldLeft()` над `foldRight()` . `foldRight()` менее результативен.

## Для каждого

`foreach` необычен среди итераторов коллекций тем, что он не возвращает результат. Вместо этого он применяет функцию к каждому элементу, который имеет только побочные эффекты. Например:

```
scala> val x = List(1,2,3)
x: List[Int] = List(1, 2, 3)

scala> x.foreach { println }
```

```
1
2
3
```

Функция, предоставленная `foreach` может иметь любой тип возврата, но результат будет отброшен . Обычно `foreach` используется, когда желательны побочные эффекты. Если вы хотите преобразовать данные, используйте `map` , `filter` , `for comprehension` или другой вариант.

**Пример отбрасывания результатов**

```
def myFunc(a: Int) : Int = a * 2
List(1,2,3).foreach(myFunc) // Returns nothing
```

## уменьшить

`reduceLeft()` `reduce()` , `reduceLeft()` и `reduceRight` похожи на складки. Функция, переданная для уменьшения, принимает два значения и дает третью. При работе в списке первые два значения являются первыми двумя значениями в списке. Результат функции и следующее значение в списке затем повторно применяются к функции, что дает новый результат. Этот новый результат применяется со следующим значением списка и так далее, пока не будет больше элементов. Окончательный результат возвращается.

```
val nums = List(1,2,3,4,5)
sum = nums.reduce({ (a, b) => a + b })
println(sum) //prints 15

val names = List("John","Koby", "Josh", "Matilda", "Zac", "Mary Poppins")

def findLongest(nameA:String, nameB:String):String = {
  if (nameA.length > nameB.length) nameA else nameB
}

def findLastAlphabetically(nameA:String, nameB:String):String = {
  if (nameA > nameB) nameA else nameB
}

val longestName:String = names.reduce(findLongest(_,_))
println(longestName) //prints Mary Poppins

//You can also omit the arguments if you want
val lastAlphabetically:String = names.reduce(findLastAlphabetically)
println(lastAlphabetically) //prints Zac
```

Существуют некоторые различия в том, как работают функции сокращения по сравнению с функциями fold. Они есть:

1. Функции уменьшения не имеют начального значения аккумулятора.
2. Сокращение функций нельзя вызывать в пустых списках.
3. Функции сокращения могут возвращать только тип или супертип списка.

Прочитайте Коллекции онлайн: https://riptutorial.com/ru/scala/topic/686/коллекции

# глава 25: Комбинировщики Parser

## замечания

### Случаи ParseResult

`ParseResult` выпускается в трех вариантах:

- Успех, с отметкой о начале матча и следующем символе, который будет соответствовать.
- Отказ, с отметкой о начале того, где была предпринята попытка матча. В этом случае синтаксический анализатор возвращается в это положение, где это будет происходить при разборе парсинга.
- Ошибка, которая останавливает разбор. Никакой обратный отсчет или дальнейший синтаксический анализ не происходит.

## Examples

### Основной пример

```
import scala.util.parsing.combinator._

class SimpleParser extends RegexParsers {
  // Define a grammar rule, turn it into a regex, and apply it the input.
  def word: Parser[String] = """[A-Z][a-z]+""".r ^^ { _.toString }
}

object SimpleParser extends SimpleParser {
  val parseAlice = parse(word, "Alice went to Alamo Square.")
  val parseBarb = parse(word, "barb went Upside Down.")
}

//Successfully finds a match
println(SimpleParser.parseAlice)
//Fails to find a match
println(SimpleParser.parseBarb)
```

Выход будет следующим:

```
[1.6] parsed: Alice
res0: Unit = ()

[1.1] failure: string matching regex `[A-Z][a-z]+' expected but `b' found

barb went Upside Down.
^
```

`[1.6]` в примере `Alice` указывает, что начало матча находится в позиции `1` , а оставшийся

совпадение первого конца начинается с позиции `6` .

Прочитайте Комбинировщики Parser онлайн: https://riptutorial.com/ru/scala/topic/3730/комбинировщики-parser

# глава 26: Кортеж

## замечания

### Почему кортежи ограничены длиной 23?

Кортежи переписываются как объекты компилятором. У компилятора есть доступ к `Tuple1` через `Tuple22` . Этот произвольный предел был решен разработчиками языка.

### Почему количество кортежей подсчитывается от 0?

`Tuple0` эквивалентно `Unit` .

## Examples

### Создание нового кортежа

Кортеж представляет собой разнородный набор из двух-двадцати двух значений. Кортеж можно определить с помощью круглых скобок. Для кортежей размера `2` (также называемого «парой») есть синтаксис стрелки.

```
scala> val x = (1, "hello")
x: (Int, String) = (1,hello)
scala> val y = 2 -> "world"
y: (Int, String) = (2,world)
scala> val z = 3 → "foo"     //example of using U+2192 RIGHTWARD ARROW
z: (Int, String) = (3,foo)
```

`x` - кортеж размера два. Чтобы получить доступ к элементам кортежа, используйте `._1` , через `._22` . Например, мы можем использовать `x._1` для доступа к первому элементу кортежа `x` . `x._2` обращается ко второму элементу. Более элегантно, вы можете использовать экстракторы кортежей .

Синтаксис стрелки для создания кортежей размера два в основном используется в Картах, которые представляют собой коллекции `(key -> value)` :

```
scala> val m = Map[Int, String](2 -> "world")
m: scala.collection.immutable.Map[Int,String] = Map(2 -> world)

scala> m + x
res0: scala.collection.immutable.Map[Int,String] = Map(2 -> world, 1 -> hello)

scala> (m + x).toList
res1: List[(Int, String)] = List((2,world), (1,hello))
```

Синтаксис для пары на карте - это синтаксис стрелки, дающий понять, что 1 - это ключ, а а

- значение, связанное с этим ключом.

## Кортежи в коллекциях

Кортежи часто используются в коллекциях, но их нужно обрабатывать определенным образом. Например, учитывая следующий список кортежей:

```
scala> val l = List(1 -> 2, 2 -> 3, 3 -> 4)
l: List[(Int, Int)] = List((1,2), (2,3), (3,4))
```

Кажется естественным добавить элементы вместе, используя неявную распаковку:

```
scala> l.map((e1: Int, e2: Int) => e1 + e2)
```

Однако это приводит к следующей ошибке:

```
<console>:9: error: type mismatch;
 found   : (Int, Int) => Int
 required: ((Int, Int)) => ?
              l.map((e1: Int, e2: Int) => e1 + e2)
```

Scala не может неявно распаковать кортежи таким образом. У нас есть две возможности исправить эту карту. Первый заключается в использовании позиционных аксессуаров `_1` и `_2` :

```
scala> l.map(e => e._1 + e._2)
res1: List[Int] = List(3, 5, 7)
```

Другой вариант - использовать оператор `case` для распаковки кортежей с использованием сопоставления с образцом:

```
scala> l.map{ case (e1: Int, e2: Int) => e1 + e2}
res2: List[Int] = List(3, 5, 7)
```

Эти ограничения применяются для любой функции более высокого порядка, применяемой к набору кортежей.

Прочитайте Кортеж онлайн: https://riptutorial.com/ru/scala/topic/4971/кортеж

# глава 27: Лучшие практики

## замечания

> Предпочитают vals, неизменные объекты и методы без побочных эффектов. Сначала доходите до них. Используйте vars, изменяемые объекты и методы с побочными эффектами, когда у вас есть определенная потребность и оправдание для них.

- *Программирование в Scala* , Odersky, Spoon и Venners

В этом выступлении Одерского есть больше примеров и рекомендаций.

## Examples

### Будь проще

Не перегружайте простые задачи. Большую часть времени вам потребуется только:

- алгебраические типы данных
- структурная рекурсия
- monad-like api ( `map` , `flatMap` , `fold` )

В Scala есть много сложных вещей, таких как:

- `Cake pattern` или `Reader Monad` для `Reader Monad` для инъекций зависимостей.
- Передача произвольных значений в виде `implicit` аргументов.

Эти вещи не понятны для новичков: не используйте их, прежде чем понимать их. Использование передовых концепций без реальной необходимости сводит на нет код, делая его *менее* удобным.

### Не ставьте слишком много в одном выражении.

- Найдите значащие имена для вычислительных единиц.
- Используйте `for` понимания или `map` чтобы комбинировать вычисления вместе.

Допустим, у вас есть что-то вроде этого:

```
if (userAuthorized.nonEmtpy) {
  makeRequest().map {
    case Success(respone) =>
      someProcessing(..)
      if (resendToUser) {
        sendToUser(...)
```

```
      }
    ...
  }
}
```

Если все ваши функции возвращают `Either` или другой тип, подобный `Validation` , вы можете написать:

```
for {
  user     <- authorizeUser
  response <- requestToThirdParty(user)
  _        <- someProcessing(...)
} {
  sendToUser
}
```

## Предпочитаете функциональный стиль, разумно

По умолчанию:

- Используйте `val` , а не `var` , где это возможно. Это позволяет вам воспользоваться преимуществами ряда функциональных утилит, включая распределение работы.
- Используйте `recursion` и `comprehensions` s, а не петли.
- Используйте неизменяемые коллекции. Это, по возможности, механизм использования `val` .
- Сосредоточьтесь на преобразованиях данных, логике стиля CQRS, а не CRUD.

Есть веские причины выбирать нефункциональный стиль:

- `var` может использоваться для локального состояния (например, внутри актера).
- `mutable` дает лучшую производительность в определенных ситуациях.

Прочитайте Лучшие практики онлайн: https://riptutorial.com/ru/scala/topic/4376/лучшие-практики

# глава 28: макрос

## Вступление

Макросы - это форма метапрограммирования времени компиляции. Некоторые элементы кода Scala, такие как аннотации и методы, могут быть сделаны для преобразования другого кода при компиляции. Макросы - это обычный код Scala, который работает с типами данных, которые представляют другой код. Плагин [Macro Paradise] [] расширяет возможности макросов за пределами базового языка. [Макро-рай]: http://docs.scala-lang.org/overviews/macros/paradise.html

## Синтаксис

- def x () = macro x_impl // x - макрос, где x_impl используется для преобразования кода
- def macroTransform (annottees: Any *): Any = macro impl // Использовать в аннотации, чтобы сделать их макросами

## замечания

Макросы - это языковая функция, которая должна быть активирована либо путем импорта `scala.language.macros` либо с помощью опции `-language:macros` . Для этого требуются только макроопределения; код, который использует макросы, не должен этого делать.

## Examples

### Макро-аннотация

Эта простая макрокоманда выводит аннотированный элемент как есть.

```
import scala.annotation.{compileTimeOnly, StaticAnnotation}
import scala.reflect.macros.whitebox.Context

@compileTimeOnly("enable macro paradise to expand macro annotations")
class noop extends StaticAnnotation {
  def macroTransform(annottees: Any*): Any = macro linkMacro.impl
}

object linkMacro {
  def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
    import c.universe._

    c.Expr[Any](q"{..$annottees}")
  }
}
```

Аннотации `@compileTimeOnly` генерируют ошибку с сообщением о том, что для использования этого макроса должен быть включен плагин компилятора `paradise` . Инструкции по включению этого через SBT приведены здесь .

Вы можете использовать указанный выше макрос следующим образом:

```
@noop
case class Foo(a: String, b: Int)

@noop
object Bar {
  def f(): String = "hello"
}

@noop
def g(): Int = 10
```

## Макросы метода

Когда метод определяется как макрос, компилятор принимает код, который передается в качестве аргумента, и превращает его в AST. Затем он вызывает реализацию макроса с этим AST, и он возвращает новый AST, который затем сплавляется обратно на его сайт вызова.

```
import reflect.macros.blackbox.Context

object Macros {
  // This macro simply sees if the argument is the result of an addition expression.
  // E.g. isAddition(1+1) and isAddition("a"+1).
  // but !isAddition(1+1-1), as the addition is underneath a subtraction, and also
  // !isAddition(x.+), and !isAddition(x.+(a,b)) as there must be exactly one argument.
  def isAddition(x: Any): Boolean = macro isAddition_impl

  // The signature of the macro implementation is the same as the macro definition,
  // but with a new Context parameter, and everything else is wrapped in an Expr.
  def isAddition_impl(c: Context)(expr: c.Expr[Any]): c.Expr[Boolean] = {
    import c.universe._ // The universe contains all the useful methods and types
    val plusName = TermName("+").encodedName // Take the name + and encode it as $plus
    expr.tree match { // Turn expr into an AST representing the code in isAddition(...)
      case Apply(Select(_, `plusName`), List(_)) => reify(true)
      // Pattern match the AST to see whether we have an addition
      // Above we match this AST
      //             Apply (function application)
      //            /     \
      //         Select  List(_) (exactly one argument)
      // (selection ^ of entity, basically the . in x.y)
      //      /          \
      //    _              \
      //               `plusName` (method named +)
      case _                                      => reify(false)
      // reify is a macro you use when writing macros
      // It takes the code given as its argument and creates an Expr out of it
    }
  }
}
```

Также возможно иметь макросы, которые берут `Tree` качестве аргументов. Подобно тому, как `reify` используется для создания `Expr` s, строковый интерполятор `q` (для квазиквазот) позволяет нам создавать и деконструировать `Tree` s. Обратите внимание, что мы могли бы использовать `q` выше ( `expr.tree` , удивление, само `Tree` ) тоже, но не для демонстрационных целей.

```
// No Exprs, just Trees
def isAddition_impl(c: Context)(tree: c.Tree): c.Tree = {
  import c.universe._
  tree match {
    // q is a macro too, so it must be used with string literals.
    // It can destructure and create Trees.
    // Note how there was no need to encode + this time, as q is smart enough to do it itself.
    case q"${_} + ${_}" => q"true"
    case _              => q"false"
  }
}
```

## Ошибки в макросах

Макросы могут запускать предупреждения и ошибки компилятора с помощью их `Context` .

Скажем, мы особенно переусердствовали, когда дело доходило до плохого кода, и мы хотим отметить каждый экземпляр технической задолженности информационным сообщением компилятора (давайте не будем думать о том, насколько плохая эта идея). Мы можем использовать макрос, который ничего не делает, кроме как излучать такое сообщение.

```
import reflect.macros.blackbox.Context

def debtMark(message: String): Unit = macro debtMark_impl
def debtMarkImpl(c: Context)(message: c.Tree): c.Tree = {
  message match {
    case Literal(Constant(msg: String)) => c.info(c.enclosingPosition, msg, false)
    // false above means "do not force this message to be shown unless -verbose"
    case _                              => c.abort(c.enclosingPosition, "Message must be a
string literal.")
    // Abort causes the compilation to completely fail. It's not even a compile error, where
    // multiple can stack up; this just kills everything.
  }
  q"()" // At runtime this method does nothing, so we return ()
}
```

Кроме того, вместо использования `???` чтобы отметить невыполненный код, мы можем создать два макроса, `!!!` и `?!?` , которые выполняют одну и ту же цель, но выдают предупреждения компилятора. `?!?` выдает предупреждение, и `!!!` приведет к прямой ошибке.

```
import reflect.macros.blackbox.Context

def ?!? : Nothing = macro impl_?!?
```

```
def !!! : Nothing = macro impl_!!!

def impl_?!?(c: Context): c.Tree = {
  import c.universe._
  c.warning(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
  // If someone were to shadow the scala package, scala.Predef.??? would not work, as it
  // would end up referring to the scala that shadows and not the actual scala.
  // ROOTPKG is the very root of the tree, and acts like it is imported anew in every
  // expression. It is actually named _root_, but if someone were to shadow it, every
  // reference to it would be an error. It allows us to safely access ??? and know that
  // it is the one we want.
}

def impl_!!!(c: Context): c.Tree = {
  import c.universe._
  c.error(c.enclosingPosition, "Unimplemented!")
  q"${termNames.ROOTPKG}.scala.Predef.???"
}
```

Прочитайте макрос онлайн: https://riptutorial.com/ru/scala/topic/3808/макрос

# глава 29: Монады

## Examples

### Определение Монады

Неофициально монада представляет собой контейнер элементов, обозначенный как `F[_]` , заполненный двумя функциями: `flatMap` (для преобразования этого контейнера) и `unit` (для создания этого контейнера).

Примеры общей библиотеки включают `List[T]` , `Set[T]` и `Option[T]` .

**Формальное определение**

Monad `M` является параметрическим типом `M[T]` с двумя операциями `flatMap` и `unit` , такими как:

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}

def unit[T](x: T): M[T]
```

Эти функции должны удовлетворять трем законам:

1. *Ассоциативность* : `(m flatMap f) flatMap g = m flatMap (x => f(x) flatMap g)`
   То есть, если последовательность не изменяется, вы можете применять термины в любом порядке. Таким образом, применяя `m` к `f` , а затем применяя результат к `g` получите тот же результат, что и применение `f` к `g` , а затем примените `m` к этому результату.
2. *Левая единица* : `unit(x) flatMap f == f(x)`
   То есть единичная монада `x` плоская карта `f` , эквивалентна применению `f` к `x` .
3. *Правый блок* : `m flatMap unit == m`
   Это «личность»: любая монада, сопоставленная с единицей, вернет монаду, эквивалентную самому себе.

**Пример** :

```
val m = List(1, 2, 3)
def unit(x: Int): List[Int] = List(x)
def f(x: Int): List[Int] = List(x * x)
def g(x: Int): List[Int] = List(x * x * x)
val x = 1
```

1. *Ассоциативность* :

```
(m flatMap f).flatMap(g) == m.flatMap(x => f(x) flatMap g) //Boolean = true
//Left side:
List(1, 4, 9).flatMap(g) // List(1, 64, 729)
//Right side:
 m.flatMap(x => (x * x) * (x * x) * (x * x)) //List(1, 64, 729)
```

2. *Левая единица*

```
unit(x).flatMap(x => f(x)) == f(x)
List(1).flatMap(x => x * x) == 1 * 1
```

3. *Правый блок*

```
//m flatMap unit == m
m.flatMap(unit) == m
List(1, 2, 3).flatMap(x => List(x)) == List(1,2,3) //Boolean = true
```

### Стандартные коллекции - Monads

Большинство стандартных коллекций являются монадами ( `List[T]` , `Option[T]` ) или monad-like ( `Either[T]` , `Future[T]` ). Эти коллекции могут быть легко объединены вместе `for` понимания (которые являются эквивалентным способом написания преобразований `flatMap` ):

```
val a = List(1, 2, 3)
val b = List(3, 4, 5)
for {
  i <- a
  j <- b
} yield(i * j)
```

Вышеупомянутое эквивалентно:

```
a flatMap {
  i => b map {
    j => i * j
  }
}
```

Поскольку монада сохраняет *структуру данных* и действует только на элементы внутри этой структуры, мы можем бесконечно цепные монадические структуры данных, как показано здесь для понимания.

Прочитайте Монады онлайн: https://riptutorial.com/ru/scala/topic/4112/монады

# глава 30: Настройка Scala

## Examples

### В Linux через dpkg

В дистрибутивах на основе Debian, включая Ubuntu, наиболее простым способом является использование файла установки `.deb` . Перейдите на сайт Scala . Выберите версию, которую вы хотите установить, затем прокрутите вниз и найдите `scala-xxxdeb` .

Вы можете установить scala deb из командной строки:

```
sudo dpkg -i scala-x.x.x.deb
```

Чтобы убедиться, что он установлен правильно, в командной строке терминала:

```
which scala
```

Ответ должен быть эквивалентен тому, что вы разместили в переменной PATH. Чтобы убедиться, что scala работает:

```
scala
```

Это должно запустить Scala REPL и сообщить о версии (которая, в свою очередь, должна соответствовать загруженной вами версии).

### Установка Ubuntu через ручную загрузку и настройку

Загрузите предпочтительную версию из Lightbend с помощью `curl` :

```
curl -O http://downloads.lightbend.com/scala/2.xx.x/scala-2.xx.x.tgz
```

Разархивируйте `tar` файл в `/usr/local/share` или `/opt/bin` :

```
unzip scala-2.xx.x.tgz
mv scala-2.xx.x /usr/local/share/scala
```

Добавьте `PATH` в `~/.profile` или `~/.bash_profile` или `~/.bashrc` , включив этот текст в один из этих файлов:

```
$SCALA_HOME=/usr/local/share/scala
export PATH=$SCALA_HOME/bin:$PATH
```

Чтобы убедиться, что он установлен правильно, в командной строке терминала:

```
which scala
```

Ответ должен быть эквивалентен тому, что вы разместили в переменной `PATH` . Чтобы убедиться, что `scala` работает:

```
scala
```

Это должно запустить Scala REPL и сообщить о версии (которая, в свою очередь, должна соответствовать загруженной вами версии).

### Mac OSX через Macports

На компьютерах Mac OSX с установленными MacPorts откройте окно терминала и введите:

```
port list | grep scala
```

Здесь будут перечислены все доступные для Scala пакеты. Чтобы установить один (в этом примере версия 2.11 Scala):

```
sudo port install scala2.11
```

( `2.11` может измениться, если вы хотите установить другую версию.)

Все зависимости будут автоматически установлены и обновлен параметр `$PATH` . Чтобы убедиться, что все работает:

```
which scala
```

Это покажет вам путь к установке Scala.

```
scala
```

Это откроет Scala REPL и сообщит номер версии.

Прочитайте Настройка Scala онлайн: https://riptutorial.com/ru/scala/topic/2921/настройка-scala

# глава 31: Обрабатывающие устройства (меры)

## Синтаксис

- class Meter (val meter: Double) расширяет AnyVal
- тип Meter = Double

## замечания

Рекомендуется использовать классы значений для единиц или выделенной библиотеки для них.

## Examples

### Тип псевдонимов

```
type Meter = Double
```

Этот простой подход имеет серьезные недостатки для обработки элементов, поскольку любой другой тип, который является `Double` будет совместим с ним:

```
type Second = Double
var length: Meter = 3
val duration: Second = 1
length = duration
length = 0d
```

Все перечисленные выше компиляции, поэтому в этом случае единицы могут использоваться только для маркировки типов ввода / вывода для считывателей кода (только для намерения).

### Классы значений

```
case class Meter(meters: Double) extends AnyVal
case class Gram(grams: Double) extends AnyVal
```

Классы значений предоставляют безопасный тип кодирования единиц, даже если для их использования требуется несколько больше символов:

```
var length = Meter(3)
var weight = Gram(4)
```

```
//length = weight //type mismatch; found : Gram required: Meter
```

Расширяя `AnyVal` s, для их использования нет штрафа за выполнение, на уровне JVM это обычные примитивные типы (в данном случае это `Double` ).

Если вы хотите автоматически генерировать другие единицы (например, `Velocity` aka `MeterPerSecond` ), этот подход не самый лучший, хотя есть библиотеки, которые также могут использоваться в этих случаях:

- Squants
- единицы
- ScalaQuantity

Прочитайте Обрабатывающие устройства (меры) онлайн:
https://riptutorial.com/ru/scala/topic/5966/обрабатывающие-устройства--меры-

# глава 32: Обработка XML

## Examples

### Уничтожить или допечатать XML

Утилита `PrettyPrinter` будет «печатать» XML-документы. Следующий фрагмент кода довольно печатает неформатированный xml:

```
import scala.xml.{PrettyPrinter, XML}
val xml = XML.loadString("<a>Alana<b><c>Beth</c><d>Catie</d></b></a>")
val formatted = new PrettyPrinter(150, 4).format(xml)
print(formatted)
```

Это выведет контент, используя ширину страницы `150` и константу отступов в `4` символах пробела:

```
<a>
    Alana
    <b>
        <c>Beth</c>
        <d>Catie</d>
    </b>
</a>
```

Вы можете использовать `XML.loadFile("nameoffile.xml")` для загрузки xml из файла вместо строки.

Прочитайте Обработка XML онлайн: https://riptutorial.com/ru/scala/topic/1453/обработка-xml

# глава 33: Обработка ошибок

## Examples

### Пытаться

Использование Try с `map` , `getOrElse` и `flatMap` :

```
import scala.util.Try

val i = Try("123".toInt)     // Success(123)
i.map(_ + 1).getOrElse(321)  // 124

val j = Try("abc".toInt)     // Failure(java.lang.NumberFormatException)
j.map(_ + 1).getOrElse(321)  // 321

Try("123".toInt) flatMap { i =>
  Try("234".toInt)
    .map(_ + i)
}                            // Success(357)
```

Использование `Try` с сопоставлением с образцом:

```
Try(parsePerson("John Doe")) match {
    case Success(person) => println(person.surname)
    case Failure(ex) => // Handle error ...
}
```

### Или

Различные типы данных для ошибки / успеха

```
def getPersonFromWebService(url: String): Either[String, Person] = {

    val response = webServiceClient.get(url)

    response.webService.status match {
        case 200 => {
            val person = parsePerson(response)
            if(!isValid(person)) Left("Validation failed")
            else Right(person)
        }

        case _ => Left(s"Request failed with error code $response.status")
    }
}
```

Совпадение шаблонов по любому значению

```
getPersonFromWebService("http://some-webservice.com/person") match {
```

```
    case Left(errorMessage) => println(errorMessage)
    case Right(person) => println(person.surname)
}
```

Преобразовать любое значение в параметр

```
val maybePerson: Option[Person] = getPersonFromWebService("http://some-
webservice.com/person").right.toOption
```

### вариант

Использование `null` значений категорически не рекомендуется, если они не взаимодействуют с устаревшим кодом Java, который ожидает `null` . Вместо этого, `Option` следует использовать , когда результат функции может быть что - то ( `Some` ) или ничего ( `None` ).

Блок try-catch более подходит для обработки ошибок, но если функция может законно вернуть ничего, `Option` подходит для использования и проста.

`Option[T]` может быть либо `Some(value)` (содержит значение типа `T` ), либо `None` :

```
def findPerson(name: String): Option[Person]
```

Если ни один человек не найден, `None` может быть возвращено. В противном случае возвращается объект типа `Some` содержащий объект `Person` . Ниже приведены способы обработки объекта типа `Option` .

## Соответствие шаблону

```
findPerson(personName) match {
    case Some(person) => println(person.surname)
    case None => println(s"No person found with name $personName")
}
```

## Использование карты и getOrElse

```
val name = findPerson(personName).map(_.firstName).getOrElse("Unknown")
println(name) // Prints either the name of the found person or "Unknown"
```

## Использование сгиба

```
val name = findPerson(personName).fold("Unknown")(_.firstName)
// equivalent to the map getOrElse example above.
```

## Преобразование в Java

Если вам требуется преобразовать тип `Option` в нулевой Java-тип для взаимодействия:

```
val s: Option[String] = Option("hello")
s.orNull           // "hello": String
s.getOrElse(null)  // "hello": String

val n: Option[Int] = Option(42)
n.orNull           // compilation failure (Cannot prove that Null <:< Int.)
n.getOrElse(null)  // 42
```

### Обработка ошибок, возникающих в фьючерсах

Когда `exception` выбрасывается из `Future` , вы можете (должны) использовать `recover` для его обработки.

Например,

```
def runFuture: Future = Future { throw new FairlyStupidException }

val itWillBeAwesome: Future = runFuture
```

... выкинет `Exception` из `Future` . Но, видя, что мы можем с высокой вероятностью предсказать, что `Exception` типа `FairlyStupidException` с большой вероятностью, мы можем специально обработать этот случай элегантным способом:

```
val itWillBeAwesomeOrIllRecover = runFuture recover {
    case stupid: FairlyStupidException =>
         BadRequest("Another stupid exception!")
}
```

Как вы видите, метод, данный для `recover` является `PartialFunction` над доменом всех `Throwable` , поэтому вы можете обрабатывать только несколько типов, а затем позволить остальным перейти в эфир обработки исключений на более высоких уровнях в стеке `Future` .

Обратите внимание, что это похоже на запуск следующего кода в контексте, не относящемся к `Future` :

```
def runNotFuture: Unit = throw new FairlyStupidException

try {
    runNotFuture
} catch {
    case e: FairlyStupidException => BadRequest("Another stupid exception!")
}
```

Очень важно обрабатывать исключения, созданные в `Future` потому что большую часть

времени они более коварны. Обычно они не получают все на вашем лице, потому что они работают в другом контексте исполнения и потоке и, следовательно, не предлагают вам исправить их, когда они происходят, особенно если вы ничего не замечаете в журналах или поведение приложение.

### Использование предложений try-catch

В дополнение к функциональным конструкциям, таким как `Try` , `Option` и `Either` для обработки ошибок, Scala также поддерживает синтаксис, подобный Java, с использованием предложения try-catch (с потенциальным, наконец, блоком). Предложение catch - это совпадение с шаблоном:

```
try {
  // ... might throw exception
} catch {
  case ioe: IOException => ... // more specific cases first
  case e: Exception => ...
  // uncaught types will be thrown
} finally {
  // ...
}
```

### Преобразование исключений в типы или типы

Для преобразования исключения в `Either` или `Option` тип, вы можете использовать методы, предусмотренные в `scala.util.control.Exception`

```
import scala.util.control.Exception._

val plain = "71a"
val optionInt: Option[Int] = catching(classOf[java.lang.NumberFormatException]) opt {
plain.toInt }
val eitherInt = Either[Throwable, Int] = catching(classOf[java.lang.NumberFormatException])
either { plain.toInt }
```

Прочитайте Обработка ошибок онлайн: https://riptutorial.com/ru/scala/topic/910/обработка-ошибок

# глава 34: Объем

## Вступление

Область Scala определяет, где можно получить доступ к значению ( `def` , `val` , `var` или `class` ).

## Синтаксис

- декларация
- частная декларация
- частное [это] заявление
- частная декларация [fromWhere]
- защищенная декларация
- защищенная декларация [fromWhere]

## Examples

### Открытая (по умолчанию) область

По умолчанию область видимости является `public` , доступ к которой можно получить из любого места.

```
package com.example {
  class FooClass {
    val x = "foo"
  }
}

package an.other.package {
  class BarClass {
    val foo = new com.example.FooClass
    foo.x // <- Accessing a public value from another package
  }
}
```

### Частный охват

Когда область является частной, ее можно получить только из текущего класса или других экземпляров текущего класса.

```
package com.example {
  class FooClass {
    private val x = "foo"
    def aFoo(otherFoo: FooClass) {
      otherFoo.x // <- Accessing from another instance of the same class
```

```
    }
  }
  class BarClass {
    val f = new FooClass
    f.x // <- This will not compile
  }
}
```

### Частный пакетный охват

Вы можете указать пакет, в котором можно получить доступ к частному значению.

```
package com.example {
  class FooClass {
    private val x = "foo"
    private[example] val y = "bar"
  }
  class BarClass {
    val f = new FooClass
    f.x // <- Will not compile
    f.y // <- Will compile
  }
}
```

### Личный охват объекта

Наиболее ограничительной областью является область *«объект-частный»* , которая позволяет только доступ к этому значению из одного и того же экземпляра объекта.

```
class FooClass {
  private[this] val x = "foo"
  def aFoo(otherFoo: FooClass) = {
    otherFoo.x // <- This will not compile, accessing x outside the object instance
  }
}
```

### Защищенная область действия

Защищенная область позволяет доступ к значению из любых подклассов текущего класса.

```
class FooClass {
  protected val x = "foo"
}
class BarClass extends FooClass {
  val y = x // It is a subclass instance, will compile
}
class ClassB {
  val f = new FooClass
  f.x // <- This will not compile
}
```

### Защищенный пакет

Область, защищенная пакетом, позволяет доступ к значению только из любого подкласса в определенном пакете.

```
package com.example {
  class FooClass {
    protected[example] val x = "foo"
  }
  class ClassB extends FooClass {
    val y = x // It's in the protected scope, will compile
  }
}
package com {
  class BarClass extends com.example.FooClass {
    val y = x // <- Outside the protected scope, will not compile
  }
}
```

Прочитайте Объем онлайн: https://riptutorial.com/ru/scala/topic/9705/объем

# глава 35: Операторы в Scala

## Examples

### Встроенные операторы

Scala имеет следующие встроенные операторы (методы / языковые элементы с предопределенными правилами приоритета):

| Тип | Условное обозначение | пример |
|---|---|---|
| Арифметические операторы | `+ - * / %` | `a + b` |
| Операторы отношения | `== != > < >= <=` | `a > b` |
| Логические операторы | `&& & \|\| \| !` | `a && b` |
| Побитовые операторы | `& \| ^ ~ << >> >>>` | `a & b` , `~a` , `a >>> b` |
| Операторы присваивания | `= += -= *= /= %= <<= >>= &= ^= \|=` | `a += b` |

Операторы Scala имеют то же значение, что и в Java

**Примечание** : методы, заканчивающиеся на `:` привязка к правой (и правой ассоциативной), поэтому вызов со `list.::(value)` может быть записан как `value :: list` с синтаксисом оператора. ( `1 :: 2 :: 3 :: Nil` совпадает с `1 :: (2 :: (3 :: Nil))` )

### Перегрузка оператора

В Scala вы можете определить своих собственных операторов:

```
class Team {
   def +(member: Person) = ...
}
```

С приведенными выше определениями вы можете использовать его так:

```
ITTeam + Jack
```

или же

```
ITTeam.+(Jack)
```

Чтобы определить унарные операторы, вы можете префикс его с помощью `unary_` .

Например, `unary_!`

```
class MyBigInt {
   def unary_! = ...
}

var a: MyBigInt = new MyBigInt
var b = !a
```

## Приоритет оператора

| категория | оператор | Ассоциативность |
|---|---|---|
| постфикс | `() []` | Слева направо |
| Одинарный | `! ~` | Справа налево |
| Multiplicative | `* / %` | Слева направо |
| присадка | `+ -` | Слева направо |
| сдвиг | `>> >>> <<` | Слева направо |
| реляционный | `> >= < <=` | Слева направо |
| равенство | `== !=` | Слева направо |
| Побитовое и | `&` | Слева направо |
| Побитовое xor | `^` | Слева направо |
| Побитовое или | `\|` | Слева направо |
| Логические и | `&&` | Слева направо |
| Логические или | `\|\|` | Слева направо |
| присваивание | `= += -= *= /= %= >>= <<= &= ^= \| =` | Справа налево |
| запятая | `,` | Слева направо |

Программирование в Scala дает следующий контур, основанный на 1-м символе оператора. Eg `>` - 1-й символ в операторе `>>>` :

| оператор |
|---|
| (все другие специальные символы) |
| `*` |

| оператор |
| --- |
| `/ %` |
| `+ -` |
| `:` |
| `= !` |
| `< >` |
| `&` |
| `^` |
| `\|` |
| (все буквы) |
| (все операторы присваивания) |

Единственное исключение из этого правила касается *операторов присваивания* , например, `+=` , `*=` и т. Д. Если оператор заканчивается равным символом (=) и не является одним из операторов сравнения `<=` , `>=` , `==` or `!=` , то приоритет оператора такой же, как и простое назначение. Другими словами, он ниже, чем у любого другого оператора.

Прочитайте Операторы в Scala онлайн: https://riptutorial.com/ru/scala/topic/6604/операторы-в-scala

# глава 36: отражение

## Examples

### Загрузка класса с использованием отражения

```
import scala.reflect.runtime.universe._
val mirror = runtimeMirror(getClass.getClassLoader)
val module = mirror.staticModule("org.data.TempClass")
```

Прочитайте отражение онлайн: https://riptutorial.com/ru/scala/topic/5824/отражение

# глава 37: пакеты

## Вступление

Пакеты в Scala управляют пространствами имен в больших программах. Например, имя `connection` может возникать в пакетах `com.sql` и `org.http` . Вы можете использовать полностью соответствующие `com.sql.connection` и `org.http.connection` , соответственно, для доступа к каждому из этих пакетов.

## Examples

### Структура упаковки

```
package com {
    package utility {
        package serialization {
            class Serializer
            ...
        }
    }
}
```

### Пакеты и файлы

Предложение пакета напрямую не привязывается к файлу, где он найден. Можно найти общие элементы предложения пакета в разных файлах. Например, предложения пакета ниже могут быть найдены в файле math1.scala и в файле math2.scala.

Файл math1.scala

```
package org {
    package math {
        package statistics {
            class Interval
        }
    }
}
```

Файл math2.scala

```
package org {
    package math{
        package probability {
            class Density
        }
    }
}
```

Файл study.scala

```
import org.math.probability.Density
import org.math.statistics.Interval

object Study {

    def main(args: Array[String]): Unit = {
        var a = new Interval()
        var b = new Density()
    }
}
```

## Назначение именования пакетов

Пакеты Scala должны соответствовать соглашениям об именах пакетов Java.
Имена пакетов записываются в нижнем регистре, чтобы избежать конфликтов с именами классов или интерфейсов. Компании используют свое обратное доменное имя в Интернете, чтобы начать имена своих пакетов, например,

```
io.super.math
```

Прочитайте пакеты онлайн: https://riptutorial.com/ru/scala/topic/8231/пакеты

# глава 38: Параллельные коллекции

## замечания

Параллельные коллекции облегчают параллельное программирование, скрывая детали параллелизма низкого уровня. Это упрощает использование многоядерных архитектур. Примеры параллельных коллекций включают `ParArray` , `ParVector` , `mutable.ParHashMap` , `immutable.ParHashMap` и `ParRange` . Полный список можно найти в документации .

## Examples

### Создание и использование параллельных коллекций

Чтобы создать параллельную коллекцию из последовательной коллекции, вызовите метод `par` . Чтобы создать последовательный сбор из параллельной коллекции, вызовите метод `seq` . В этом примере показано, как вы превращаете обычный `Vector` в `ParVector` , а затем снова:

```
scala> val vect = (1 to 5).toVector
vect: Vector[Int] = Vector(1, 2, 3, 4, 5)

scala> val parVect = vect.par
parVect: scala.collection.parallel.immutable.ParVector[Int] = ParVector(1, 2, 3, 4, 5)

scala> parVect.seq
res0: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4, 5)
```

Метод `par` может быть привязан, что позволяет конвертировать последовательную коллекцию в параллельную коллекцию и немедленно выполнять действие над ней:

```
scala> vect.map(_ * 2)
res1: scala.collection.immutable.Vector[Int] = Vector(2, 4, 6, 8, 10)

scala> vect.par.map(_ * 2)
res2: scala.collection.parallel.immutable.ParVector[Int] = ParVector(2, 4, 6, 8, 10)
```

В этих примерах работа фактически распределяется по нескольким модулям обработки, а затем снова объединяется после завершения работы без вмешательства разработчика.

### Ловушки

**Не используйте параллельные коллекции, когда элементы коллекции должны быть получены в определенном порядке.**

Параллельные коллекции выполняют операции одновременно. Это означает, что вся

работа делится на части и распространяется на разные процессоры. Каждый процессор не знает о работе, выполняемой другими. Если *порядок сбора* имеет значение, то работа, обрабатываемая параллельно, недетерминирована. (Выполнение того же кода дважды может дать разные результаты.)

---

### Неассоциативные операции

Если операция неассоциативна (если порядок выполнения имеет значение), то результат в распараллеленном наборе будет недетерминированным.

```
scala> val list = (1 to 1000).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10...

scala> list.reduce(_ - _)
res0: Int = -500498

scala> list.reduce(_ - _)
res1: Int = -500498

scala> list.reduce(_ - _)
res2: Int = -500498

scala> val listPar = list.par
listPar: scala.collection.parallel.immutable.ParSeq[Int] = ParVector(1, 2, 3, 4, 5, 6, 7, 8,
9, 10...

scala> listPar.reduce(_ - _)
res3: Int = -408314

scala> listPar.reduce(_ - _)
res4: Int = -422884

scala> listPar.reduce(_ - _)
res5: Int = -301748
```

---

### Побочные эффекты

Операции, которые имеют побочные эффекты, такие как `foreach` , могут не выполняться по желанию при параллельных сборах из-за условий гонки. Избегайте этого, используя функции, которые не имеют побочных эффектов, таких как `reduce` или `map` .

```
scala> val wittyOneLiner = Array("Artificial", "Intelligence", "is", "no", "match", "for",
"natural", "stupidity")

scala> wittyOneLiner.foreach(word => print(word + " "))
Artificial Intelligence is no match for natural stupidity

scala> wittyOneLiner.par.foreach(word => print(word + " "))
match natural is for Artificial no stupidity Intelligence

scala> print(wittyOneLiner.par.reduce{_ + " " + _})
Artificial Intelligence is no match for natural stupidity
```

```
scala> val list = (1 to 100).toList
list: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15...
```

Прочитайте Параллельные коллекции онлайн: https://riptutorial.com/ru/scala/topic/3882/параллельные-коллекции

# глава 39: Перегрузка оператора

## Examples

### Определение пользовательских операторов Infix

Операторы Scala (такие как `+` , `-` , `*` , `++` и т. Д.) - это просто методы. Например, `1 + 2` можно записать как `1.+(2)` . Такие методы называются *«инфиксными операторами»* .

Это означает, что пользовательские методы могут быть определены на ваших собственных типах, повторно используя эти операторы:

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)

    new Matrix(rows, cols, newData)
  }
}
```

Эти операторы, определенные как-методы, могут быть использованы следующим образом:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val b = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))

// could also be written a.+(b)
val sum = a + b
```

Обратите внимание, что операторы infix могут иметь только один аргумент; объект перед тем, как оператор вызовет его собственный оператор на объект после оператора. Любой метод Scala с единственным аргументом может использоваться как оператор инфикса.

> Это следует использовать с участием. Обычно это считается хорошей практикой, только если ваш собственный метод делает именно то, чего можно ожидать от этого оператора. В случае сомнений используйте более консервативное название, например `add` вместо `+` .

### Определение пользовательских унарных операторов

Унарные операторы могут быть определены путем добавления оператора с помощью `unary_` . Унарные операторы ограничены `unary_+` , `unary_-` , `unary_!` и `unary_~` :

```
class Matrix(rows: Int, cols: Int, val data: Seq[Seq[Int]]){
  def +(that: Matrix) = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) + that.data(r)(c)
```

```
    new Matrix(rows, cols, newData)
  }

  def unary_- = {
    val newData = for (r <- 0 until rows) yield
      for (c <- 0 until cols) yield this.data(r)(c) * -1

    new Matrix(rows, cols, newData)
  }
}
```

Унарный оператор можно использовать следующим образом:

```
val a = new Matrix(2, 2, Seq(Seq(1,2), Seq(3,4)))
val negA = -a
```

> Это следует использовать с участием. Перегрузка унарного оператора с определением, которое не является ожидаемым, может привести к путанице кода.

Прочитайте Перегрузка оператора онлайн: https://riptutorial.com/ru/scala/topic/2271/перегрузка-оператора

# глава 40: Перечисления

## замечания

Подход с `sealed trait case objects` и `case objects` является предпочтительным, поскольку перечисление Scala имеет несколько проблем:

1. Перечисления имеют один и тот же тип после стирания.
2. Компилятор не жалуется на то, что «Матч не является исчерпывающим», если он пропущен, он не будет работать во время выполнения `scala.MatchError` :

```
def isWeekendWithBug(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sun | WeekDays.Sat => true
}

isWeekendWithBug(WeekDays.Fri)
scala.MatchError: Fri (of class scala.Enumeration$Val)
```

Сравнить с:

```
def isWeekendWithBug(day: WeekDay): Boolean = day match {
  case WeekDay.Sun | WeekDay.Sat => true
}

Warning: match may not be exhaustive.
It would fail on the following inputs: Fri, Mon, Thu, Tue, Wed
def isWeekendWithBug(day: WeekDay): Boolean = day match {
                                              ^
```

Более подробное объяснение представлено в этой статье о Scala Enumeration .

## Examples

### Дни недели с использованием Scala Enumeration

Подобные Java-перечисления можно создать, расширив Enumeration .

```
object WeekDays extends Enumeration {
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}

def isWeekend(day: WeekDays.Value): Boolean = day match {
  case WeekDays.Sat | WeekDays.Sun => true
  case _ => false
}

isWeekend(WeekDays.Sun)
res0: Boolean = true
```

Также возможно добавить человеко-читаемое имя для значений в перечислении:

```
object WeekDays extends Enumeration {
      val Mon = Value("Monday")
      val Tue = Value("Tuesday")
      val Wed = Value("Wednesday")
      val Thu = Value("Thursday")
      val Fri = Value("Friday")
      val Sat = Value("Saturday")
      val Sun = Value("Sunday")
}

println(WeekDays.Mon)
>> Monday

WeekDays.withName("Monday") == WeekDays.Mon
>> res0: Boolean = true
```

Остерегайтесь поведения нестандартного типа, в котором различные перечисления могут оцениваться как один и тот же тип экземпляра:

```
object Parity extends Enumeration {
   val Even, Odd = Value
}

WeekDays.Mon.isInstanceOf[Parity.Value]
>> res1: Boolean = true
```

## Использование закрытых объектов и объектов

Альтернативой расширению `Enumeration` является использование `sealed` объектов:

```
sealed trait WeekDay

object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
}
```

`sealed` ключевое слово гарантирует, что черта `WeekDay` не может быть расширена в другом файле. Это позволяет компилятору сделать определенные предположения, в том числе, что все возможные значения `WeekDay` уже перечислены.

Один из недостатков заключается в том, что этот метод не позволяет вам получить список всех возможных значений. Чтобы получить такой список, он должен быть указан явно:

```
val allWeekDays = Seq(Mon, Tue, Wed, Thu, Fri, Sun, Sat)
```

Классы классов также могут распространять `sealed` черту. Таким образом, объекты и классы case могут быть смешаны для создания сложных иерархий:

```
sealed trait CelestialBody

object CelestialBody {
  case object Earth extends CelestialBody
  case object Sun extends CelestialBody
  case object Moon extends CelestialBody
  case class Asteroid(name: String) extends CelestialBody
}
```

Другой недостаток заключается в том, что нет способа получить доступ к имени переменной для перечисления `sealed` объекта или выполнить поиск по нему. Если вам нужно какое-то имя, связанное с каждым значением, оно должно быть определено вручную:

```
  sealed trait WeekDay { val name: String }

  object WeekDay {
      case object Mon extends WeekDay { val name = "Monday" }
      case object Tue extends WeekDay { val name = "Tuesday" }
      (...)
  }
```

Или просто:

```
  sealed case class WeekDay(name: String)

  object WeekDay {
      object Mon extends WeekDay("Monday")
      object Tue extends WeekDay("Tuesday")
      (...)
  }
```

## Использование закрытых объектов и объектов case и allValues-macro

Это просто расширение по запечатанному варианту, где макрос генерирует набор со всеми экземплярами во время компиляции. Это прекрасно исключает недостаток, который разработчик может добавить к перечислению, но забудьте добавить его в набор allElements.

Этот вариант особенно удобен для больших переписей.

```
import EnumerationMacros._

sealed trait WeekDay
object WeekDay {
  case object Mon extends WeekDay
  case object Tue extends WeekDay
  case object Wed extends WeekDay
  case object Thu extends WeekDay
```

```
  case object Fri extends WeekDay
  case object Sun extends WeekDay
  case object Sat extends WeekDay
  val allWeekDays: Set[WeekDay] = sealedInstancesOf[WeekDay]
}
```

Для этого вам понадобится этот макрос:

```
import scala.collection.immutable.TreeSet
import scala.language.experimental.macros
import scala.reflect.macros.blackbox

/**
A macro to produce a TreeSet of all instances of a sealed trait.
Based on Travis Brown's work:
http://stackoverflow.com/questions/13671734/iteration-over-a-sealed-trait-in-scala
CAREFUL: !!! MUST be used at END OF code block containing the instances !!!
*/
object EnumerationMacros {
  def sealedInstancesOf[A]: TreeSet[A] = macro sealedInstancesOf_impl[A]

  def sealedInstancesOf_impl[A: c.WeakTypeTag](c: blackbox.Context) = {
    import c.universe._

    val symbol = weakTypeOf[A].typeSymbol.asClass

    if  (!symbol.isClass || !symbol.isSealed)
      c.abort(c.enclosingPosition, "Can only enumerate values of a sealed trait or class.")
    else {

      val children = symbol.knownDirectSubclasses.toList

      if (!children.forall(_.isModuleClass)) c.abort(c.enclosingPosition, "All children must
be objects.")
      else c.Expr[TreeSet[A]] {

        def sourceModuleRef(sym: Symbol) =
Ident(sym.asInstanceOf[scala.reflect.internal.Symbols#Symbol
          ].sourceModule.asInstanceOf[Symbol]
        )

        Apply(
          Select(
            reify(TreeSet).tree,
            TermName("apply")
          ),
          children.map(sourceModuleRef(_))
        )
      }
    }
  }
}
```

Прочитайте Перечисления онлайн: https://riptutorial.com/ru/scala/topic/1499/перечисления

# глава 41: Пользовательские функции для улья

## Examples

### Простое UUF Hive в Apache Spark

```
import org.apache.spark.sql.functions._

// Create a function that uses the content of the column inside the dataframe
val code = (param: String) => if (param == "myCode") 1 else 0
// With that function, create the udf function
val myUDF = udf(code)
// Apply the udf to a column inside the existing dataframe, creating a dataframe with the
additional new column
val newDataframe = aDataframe.withColumn("new_column_name", myUDF(col(inputColumn)))
```

Прочитайте Пользовательские функции для улья онлайн:
https://riptutorial.com/ru/scala/topic/8241/пользовательские-функции-для-улья

# глава 42: Программирование на уровне

## Examples

### Введение в программирование на уровне

Если мы рассмотрим гетерогенный список, в котором элементы списка имеют разные, но известные типы, может быть желательно иметь возможность выполнять операции над элементами списка вместе, не отбрасывая информацию о типе элементов. Следующий пример реализует операцию отображения над простым гетерогенным списком.

Поскольку тип элемента изменяется, класс операций, которые мы можем выполнить, ограничен некоторой формой проекции типа, поэтому мы определяем характеристику `Projection` с абстрактным `type Apply[A]` вычисляя *тип* результата проекции, и `def apply[A](a: A): Apply[A]` вычисление *значения* результата проекции.

```
trait Projection {
  type Apply[A] // <: Any
  def apply[A](a: A): Apply[A]
}
```

При реализации `type Apply[A]` мы программируем на уровне типа (в отличие от уровня значения).

Наш гетерогенный тип списка определяет операцию `map` параметризованную нужной проекцией, а также тип проекции. Результат операции карты является абстрактным, будет варьироваться в зависимости от реализации класса и проекции и, естественно, должен быть `HList` :

```
sealed trait HList {
  type Map[P <: Projection] <: HList
  def map[P <: Projection](p: P): Map[P]
}
```

В случае `HNil` , пустой гетерогенного списка, результат любой проекции всегда будет сам. Здесь мы объявляем `trait HNil` в качестве удобства, чтобы мы могли писать `HNil` как тип вместо `HNil.type` :

```
sealed trait HNil extends HList
case object HNil extends HNil {
  type Map[P <: Projection] = HNil
  def map[P <: Projection](p: P): Map[P] = HNil
}
```

`HCons` - непустой гетерогенный список. Здесь мы утверждаем, что при применении операции с картами результирующий тип заголовка таков, что является результатом применения

проекции к начальному значению ( `P#Apply[H]` ) и что полученный хвостовой тип - это результат преобразования проекция над хвостом ( `T#Map[P]` ), который, как известно, является `HList` :

```
case class HCons[H, T <: HList](head: H, tail: T) extends HList {
  type Map[P <: Projection] = HCons[P#Apply[H], T#Map[P]]
  def map[P <: Projection](p: P): Map[P] = HCons(p.apply(head), tail.map(p))
}
```

Наиболее очевидной такой проекцией является выполнение какой-либо формы операции обертывания - следующий пример дает экземпляр `HCons[Option[String], HCons[Option[Int], HNil]]` :

```
HCons("1", HCons(2, HNil)).map(new Projection {
  type Apply[A] = Option[A]
  def apply[A](a: A): Apply[A] = Some(a)
})
```

Прочитайте Программирование на уровне онлайн: https://riptutorial.com/ru/scala/topic/3738/программирование-на-уровне

# глава 43: Работа с данными в неизменном стиле

## замечания

## Имена значений и переменных должны быть в нижнем верблюжьем корпусе

> Константные имена должны быть в верхнем верблюжьем корпусе. То есть, если член является окончательным, неизменным и принадлежит объекту пакета или объекту, его можно считать константой
>
> Метод, значения и имена переменных должны быть в нижнем верблюжьем корпусе

Источник: http://docs.scala-lang.org/style/naming-conventions.html

Этот компилятор:

```
val (a,b) = (1,2)
// a: Int = 1
// b: Int = 2
```

но это не так:

```
val (A,B) = (1,2)
// error: not found: value A
// error: not found: value B
```

## Examples

### Это не просто val vs. var

#### `val` И `var`

```
scala> val a = 123
a: Int = 123

scala> a = 456
<console>:8: error: reassignment to val
       a = 456

scala> var b = 123
b: Int = 123
```

```
scala> b = 321
b: Int = 321
```

- ссылки `val` неизменяемы: как `final` переменная в `Java` , после ее инициализации вы не можете ее изменить
- ссылки `var` переназначаются как объявление простой переменной в Java

## Неизменяемые и взаимозаменяемые коллекции

```
  val mut = scala.collection.mutable.Map.empty[String, Int]
  mut += ("123" -> 123)
  mut += ("456" -> 456)
  mut += ("789" -> 789)

  val imm = scala.collection.immutable.Map.empty[String, Int]
  imm + ("123" -> 123)
  imm + ("456" -> 456)
  imm + ("789" -> 789)

  scala> mut
    Map(123 -> 123, 456 -> 456, 789 -> 789)

  scala> imm
    Map()

scala> imm + ("123" -> 123) + ("456" -> 456) + ("789" -> 789)
    Map(123 -> 123, 456 -> 456, 789 -> 789)
```

Стандартная библиотека Scala предлагает как неизменяемые, так и изменяемые структуры данных, а не ссылку на нее. Каждый раз, когда неизменяемая структура данных получает «изменение», создается новый экземпляр вместо изменения первоначальной коллекции на месте. Каждый экземпляр коллекции может иметь значительную структуру с другим экземпляром.

Mutable and Immutable Collection (Официальная документация Scala)

## Но я не могу использовать неизменность в этом случае!

Давайте возьмем в качестве примера функцию, которая принимает 2 `Map` и возвращает `Map` содержащую каждый элемент в `ma` и `mb` :

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int]
```

Первая попытка может быть итерирована через элементы одного из карт, используя `for ((k, v) <- map)` и как-то вернуть объединенную карту.

```
def merge2Maps(ma: ..., mb: ...): Map[String, Int] = {

  for ((k, v) <- mb) {
```

```
    ???
  }

}
```

Это очень первый шаг немедленно добавить Ограничить: **мутацию вне что `for` теперь *требуется.*** Это более ясно , когда де-обсахаривания `for` :

```
// this:
for ((k, v) <- map) { ??? }

// is equivalent to:
map.foreach { case (k, v) => ??? }
```

### «Почему мы должны мутировать?»

`foreach` полагается на побочные эффекты. Каждый раз, когда мы хотим, чтобы что-то произошло в рамках `foreach` нам нужно «что-то побочное», в этом случае мы могли бы мутировать переменный `var result` или мы можем использовать изменяемую структуру данных.

### Создание и заполнение карты `result`

Предположим, что `ma` и `mb` являются `scala.collection.immutable.Map` , мы могли бы создать `result` Map from `ma` :

```
val result = mutable.Map() ++ ma
```

Затем итерации через `mb` добавляя его элементы, и если `key` текущего элемента на `ma` уже существует, давайте переопределим его с `mb` one.

```
mb.foreach { case (k, v) => result += (k -> v) }
```

### Мутируемая реализация

Пока что так хорошо, нам «пришлось использовать изменчивые коллекции», и правильная реализация могла бы быть:

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  val result = scala.collection.mutable.Map() ++ ma
  mb.foreach { case (k, v) => result += (k -> v) }
  result.toMap // to get back an immutable Map
}
```

Как и ожидалось:

```
scala> merge2Maps(Map("a" -> 11, "b" -> 12), Map("b" -> 22, "c" -> 23))
  Map(a -> 11, b -> 22, c -> 23)
```

### Складывание на помощь

Как мы можем избавиться от `foreach` в этом сценарии? Если все, что нам нужно делать, это в основном перебирать элементы коллекции и применять функцию, тогда как при накоплении результата на опции может использоваться `.foldLeft` :

```
def merge2Maps(ma: Map[String, Int], mb: Map[String, Int]): Map[String, Int] = {
  mb.foldLeft(ma) { case (result, (k, v)) => result + (k -> v) }
  // or more concisely mb.foldLeft(ma) { _ + _ }
}
```

В этом случае наш «результат» - это накопленное значение, начиная с `ma` , `zero` `.foldLeft` .

### Промежуточный результат

Очевидно, это непреложное решение создает и уничтожает многие экземпляры `Map` во время сворачивания, но стоит упомянуть, что эти экземпляры не являются полным клоном `Map` накопленной, но вместо этого разделяют значительную структуру (данные) с существующим экземпляром.

### Простая разумность

Легче рассуждать о семантике, если она более декларативная, как подход `.foldLeft` . Использование неизменяемых структур данных может помочь упростить нашу реализацию.

Прочитайте Работа с данными в неизменном стиле онлайн:
https://riptutorial.com/ru/scala/topic/6298/работа-с-данными-в-неизменном-стиле

# глава 44: Работа с люлькой

## Examples

### Основная настройка

1. Создайте файл с именем `SCALA_PROJECT/build.gradle` с этим содержимым:

```
group 'scala_gradle'
version '1.0-SNAPSHOT'

apply plugin: 'scala'

repositories {
    jcenter()
    mavenCentral()
    maven {
        url "https://repo.typesafe.com/typesafe/maven-releases"
    }
}

dependencies {
    compile group: 'org.scala-lang', name: 'scala-library', version: '2.10.6'
}


task  "create-dirs" << {
    sourceSets*.scala.srcDirs*.each { it.mkdirs() }
    sourceSets*.resources.srcDirs*.each { it.mkdirs() }
}
```

2. Запустите `gradle tasks` чтобы увидеть доступные задачи.

3. Запустите `gradle create-dirs` чтобы создать каталог `src/scala, src/resources` .

4. Запустите `gradle build` для создания зависимостей проекта и загрузки.

### Создайте свой собственный плагин Gradle Scala

Пройдя пример **Basic Setup** , вы можете повторить большую часть его в каждом проекте Scala Gradle. Пахнет шаблоном кода ...

Что делать, если вместо применения плагина Scala, предлагаемого Gradle, вы можете применить свой собственный плагин Scala, который будет отвечать за обработку всей вашей общей логики сборки, одновременно расширяя уже существующий плагин.

Этот пример будет преобразовывать предыдущую логику сборки в многоразовый плагин Gradle.

К счастью, в Gradle вы можете легко создавать пользовательские плагины с помощью Gradle API, как указано в документации . В качестве языка реализации вы можете использовать Scala самостоятельно или даже Java. Однако большинство примеров, которые вы можете найти в документах, написаны в Groovy. Если вам нужно больше образцов кода или вы хотите понять, что находится за плагином Scala, например, вы можете проверить рельеф github gradle.

# Написание плагина

### Требования

Пользовательский плагин добавит следующие функции при применении к проекту:

- `scalaVersion` свойства `scalaVersion` , который будет иметь два переопределяемых свойства по умолчанию
  - major = "2.12"
  - minor = "0"
- функция `withScalaVersion` , которая применяется к имени зависимостей, добавит основную версию scala для обеспечения совместимости с двоичными файлами (оператор sbt `%%` может позвонить на звонок, в противном случае перейдите сюда, прежде чем продолжить)
- задание `createDirs` для создания необходимого дерева каталогов, как и в предыдущем примере

### Руководство по внедрению

1. создайте новый проект градиента и добавьте следующее для `build.gradle`

```
apply plugin: 'scala'
apply plugin: 'maven'

repositories {
    mavenLocal()
    mavenCentral()
}

dependencies {
    compile gradleApi()
    compile "org.scala-lang:scala-library:2.12.0"
}
```

**Примечания** :

- реализация плагина написана в Scala, поэтому нам нужен плагин Scala Gradle Scala
- для использования плагина из других проектов используется плагин Gradle Maven; это добавляет задачу `install` используемую для сохранения контейнера проекта в локальный репозиторий Maven

- `compile gradleApi()` добавляет `gradle-api-<gradle_version>.jar` в `gradle-api-<gradle_version>.jar` к классам

2. создать новый класс Scala для реализации плагина

```
package com.btesila.gradle.plugins

import org.gradle.api.{Plugin, Project}

class ScalaCustomPlugin extends Plugin[Project] {
    override def apply(project: Project): Unit = {
        project.getPlugins.apply("scala")
    }
}
```

**Примечания** :

- чтобы реализовать плагин, просто расширьте свойство `Plugin` типа `Project` и переопределите метод `apply`
- в рамках метода apply у вас есть доступ к экземпляру `Project` , к которому применяется плагин, и вы можете использовать его для добавления к нему логики сборки
- этот плагин не делает ничего, кроме применения уже существующего плагина Gradle Scala

3. добавить `scalaVersion` объекта `scalaVersion`

Во-первых, мы создаем класс `ScalaVersion` , который будет содержать два свойства версии

```
class ScalaVersion {
  var major: String = "2.12"
  var minor: String = "0"
}
```

Одной из замечательных особенностей плагинов Gradle является то, что вы всегда можете добавлять или переопределять определенные свойства. Плагин получает этот пользовательский ввод через `ExtensionContainer` прикрепленный к экземпляру `Project` gradle. Для получения дополнительной информации проверьте это .
Добавив следующее к методу `apply` , мы в основном делаем это:

- если в `scalaVersion` определено свойство `scalaVersion` , мы добавляем один со значениями по умолчанию
- в противном случае мы получаем существующий экземпляр `ScalaVersion` , чтобы использовать его дальше

```
var scalaVersion = new ScalaVersion
if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
   project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
```

```
else
   scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]
```

Это эквивалентно написанию следующего файла сборки проекта, который применяет плагин:

```
ext {
    scalaVersion.major = "2.12"
    scalaVersion.minor = "0"

}
```

4. добавьте библиотеку `scala-lang` в зависимости от проекта, используя `scalaVersion`

```
project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")
```

Это эквивалентно написанию следующего файла сборки проекта, который применяет плагин:

```
compile "org.scala-lang:scala-library:2.12.0"
```

5. добавьте функцию `withScalaVersion`

```
val withScalaVersion = (lib: String) => {
    val libComp = lib.split(":")
    libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
    libComp.mkString(":")
}
project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)
```

6. наконец, создать задачу `createDirs` и добавить его в проект
   Реализовать задачу Gradle, расширив `DefaultTask` :

```
class CreateDirs extends DefaultTask {
  @TaskAction
  def createDirs(): Unit = {
    val sourceSetContainer =
this.getProject.getConvention.getPlugin(classOf[JavaPluginConvention]).getSourceSets

    sourceSetContainer forEach { sourceSet =>
      sourceSet.getAllSource.getSrcDirs.forEach(file => if (!file.getName.contains("java"))
file.mkdirs())
    }
  }
}
```

**Примечание** . `SourceSetContainer` имеет информацию обо всех исходных каталогах, присутствующих в проекте. То, что делает плагин Gradle Scala, заключается в добавлении

дополнительных наборов источников в Java, как вы можете видеть в документах плагина .

Добавьте задачу `createDir` в проект, добавив это к методу `apply` :

```
project.getTasks.create("createDirs", classOf[CreateDirs])
```

В конце концов, ваш класс `ScalaCustomPlugin` должен выглядеть следующим образом:

```
class ScalaCustomPlugin extends Plugin[Project] {
  override def apply(project: Project): Unit = {
    project.getPlugins.apply("scala")

    var scalaVersion = new ScalaVersion
    if (!project.getExtensions.getExtraProperties.has("scalaVersion"))
      project.getExtensions.getExtraProperties.set("scalaVersion", scalaVersion)
    else
      scalaVersion =
project.getExtensions.getExtraProperties.get("scalaVersion").asInstanceOf[ScalaVersion]

    project.getDependencies.add("compile", s"org.scala-lang:scala-
library:${scalaVersion.major}.${scalaVersion.minor}")

    val withScalaVersion = (lib: String) => {
      val libComp = lib.split(":")
      libComp.update(1, s"${libComp(1)}_${scalaVersion.major}")
      libComp.mkString(":")
    }
    project.getExtensions.getExtraProperties.set("withScalaVersion", withScalaVersion)

    project.getTasks.create("createDirs", classOf[CreateDirs])
  }
}
```

### Установка проекта плагина в локальный репозиторий Maven

Это делается очень просто, выполнив `gradle install`
Вы можете проверить установку, перейдя в каталог локального репозитория, обычно находящийся в `~/.m2/repository`

### Как Gradle находит наш новый плагин?

Каждый плагин Gradle имеет `id` который используется в заявлении `apply` . Например, написав следующий файл сборки, он переводит на триггер для Gradle, чтобы найти и применить плагин с id `scala` .

```
apply plugin: 'scala'
```

Точно так же мы хотели бы применить наш новый плагин следующим образом,

```
apply plugin: "com.btesila.scala.plugin"
```

что наш плагин будет иметь идентификатор `com.btesila.scala.plugin` .

Чтобы установить этот идентификатор, добавьте следующий файл:
**SRC / основные** / **ресурсы** / **META-INF / Gradle-плагин** / **com.btesil.scala.plugin.properties**

```
implementation-class=com.btesila.gradle.plugins.ScalaCustomPlugin
```

После этого снова `gradle install` .

# Использование плагина

1. создайте новый пустой проект Gradle и добавьте в файл сборки следующее:

```
buildscript {
    repositories {
        mavenLocal()
        mavenCentral()
    }

    dependencies {
        //modify this path to match the installed plugin project in your local repository
        classpath 'com.btesila:working-with-gradle:1.0-SNAPSHOT'
    }
}

repositories {
    mavenLocal()
    mavenCentral()
}


apply plugin: "com.btesila.scala.plugin"
```

2. run `gradle createDirs` - теперь вы должны создать все исходные каталоги
3. переопределите версию scala, добавив ее в файл сборки:

```
ext {
    scalaVersion.major = "2.11"
    scalaVersion.minor = "8"

}
println(project.ext.scalaVersion.major)
println(project.ext.scalaVersion.minor)
```

4. добавьте библиотеку зависимостей, которая является бинарной, совместимой с версией Scala

```
dependencies {
    compile withScalaVersion("com.typesafe.scala-logging:scala-logging:3.5.0")
}
```

Это оно! Теперь вы можете использовать этот плагин для всех своих проектов, не повторяя тот же старый шаблон.

Прочитайте Работа с люлькой онлайн: https://riptutorial.com/ru/scala/topic/3304/работа-с-люлькой

# глава 45: Разница типов

## Examples

### ковариации

Символ `+` отмечает параметр типа как *ковариантный* - здесь мы говорим, что « `Producer` ковариантен на `A` »:

```
trait Producer[+A] {
  def produce: A
}
```

Параметр ковариантного типа можно рассматривать как «выходной» тип. Маркировка `A` как ковариантная утверждает, что `Producer[X] <: Producer[Y]` при условии, что `X <: Y` Например, `Producer[Cat]` является действительным `Producer[Animal]` , так как все произведенные кошки также являются действительными животными.

Параметр ковариантного типа не может отображаться в контравариантном (входном) положении. Следующий пример не будет компилироваться, поскольку мы утверждаем, что `Co[Cat] <: Co[Animal]` , но `Co[Cat]` имеет `def handle(a: Cat): Unit` который не может обрабатывать любое `Animal` по требованию `Co[Animal]` !

```
trait Co[+A] {
  def produce: A
  def handle(a: A): Unit
}
```

Один из подходов к решению этого ограничения заключается в использовании параметров типа, ограниченных параметром ковариантного типа. В следующем примере мы знаем, что `B` является супертипом `A` Поэтому данный параметр `Option[X] <: Option[Y]` для `X <: Y` , мы знаем, что `Option[X] def getOrElse[B >: X](b: => B): B` может принимать любой супертип `X` - который включает в себя супертипы `Y` как требуется `Option[Y]` :

```
trait Option[+A] {
  def getOrElse[B >: A](b: => B): B
}
```

### неизменность

По умолчанию все параметры типа инвариантны - данный `trait A[B]` , мы говорим, что « `A` инвариантно на `B` ». Это означает, что, учитывая две параметризации `A[Cat]` и `A[Animal]` , мы не утверждаем никакого отношения sub / superclass между этими двумя типами - он не считает, что `A[Cat] <: A[Animal]` и `A[Cat] >: A[Animal]` независимо от отношения между `Cat` и `Animal`

.

Аннотации вариации предоставляют нам способ объявления таких отношений и налагают правила использования параметров типа, чтобы отношения оставались действительными.

### контрвариация

Символ `-` обозначает параметр типа как *контравариантный* - здесь мы говорим, что « `Handler` контравариантен на `A` »:

```
trait Handler[-A] {
  def handle(a: A): Unit
}
```

Параметр контравариантного типа можно рассматривать как «входной» тип. Маркировка `A` как контравариантная утверждает, что `Handler[X] <: Handler[Y]` при условии, что `X >: Y` Например, `Handler[Animal]` является допустимым `Handler[Cat]` , поскольку `Handler[Animal]` также должен обрабатывать кошек.

Параметр контравариантного типа не может появляться в ковариантной (выходной) позиции. Следующий пример не будет компилироваться, поскольку мы утверждаем, что `Contra[Animal] <: Contra[Cat]` , однако `Contra[Animal]` имеет `def produce: Animal` которому не гарантируется создание кошек по требованию `Contra[Cat]` !

```
trait Contra[-A] {
   def handle(a: A): Unit
   def produce: A
}
```

Однако опасайтесь: для целей перегрузки разрешения контравариантность также интуитивно инвертирует специфичность типа на контравариантном параметре типа. `Handler[Animal]` считается «более конкретным», чем `Handler[Cat]` .

Поскольку невозможно перегрузить методы для параметров типа, это поведение обычно становится проблематичным при разрешении неявных аргументов. В следующем `ofCat` никогда не будет использоваться, поскольку тип возвращаемого значения `ofAnimal` более специфичен:

```
implicit def ofAnimal: Handler[Animal] = ???
implicit def ofCat: Handler[Cat] = ???

implicitly[Handler[Cat]].handle(new Cat)
```

В настоящее время это поведение изменено в точках , и именно поэтому (в качестве примера) `scala.math.Ordering` является инвариантным по его параметру типа `T` Один из способов - сделать ваш инвариант typeclass и ввести параметризацию неявного определения в случае, если вы хотите, чтобы оно применимо к подклассам заданного типа:

```
trait Person
object Person {
  implicit def ordering[A <: Person]: Ordering[A] = ???
}
```

## Ковариация коллекции

Поскольку коллекции, как правило, ковариантны по типу элемента *, может быть передан набор подтипов, где ожидается супер-тип:

```
trait Animal { def name: String }
case class Dog(name: String) extends Animal

object Animal {
  def printAnimalNames(animals: Seq[Animal]) = {
    animals.foreach(animal => println(animal.name))
  }
}

val myDogs: Seq[Dog] = Seq(Dog("Curly"), Dog("Larry"), Dog("Moe"))

Animal.printAnimalNames(myDogs)
// Curly
// Larry
// Moe
```

Это может показаться не волшебством, но тот факт, что `Seq[Dog]` принят методом, который ожидает `Seq[Animal]` - это все понятие более высокого типа (здесь: `Seq` ), являющееся ковариантным в своем параметре типа.

* Контрпример, являющийся набором стандартной библиотеки

## Ковариация по инвариантной черты

Существует также способ, чтобы один метод принимал ковариантный аргумент, вместо того чтобы иметь ковариант целиком. Это может быть необходимо, потому что вы хотели бы использовать `T` в контравариантном положении, но все равно иметь ковариантность.

```
trait LocalVariance[T]{
  /// ??? throws a NotImplementedError
  def produce: T = ???
  // the implicit evidence provided by the compiler confirms that S is a
  // subtype of T.
  def handle[S](s: S)(implicit evidence: S <:< T) = {
    // and we can use the evidence to convert s into t.
    val t: T = evidence(s)
    ???
  }
}

trait A {}
trait B extends A {}
```

```
object Test {
  val lv = new LocalVariance[A] {}

  // now we can pass a B instead of an A.
  lv.handle(new B {})
}
```

Прочитайте Разница типов онлайн: https://riptutorial.com/ru/scala/topic/1651/разница-типов

# глава 46: Регулярные выражения

## Синтаксис

- re.findAllIn (s: CharSequence): MatchIterator
- re.findAllMatchIn (s: CharSequence): Итератор [Матч]
- re.findFirstIn (s: CharSequence): Option [String]
- re.findFirstMatchIn (s: CharSequence): Option [Match]
- re.findPrefixMatchIn (s: CharSequence): Option [Match]
- re.findPrefixOf (s: CharSequence): Option [String]
- re.replaceAllIn (s: CharSequence, replacer: Match => String): String
- re.replaceAllIn (s: CharSequence, replacement: String): String
- re.replaceFirstIn (s: CharSequence, replacement: String): String
- re.replaceSomeIn (s: CharSequence, replacer: Match => Option [String]): String
- re.split (s: CharSequence): Array [String]

## Examples

### Объявление регулярных выражений

Метод `r` неявно предоставляется через scala.collection.immutable.StringOps создает экземпляр scala.util.matching.Regex из строки темы. Синтаксис синтаксиса Scala с тремя кавычками полезен здесь, так как вам не нужно скрывать обратную косую черту, как в Java:

```
val r0: Regex = """(\d{4})-(\d${2})-(\d{2})""".r     // :)
val r1: Regex = "(\\d{4})-(\\d{2})-(\\d{2})".r // :(
```

scala.util.matching.Regex реализует идиоматическое регулярное выражение API для Scala как обертки над java.util.regex.Pattern , а поддерживаемый синтаксис тот же. При этом поддержка Scala для многострочных строковых литералов делает флаг `x` существенно более полезным, позволяя комментировать и игнорируя пробел шаблона:

```
val dateRegex = """(?x:
   (\d{4}) # year
  -(\d{2}) # month
  -(\d{2}) # day
)""".r
```

Существует перегруженная версия `r` , `def r(names: String*): Regex` которая позволяет вам назначать имена групп для захвата вашего шаблона. Это несколько хрупко, поскольку имена отделяются от захватов и должны использоваться только в том случае, если регулярное выражение будет использоваться в нескольких местах:

```
"""(\d{4})-(\d{2})-(\d{2})""".r("y", "m", "d").findFirstMatchIn(str) match {
  case Some(matched) =>
    val y = matched.group("y").toInt
    val m = matched.group("m").toInt
    val d = matched.group("d").toInt
    java.time.LocalDate.of(y, m, d)
  case None => ???
}
```

## Повторное сопоставление шаблона в строке

```
val re = """\((.*?)\)""".r

val str =
"(The)(example)(of)(repeating)(pattern)(in)(a)(single)(string)(I)(had)(some)(trouble)(with)(once)"


re.findAllMatchIn(str).map(_.group(1)).toList
res2: List[String] = List(The, example, of, repeating, pattern, in, a, single, string, I, had,
some, trouble, with, once)
```

Прочитайте Регулярные выражения онлайн: https://riptutorial.com/ru/scala/topic/2891/регулярные-выражения

# глава 47: Рекурсия

## Examples

### Рекурсия хвоста

Используя регулярную рекурсию, каждый рекурсивный вызов подталкивает другую запись в стек вызовов. Когда рекурсия завершена, приложение должно вытолкнуть каждую запись полностью назад. Если есть много рекурсивных вызовов функций, это может закончиться огромным стеком.

Scala автоматически удаляет рекурсию, если находит рекурсивный вызов в позиции хвоста. Аннотацию ( `@tailrec` ) можно добавить к рекурсивным функциям, чтобы гарантировать, что выполняется оптимизация хвостового вызова. Затем компилятор показывает сообщение об ошибке, если он не может оптимизировать рекурсию.

## Регулярная рекурсия

Этот пример не является хвостовым рекурсивным, так как при рекурсивном вызове функция должна отслеживать умножение, которое необходимо выполнить с результатом после возврата вызова.

```
def fact(i : Int) : Int = {
   if(i <= 1) i
   else i * fact(i-1)
}

println(fact(5))
```

Вызов функции с параметром приведет к тому, что стек выглядит следующим образом:

```
(fact 5)
(* 5 (fact 4))
(* 5 (* 4 (fact 3)))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (* 2 (fact 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 * 1)))))
(* 5 (* 4 (* 3 (* 2))))
(* 5 (* 4 (* 6)))
(* 5 (* 24))
120
```

Если мы попытаемся аннотировать этот пример с помощью `@tailrec` мы получим следующее сообщение об ошибке: `could not optimize @tailrec annotated method fact: it contains a recursive call not in tail position`

# Рекурсия хвоста

В хвостовой рекурсии вы сначала выполняете свои вычисления, а затем выполняете рекурсивный вызов, передавая результаты вашего текущего шага на следующий рекурсивный шаг.

```
def fact_with_tailrec(i : Int) : Long = {
   @tailrec
   def fact_inside(i : Int, sum: Long) : Long = {
      if(i <= 1) sum
      else fact_inside(i-1,sum*i)
   }
   fact_inside(i,1)
}

println(fact_with_tailrec(5))
```

Напротив, трассировка стека для хвостового рекурсивного факториала выглядит следующим образом:

```
(fact_with_tailrec 5)
(fact_inside 5 1)
(fact_inside 4 5)
(fact_inside 3 20)
(fact_inside 2 60)
(fact_inside 1 120)
```

Существует только необходимость отслеживать один и тот же объем данных для каждого вызова функции `fact_inside` потому что функция просто возвращает значение, которое она получила до самого верха. Это означает, что даже если `fact_with_tail 1000000` вызывается, ему требуется только то же пространство, что и `fact_with_tail 3` . Это не относится к не-хвостовому рекурсивному вызову, и поскольку такие большие значения могут вызвать переполнение стека.

## Бесступенчатая рекурсия с батутом (scala.util.control.TailCalls)

Очень часто возникает ошибка `StackOverflowError` при вызове рекурсивной функции. Стандартная библиотека Scala предлагает TailCall, чтобы избежать переполнения стека, используя объекты кучи и продолжения для сохранения локального состояния рекурсии.

Два примера из скаладока TailCalls

```
import scala.util.control.TailCalls._

def isEven(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(true) else tailcall(isOdd(xs.tail))

def isOdd(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(false) else tailcall(isEven(xs.tail))
```

```
// Does this List contain an even number of elements?
isEven((1 to 100000).toList).result

def fib(n: Int): TailRec[Int] =
  if (n < 2) done(n) else for {
    x <- tailcall(fib(n - 1))
    y <- tailcall(fib(n - 2))
  } yield (x + y)

// What is the 40th entry of the Fibonacci series?
fib(40).result
```

Прочитайте Рекурсия онлайн: https://riptutorial.com/ru/scala/topic/3889/рекурсия

# глава 48: Самостоятельные типы

## Синтаксис

- trait Type {selfId => / *другие члены могут ссылаться на* `selfId` *если* `this` *означает что-то* /}
- trait Type {selfId: OtherType => / * другие члены могут использовать `selfId` и будут иметь тип `OtherType` * /
- trait Type {selfId: OtherType1 with OtherType2 => / * `selfId` имеет тип `OtherType1` и `OtherType2` * /

## замечания

Часто используется с рисунком пирога.

## Examples

### Пример простого типа

Типы типов могут использоваться в свойствах и классах для определения ограничений на конкретные классы, к которым он смешивается. Также можно использовать другой идентификатор для `this` используя этот синтаксис (полезно, когда внешний объект должен ссылаться от внутреннего объекта).

Предположим, вы хотите сохранить некоторые объекты. Для этого вы создаете интерфейсы для хранилища и добавляете значения в контейнер:

```
trait Container[+T] {
  def add(o: T): Unit
}

trait PermanentStorage[T] {
  /* Constraint on self type: it should be Container
   * we can refer to that type as `identifier`, usually `this` or `self`
   * or the type's name is used. */
  identifier: Container[T] =>

  def save(o: T): Unit = {
    identifier.add(o)
    //Do something to persist too.
  }
}
```

Таким образом, они не находятся в одной иерархии объектов, но `PermanentStorage` не может быть реализован без реализации `Container` .

Прочитайте Самостоятельные типы онлайн: https://riptutorial.com/ru/scala/topic/4639/

самостоятельные-типы

# глава 49: Символьные литералы

## замечания

Scala поставляется с концепцией **символов** - строки, которые *интернированы* , то есть: два символа с тем же именем (одна и та же последовательность символов), в отличие от строк, будут ссылаться на один и тот же объект во время выполнения.

Символы - это особенность многих языков: Lisp, Ruby и Erlang и многое другое, однако в Scala они относительно невелики. Хорошая особенность, тем не менее.

**Использование:**

Любое буквальное начало с одной цитатой ' , за которым следуют одна или несколько цифр, букв или под-баллов _ является символьным литералом. Первый символ является исключением, поскольку он не может быть цифрой.

Хорошие определения:

```
'ATM
'IPv4
'IPv6
'map_to_operations
'data_format_2006

// Using the `Symbol.apply` method

Symbol("hakuna matata")
Symbol("To be or not to be that is a question")
```

Плохие определения:

```
'8'th_division
'94_pattern
'bad-format
```

## Examples

### Замена строк в случаях

Предположим, у нас есть несколько источников *данных,* которые включают в себя *базу данных, файл, подсказку* и *список аргументов* . В зависимости от выбранного источника мы меняем наш подход:

```
def loadData(dataSource: Symbol): Try[String] = dataSource match {
  case 'database => loadDatabase() // Loading data from database
```

```
  case 'file =>  loadFile() // Loading data from file
  case 'prompt => askUser() // Asking user for data
  case 'argumentList => argumentListExtract() // Accessing argument list for data
  case _ => Failure(new Exception("Unsupported data source"))
}
```

Мы могли бы очень хорошо использовать `String` вместо `Symbol` . Мы этого не сделали, потому что в этом контексте нет ни одной функции строк.

Это делает код более простым и менее подверженным ошибкам.

Прочитайте Символьные литералы онлайн: https://riptutorial.com/ru/scala/topic/6419/символьные-литералы

# глава 50: синхронизированный

## Синтаксис

- `objectToSynchronizeOn.synchronized { /* code to run */}`
- `synchronized {/* code to run, can be suspended with wait */}`

## Examples

### синхронизировать объект

`synchronized` - это низкоуровневая конструкция параллелизма, которая может помочь предотвратить доступ нескольких потоков к тем же ресурсам. Введение для JVM с использованием языка Java .

```
anInstance.synchronized {
  // code to run when the intristic lock on `anInstance` is acquired
  // other thread cannot enter concurrently unless `wait` is called on `anInstance` to suspend
  // other threads can continue of the execution of this thread if they `notify` or
`notifyAll` `anInstance`'s lock
}
```

В случае `object` s он может синхронизироваться с классом объекта, а не с экземпляром singleton.

### синхронизировать неявно с этим

```
/* within a class, def, trait or object, but not a constructor */
synchronized {
  /* code to run when an intrisctic lock on `this` is acquired */
  /* no other thread can get the this lock unless execution is suspended with
   * `wait` on `this`
   */
}
```

Прочитайте синхронизированный онлайн: https://riptutorial.com/ru/scala/topic/3371/синхронизированный

# глава 51: Совместимость Java

## Examples

### Преобразование коллекций Scala в коллекции Java и наоборот

Когда вам нужно передать коллекцию в Java-метод:

```
import scala.collection.JavaConverters._

val scalaList = List(1, 2, 3)
JavaLibrary.process(scalaList.asJava)
```

Если Java-код возвращает коллекцию Java, вы можете превратить ее в коллекцию Scala аналогичным образом:

```
import scala.collection.JavaConverters._

val javaCollection = JavaLibrary.getList
val scalaCollection = javaCollection.asScala
```

Обратите внимание, что это декораторы, поэтому они просто обертывают базовые коллекции в интерфейсе коллекции Scala или Java. Поэтому вызовы `.asJava` и `.asScala` не копируют коллекции.

### Массивы

Массивы являются регулярными массивами JVM с завихрением, что они рассматриваются как инвариантные и имеют специальные конструкторы и неявные преобразования. Постройте их без `new` ключевого слова.

```
val a = Array("element")
```

Теперь `a` имеет тип `Array[String]` .

```
val acs: Array[CharSequence] = a
//Error: type mismatch;  found   : Array[String]  required: Array[CharSequence]
```

Хотя `String` конвертируется в `CharSequence` , `Array[String]` не конвертируется в `Array[CharSequence]` .

Вы можете использовать `Array` как и другие коллекции, благодаря неявному преобразованию в `TraversableLike` `ArrayOps` :

```
val b: Array[Int] = a.map(_.length)
```

Большинство коллекций Scala ( `TraversableOnce` ) имеют метод `toArray` который использует неявный `ClassTag` для построения массива результатов:

```
List(0).toArray
//> res1: Array[Int] = Array(0)
```

Это упрощает использование любого `TraversableOnce` в вашем коде Scala, а затем передает его в Java-код, который ожидает массив.

## Преобразования типа Scala и Java

Scala предлагает неявные преобразования между всеми основными типами коллекций в объекте JavaConverters.

Преобразования следующего типа являются двунаправленными.

| Тип Scala | Тип Java |
|---|---|
| Итератор | java.util.Iterator |
| Итератор | java.util.Enumeration |
| Итератор | java.util.Iterable |
| Итератор | java.util.Collection |
| mutable.Buffer | java.util.List |
| mutable.Set | java.util.Set |
| mutable.Map | java.util.Map |
| mutable.ConcurrentMap | java.util.concurrent.ConcurrentMap |

Некоторые другие коллекции Scala также могут быть преобразованы в Java, но не имеют преобразования обратно к исходному типу Scala:

| Тип Scala | Тип Java |
|---|---|
| Seq | java.util.List |
| mutable.Seq | java.util.List |
| Задавать | java.util.Set |
| карта | java.util.Map |

*Ссылка* :

Конверсии между Java и коллекциями Scala

## Функциональные интерфейсы для функций Scala - scala-java8-compat

Комплект совместимости Java 8 для Scala.

Большинство примеров копируются из Readme

### Преобразователи между scala.FunctionN и java.util.function

```
import java.util.function._
import scala.compat.java8.FunctionConverters._

val foo: Int => Boolean = i => i > 7
def testBig(ip: IntPredicate) = ip.test(9)
println(testBig(foo.asJava))  // Prints true

val bar = new UnaryOperator[String]{ def apply(s: String) = s.reverse }
List("cod", "herring").map(bar.asScala)    // List("doc", "gnirrih")

def testA[A](p: Predicate[A])(a: A) = p.test(a)
println(testA(asJavaPredicate(foo))(4))  // Prints false
```

### Преобразователи между классами scala.Option и java.util Необязательный, НеобязательныйДвойный, НеобязательныйInt и НеобязательныйЛог.

```
import scala.compat.java8.OptionConverters._

    class Test {
      val o = Option(2.7)
      val oj = o.asJava        // Optional[Double]
      val ojd = o.asPrimitive  // OptionalDouble
      val ojds = ojd.asScala   // Option(2.7) again
    }
```

### Преобразователи из коллекций Scala в потоки Java 8

```
import java.util.stream.IntStream

import scala.compat.java8.StreamConverters._
import scala.compat.java8.collectionImpl.{Accumulator, LongAccumulator}


  val m = collection.immutable.HashMap("fish" -> 2, "bird" -> 4)
  val parStream: IntStream = m.parValueStream
  val s: Int = parStream.sum
  // 6, potientially computed in parallel
  val t: List[String] = m.seqKeyStream.toScala[List]
  // List("fish", "bird")
  val a: Accumulator[(String, Int)] = m.accumulate // Accumulator[(String, Int)]

  val n = a.stepper.fold(0)(_ + _._1.length) +
```

```
  a.parStream.count // 8 + 2 = 10

val b: LongAccumulator = java.util.Arrays.stream(Array(2L, 3L, 4L)).accumulate
// LongAccumulator
val l: List[Long] = b.to[List] // List(2L, 3L, 4L)
```

Прочитайте Совместимость Java онлайн: https://riptutorial.com/ru/scala/topic/2441/совместимость-java

# глава 52: Соответствие шаблону

## Синтаксис

- селекторный матч partialFunction
- селекторное совпадение {список альтернатив случая) // Это наиболее распространенная форма вышеперечисленного

## параметры

| параметр | подробности |
|---|---|
| селектор | Выражение, значение которого сопоставляется с образцом. |
| альтернативы | список альтернатив, ограниченных `case` . |

## Examples

### Простой шаблонный матч

В этом примере показано, как сопоставить ввод с несколькими значениями:

```
def f(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
  case _ => "Unknown!"
}

f(2)  // "Two"
f(3)  // "Unknown!"
```

Демо-версия

Примечание: _ является случаем *падения* или по *умолчанию* , но это не требуется.

```
def g(x: Int): String = x match {
  case 1 => "One"
  case 2 => "Two"
}

g(1)  // "One"
g(3)  // throws a MatchError
```

Чтобы избежать исключения, это лучшая практика функционального программирования здесь для обработки случая по умолчанию ( `case _ => <do something>` ). Обратите внимание,

что сопоставление по *классу case* может помочь компилятору выдать предупреждение, если отсутствует случай. То же самое относится к пользовательским типам, которые расширяют запечатанный признак. Если совпадение является общим, то случай по умолчанию может не понадобиться

Также можно сопоставлять значения, не определенные в строке. Они должны быть *стабильными идентификаторами* , которые получаются либо с использованием имени с заглавной буквыми, либо с использованием обратных ссылок.

С `One` и `two` определенными где-то еще или переданными как параметры функции:

```
val One: Int = 1
val two: Int = 2
```

Их можно сопоставить следующим образом:

```
def g(x: Int): String = x match {
  case One => "One"
  case `two` => "Two"
}
```

В отличие от других языков программирования, поскольку Java, например, не проходит. Если блок-блок соответствует входу, он выполняется, и совпадение завершено. Поэтому наименее конкретный случай должен быть последним блоком случая.

```
def f(x: Int): String = x match {
  case _ => "Default"
  case 1 => "One"
}

f(5) // "Default"
f(1) // "Default"
```

## Сравнение шаблонов со стабильным идентификатором

При стандартном сопоставлении шаблонов используемый идентификатор будет затенять любой идентификатор в охватывающей области. Иногда необходимо сопоставлять переменную охватывающей области.

Следующая примерная функция принимает символ и список кортежей и возвращает новый список кортежей. Если символ существовал как первый элемент в одном из кортежей, второй элемент увеличивается. Если он еще не существует в списке, создается новый кортеж.

```
def tabulate(char: Char, tab: List[(Char, Int)]): List[(Char, Int)] = tab match {
  case Nil => List((char, 1))
  case (`char`, count) :: tail => (char, count + 1) :: tail
  case head :: tail => head :: tabulate(char, tail)
}
```

Вышеприведенное демонстрирует соответствие шаблону, в котором вход метода, `char` , поддерживается «стабильным» в совпадении шаблонов: то есть, если вы вызываете `tabulate('x', ...)` , первый оператор case будет интерпретироваться как:

```
case('x', count) => ...
```

Scala интерпретирует любую переменную, демаркированную с отметкой галочки как стабильный идентификатор: она также будет интерпретировать любую переменную, начинающуюся с заглавной буквы таким же образом.

## Согласование шаблонов на Seq

Чтобы проверить точное количество элементов в коллекции

```
def f(ints: Seq[Int]): String = ints match {
  case Seq() =>
      "The Seq is empty !"
  case Seq(first) =>
      s"The seq has exactly one element : $first"
  case Seq(first, second) =>
      s"The seq has exactly two elements : $first, $second"
  case  s @ Seq(_, _, _) =>
      s"s is a Seq of length three and looks like ${s}"  // Note individual elements are not
bound to their own names.
  case s: Seq[Int] if s.length == 4 =>
      s"s is a Seq of Ints of exactly length 4"  // Again, individual elements are not bound
to their own names.
  case _ =>
      "No match was found!"
}
```

### Демо-версия

Для извлечения первого элемента (ов) и сохранения остатка в виде коллекции:

```
def f(ints: Seq[Int]): String = ints match {
  case Seq(first, second, tail @ _*) =>
      s"The seq has at least two elements : $first, $second. The rest of the Seq is $tail"
  case Seq(first, tail @ _*) =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  // alternative syntax
  // here of course this one will never match since it checks
  // for the same thing as the one above
  case first +: tail =>
      s"The seq has at least one element : $first. The rest of the Seq is $tail"
  case _ =>
      "The seq didn't match any of the above, so it must be empty"
}
```

В общем, любая форма, которая может быть использована для построения последовательности, может использоваться для сопоставления шаблонов с существующей последовательностью.

Обратите внимание: при использовании `Nil` и `::` будет работать, если шаблон соответствует последовательности, он преобразует его в `List` и может иметь неожиданные результаты. Ограничьте себя `Seq( ...)` и `+:` чтобы избежать этого.

Обратите внимание, что при использовании `::` не будет работать для `WrappedArray` , `Vector` т. Д., См.

```
scala> def f(ints:Seq[Int]) = ints match {
     | case h :: t => h
     | case _ => "No match"
     | }
f: (ints: Seq[Int])Any

scala> f(Array(1,2))
res0: Any = No match
```

И с `+:`

```
scala> def g(ints:Seq[Int]) = ints match {
     | case h+:t => h
     | case _ => "No match"
     | }
g: (ints: Seq[Int])Any

scala> g(Array(1,2).toSeq)
res4: Any = 1
```

## Охранники (если выражения)

Операторы case можно комбинировать с выражениями, чтобы обеспечить дополнительную логику при сопоставлении шаблонов.

```
def checkSign(x: Int): String = {
    x match {
      case a if a < 0 => s"$a is a negative number"
      case b if b > 0 => s"$b is a positive number"
      case c => s"$c neither positive nor negative"
    }
}
```

Важно, чтобы ваши охранники не создавали не исчерпывающее соответствие (компилятор часто этого не поймает):

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case None    => doSomethingIfNone
}
```

Это выдает `MatchError` на нечетные числа. Вы должны либо учитывать все случаи, либо использовать случай соответствия шаблону:

```
def f(x: Option[Int]) = x match {
    case Some(i) if i % 2 == 0 => doSomething(i)
    case _ => doSomethingIfNoneOrOdd
}
```

## Сравнение шаблонов с классами case

Каждый класс case определяет экстрактор, который может использоваться для захвата членов класса case при сопоставлении с образцом:

```
case class Student(name: String, email: String)

def matchStudent1(student: Student): String = student match {
    case Student(name, email) => s"$name has the following email: $email" // extract name and
email
}
```

Все нормальные правила сопоставления шаблонов применяются - вы можете использовать защитные и постоянные выражения для управления соответствием:

```
def matchStudent2(student: Student): String = student match {
    case Student("Paul", _) => "Matched Paul" // Only match students named Paul, ignore email
    case Student(name, _) if name == "Paul" => "Matched Paul" // Use a guard to match students
named Paul, ignore email
    case s if s.name == "Paul" => "Matched Paul" // Don't use extractor; use a guard to match
students named Paul, ignore email
    case Student("Joe", email) => s"Joe has email $email" // Match students named Joe, capture
their email
    case Student(name, email) if name == "Joe" => s"Joe has email $email" // use a guard to
match students named Joe, capture their email
    case Student(name, email) => s"$name has email $email." // Match all students, capture
name and email
}
```

## Соответствие по выбору

Если вы соответствуете по типу Option :

```
def f(x: Option[Int]) = x match {
    case Some(i) => doSomething(i)
    case None    => doSomethingIfNone
}
```

Это функционально эквивалентно использованию `fold` или `map` / `getOrElse` :

```
def g(x: Option[Int]) = x.fold(doSomethingIfNone)(doSomething)
def h(x: Option[Int]) = x.map(doSomething).getOrElse(doSomethingIfNone)
```

## Шаблон соответствия закрытых черт

Когда шаблон соответствует объекту, тип которого является запечатанным признаком,

Scala проверяет во время компиляции, что все случаи «исчерпывающе согласованы»:

```
sealed trait Shape
case class Square(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape


def matchShape(shape: Shape): String = shape match {
    case Square(height, width) => "It's a square"
    case Circle(radius)        => "It's a circle"
    //no case for Point because it would cause a compiler warning.
}
```

Если новый `case class` для `Shape` позже добавил, все `match` заявления на `Shape` начнут бросать предупреждение компилятора. Это упростит рефакторинг: компилятор предупредит разработчика обо всех кодах, которые необходимо обновить.

## Совпадение шаблона с регулярным выражением

```
val emailRegex: Regex = "(.+)@(.+)\\.(.+)".r

"name@example.com" match {
  case emailRegex(userName, domain, topDomain) => println(s"Hi $userName from $domain")
  case _ => println(s"This is not a valid email.")
}
```

В этом примере регулярное выражение пытается сопоставить указанный адрес электронной почты. Если это произойдет, `userName` и `domain` извлекаются и печатаются. `topDomain` также извлекается, но с этим в этом примере ничего не делается. Вызов `.r` на String `str` эквивалентен `new Regex(str)` . Функция `r` доступна через неявное преобразование .

## Образец связующего (@)

Значок `@` связывает переменную с именем во время совпадения шаблона. Связанная переменная может быть либо полным совпадающим объектом, либо частью согласованного объекта:

```
sealed trait Shape
case class Rectangle(height: Int, width: Int) extends Shape
case class Circle(radius: Int) extends Shape
case object Point extends Shape

(Circle(5): Shape) match {
  case Rectangle(h, w) => s"rectangle, $h x $w."
  case Circle(r) if r > 9 => s"large circle"
  case c @ Circle(_) => s"small circle: ${c.radius}"  // Whole matched object is bound to c
  case Point => "point"
}
```

```
> res0: String = small circle: 5
```

Связанный идентификатор может использоваться в условных фильтрах. Таким образом:

```
case Circle(r) if r > 9 => s"large circle"
```

может быть записано как:

```
case c @ Circle(_) if c.radius > 9 => s"large circle"
```

Имя может быть привязано только к части совпадающего шаблона:

```
Seq(Some(1), Some(2), None) match {
  // Only the first element of the matched sequence is bound to the name 'c'
  case Seq(c @ Some(1), _*) => head
  case _ => None
}
```

```
> res0: Option[Int] = Some(1)
```

## Типы соответствия шаблонов

Сравнение шаблонов также можно использовать для проверки типа экземпляра, а не для использования `isInstanceOf[B]` :

```
val anyRef: AnyRef = ""

anyRef match {
  case _: Number       => "It is a number"
  case _: String       => "It is a string"
  case _: CharSequence => "It is a char sequence"
}
//> res0: String = It is a string
```

Порядок дел важен:

```
anyRef match {
  case _: Number       => "It is a number"
  case _: CharSequence => "It is a char sequence"
  case _: String       => "It is a string"
}
//> res1: String = It is a char sequence
```

Таким образом, он похож на классический оператор «switch» без пропущенных функций. Однако вы также можете сопоставлять совпадение и «извлекать» значения из соответствующего типа. Например:

```
case class Foo(s: String)
case class Bar(s: String)
case class Woo(s: String, i: Int)

def matcher(g: Any):String = {
  g match {
```

```
    case Bar(s) => s + " is classy!"
    case Foo(_) => "Someone is wicked smart!"
    case Woo(s, _) => s + " is adventerous!"
    case _ => "What are we talking about?"
  }
}

print(matcher(Foo("Diana")))  // prints 'Diana is classy!'
print(matcher(Bar("Hadas")))  // prints 'Someone is wicked smart!'
print(matcher(Woo("Beth", 27)))   // prints 'Beth is adventerous!'
print(matcher(Option("Katie")))  // prints 'What are we talking about?'
```

Обратите внимание, что в случае `Foo` и `Woo` мы используем знак подчеркивания ( `_` ) для «соответствия несвязанной переменной». То есть значение (в данном случае `Hadas` и `27` соответственно) не связано с именем и, следовательно, недоступно в обработчике для этого случая. Это полезно сокращать, чтобы соответствовать «любому» значению, не беспокоясь о том, что это за значение.

## Совпадение с образцом, скомпилированное как tablewitch или lookupswitch

`@switch` аннотации сообщают компилятор , что `match` заявление может быть заменено одной `tableswitch` инструкции на уровне байт - коды. Это небольшая оптимизация, которая позволяет удалить ненужные сравнения и переменные нагрузки во время выполнения.

Аннотации `@switch` работают только для совпадений с литеральными константами и `final val` идентификаторами `final val` . Если совпадение шаблона невозможно скомпилировать в качестве `tableswitch` / `lookupswitch` , компилятор поднимет предупреждение.

```
import annotation.switch

def suffix(i: Int) = (i: @switch) match {
  case 1 => "st"
  case 2 => "nd"
  case 3 => "rd"
  case _ => "th"
}
```

Результаты совпадают с результатами, полученными с обычным шаблоном:

```
scala> suffix(2)
res1: String = "2nd"

scala> suffix(4)
res2: String = "4th"
```

---

Из документа Scala (2.8+) - `@switch` :

> Аннотацию, которая должна применяться к выражению соответствия. Если присутствует, компилятор будет проверять, что совпадение было

скомпилировано для переключателя tableswitch или lookupswitch, и выдает ошибку, если вместо этого компилируется в серию условных выражений.

Из спецификации Java:

- tableswitch : «Таблица перехода к индексу и прыжок»
- lookupswitch : «Доступ к таблице перехода по ключевому совпадению и прыжку»

## Соответствие нескольких шаблонов сразу

| может использоваться для совпадения одного аргумента case с несколькими входами, чтобы получить тот же результат:

```
def f(str: String): String = str match {
  case "foo" | "bar" => "Matched!"
  case _ => "No match."
}

f("foo")  // res0: String = Matched!
f("bar")  // res1: String = Matched!
f("fubar")  // res2: String = No match.
```

Обратите внимание, что при совпадении **значений** этот способ работает хорошо, следующие соответствия **типов** вызовут проблемы:

```
sealed class FooBar
case class Foo(s: String) extends FooBar
case class Bar(s: String) extends FooBar

val d = Foo("Diana")
val h = Bar("Hadas")

// This matcher WILL NOT work.
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) | Bar(s) => print(s)  // Won't work: s cannot be resolved
    case Foo(_) | Bar(_) => _         // Won't work: _ is an unbound placeholder
    case _ => "Could not match"
  }
}
```

Если в последнем случае (с _ ) вам не нужно значение несвязанной переменной и просто хотите сделать что-то еще, вы в порядке:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(_) | Bar(_) => "Is either Foo or Bar."  // Works fine
    case _ => "Could not match"
  }
}
```

В противном случае вы останетесь с разбивкой своих дел:

```
def matcher(g: FooBar):String = {
  g match {
    case Foo(s) => s
    case Bar(s) => s
    case _ => "Could not match"
  }
}
```

## Выравнивание шаблонов на кортежах

Учитывая следующий `List` кортежей:

```
val pastries = List(("Chocolate Cupcake", 2.50),
                    ("Vanilla Cupcake", 2.25),
                    ("Plain Muffin", 3.25))
```

Согласование шаблонов может использоваться для обработки каждого элемента по-разному:

```
pastries foreach { pastry =>
  pastry match {
    case ("Plain Muffin", price) => println(s"Buying muffin for $price")
    case p if p._1 contains "Cupcake" => println(s"Buying cupcake for ${p._2}")
    case _ => println("We don't sell that pastry")
  }
}
```

В первом случае показано, как сопоставлять конкретную строку и получать соответствующую цену. Второй случай показывает использование if и tuple extract для сопоставления с элементами кортежа.

Прочитайте Соответствие шаблону онлайн: https://riptutorial.com/ru/scala/topic/661/соответствие-шаблону

# глава 53: Тестирование с помощью ScalaCheck

## Вступление

ScalaCheck - это библиотека, написанная на Scala и используемая для автоматизированного тестирования программ Scala или Java на основе свойств. ScalaCheck был первоначально вдохновлен библиотекой QuickBook от Haskell, но также отважился на нее.

ScalaCheck не имеет внешних зависимостей, кроме среды выполнения Scala, и отлично работает с sbt, инструментом сборки Scala. Он также полностью интегрирован в тестовые платформы ScalaTest и spec2.

## Examples

### Scalacheck с сообщениями scalatest и сообщения об ошибках

Пример использования scalache с использованием scalatest. Ниже мы проведем четыре теста:

- «показать пример передачи» - он проходит
- «показать простой пример без настраиваемого сообщения об ошибке» - просто сообщение об ошибке без подробностей, используется `&&` boolean operator
- «показать пример с сообщениями об ошибках в аргументе» - сообщение об ошибке в аргументе ( `"argument" |:` :) Вместо `&&` используется метод Props.all
- «показать пример с сообщениями об ошибках в команде» - сообщение об ошибке в команде ( `"command" |:` :) Вместо `&&` используется метод Props.all

```
import org.scalatest.prop.Checkers
import org.scalatest.{Matchers, WordSpecLike}

import org.scalacheck.Gen._
import org.scalacheck.Prop._
import org.scalacheck.Prop

object Splitter {
  def splitLineByColon(message: String): (String, String) = {
    val (command, argument) = message.indexOf(":") match {
      case -1 =>
        (message, "")
      case x: Int =>
        (message.substring(0, x), message.substring(x + 1))
    }
    (command.trim, argument.trim)
  }
```

```
  def splitLineByColonWithBugOnCommand(message: String): (String, String) = {
    val (command, argument) = splitLineByColon(message)
    (command.trim + 2, argument.trim)
  }

  def splitLineByColonWithBugOnArgument(message: String): (String, String) = {
    val (command, argument) = splitLineByColon(message)
    (command.trim, argument.trim + 2)
  }
}

class ScalaCheckSpec extends WordSpecLike with Matchers with Checkers {

  private val COMMAND_LENGTH = 4

  "ScalaCheckSpec " should {
```

```
    "show pass example" in {
      check {
        Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
          (chars, expArgument) =>
            val expCommand = new String(chars.toArray)
            val line = s"$expCommand:$expArgument"
            val (c, p) = Splitter.splitLineByColon(line)
            Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
        }

      }
    }
```

```
"show simple example without custom error message " in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        c === expCommand && expArgument === p
    }

  }
}
```

```
"show example with error messages on argument" in {
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnArgument(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
    }

  }
}
```

```
"show example with error messages on command" in {
```

```
  check {
    Prop.forAll(listOfN(COMMAND_LENGTH, alphaChar), alphaStr) {
      (chars, expArgument) =>
        val expCommand = new String(chars.toArray)
        val line = s"$expCommand:$expArgument"
        val (c, p) = Splitter.splitLineByColonWithBugOnCommand(line)
        Prop.all("command" |: c =? expCommand, "argument" |: expArgument =? p)
    }

  }
}
```

Выход (фрагменты):

```
[info] - should show example // passed
[info] - should show simple example without custom error message  *** FAILED ***
[info]    (ScalaCheckSpec.scala:73)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:73)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info] - should show example with error messages on argument *** FAILED ***
[info]    (ScalaCheckSpec.scala:86)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:86)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "" but got "2"
[info]       argument
[info] - should show example with error messages on command *** FAILED ***
[info]    (ScalaCheckSpec.scala:99)
[info]     Falsified after 0 successful property evaluations.
[info]     Location: (ScalaCheckSpec.scala:99)
[info]     Occurred when passed generated values (
[info]       arg0 = List(), // 3 shrinks
[info]       arg1 = ""
[info]     )
[info]     Labels of failing property:
[info]       Expected "2" but got ""
[info]       command
```

Прочитайте Тестирование с помощью ScalaCheck онлайн:
https://riptutorial.com/ru/scala/topic/9430/тестирование-с-помощью-scalacheck

# глава 54: Тестирование с помощью ScalaTest

## Examples

### Hello World Spec Test

Создайте класс тестирования в каталоге `src/test/scala` в файле `HelloWorldSpec.scala` . Поместите это в файл:

```
import org.scalatest.{FlatSpec, Matchers}

class HelloWorldSpec extends FlatSpec with Matchers {

  "Hello World" should "not be an empty String" in {
      val helloWorld = "Hello World"
      helloWorld should not be ("")
  }
}
```

- В этом примере используются `FlatSpec` и `Matchers` , которые являются частью библиотеки ScalaTest.
- `FlatSpec` позволяет тестировать тесты в стиле **Behavior-Driven Development (BDD)** . В этом стиле предложение используется для описания ожидаемого поведения данной единицы кода. Тест подтверждает, что код придерживается такого поведения. Дополнительную информацию см. В документации .

### Технический тест Cheatsheet

#### Настроить

Приведенные ниже тесты используют эти значения для примеров.

```
val helloWorld = "Hello World"
val helloWorldCount = 1
val helloWorldList = List("Hello World", "Bonjour Le Monde")
def sayHello = throw new IllegalStateException("Hello World Exception")
```

#### Проверка типа

Чтобы проверить тип для заданного значения `val` :

```
helloWorld shouldBe a [String]
```

Обратите внимание, что скобки здесь используются для получения типа `String` .

### Равная проверка

Чтобы проверить равенство:

```
helloWorld shouldEqual "Hello World"
helloWorld should === ("Hello World")
helloWorldCount shouldEqual 1
helloWorldCount shouldBe 1
helloWorldList shouldEqual List("Hello World", "Bonjour Le Monde")
helloWorldList === List("Hello World", "Bonjour Le Monde")
```

### Неравная проверка

Чтобы проверить неравенство:

```
helloWorld should not equal "Hello"
helloWorld !== "Hello"
helloWorldCount should not be 5
helloWorldList should not equal List("Hello World")
helloWorldList !== List("Hello World")
helloWorldList should not be empty
```

### Проверка длины

Чтобы проверить длину и / или размер:

```
helloWorld should have length 11
helloWorldList should have size 2
```

### Проверка исключений

Чтобы проверить тип и сообщение исключения:

```
val exception = the [java.lang.IllegalStateException] thrownBy {
  sayHello
}
exception.getClass shouldEqual classOf[java.lang.IllegalStateException]
exception.getMessage should include ("Hello World")
```

## Включите библиотеку ScalaTest с SBT

Используя SBT для управления зависимостью библиотеки , добавьте это в `build.sbt` :

```
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.0"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.0" % "test"
```

Более подробную информацию можно найти на сайте ScalaTest .

Прочитайте Тестирование с помощью ScalaTest онлайн:
https://riptutorial.com/ru/scala/topic/5506/тестирование-с-помощью-scalatest

# глава 55: Тип Параметрирование (Generics)

## Examples

### Тип опции

Хорошим примером параметризованного типа является тип Option . Это, по сути, просто следующее определение (с несколькими способами, определенными в типе):

```
sealed abstract class Option[+A] {
  def isEmpty: Boolean
  def get: A

  final def fold[B](ifEmpty: => B)(f: A => B): B =
    if (isEmpty) ifEmpty else f(this.get)

  // lots of methods...
}

case class Some[A](value: A) extends Option[A] {
  def isEmpty = false
  def get = value
}

case object None extends Option[Nothing] {
  def isEmpty = true
  def get = throw new NoSuchElementException("None.get")
}
```

Мы также видим, что это имеет параметризованный метод, `fold` , который возвращает что-то типа `B`

### Параметрированные методы

Тип возвращаемого метода может зависеть от *типа* параметра. В этом примере `x` - это параметр, `A` - *тип* `x` , который известен как *параметр типа* .

```
def f[A](x: A): A = x

f(1)          // 1
f("two")      // "two"
f[Float](3)   // 3.0F
```

Scala будет использовать вывод типа для определения типа возврата, который ограничивает, какие методы могут быть вызваны параметром. Таким образом, следует соблюдать осторожность: следующая ошибка времени компиляции, потому что `*` не определяется для каждого типа `A` :

```
def g[A](x: A): A = 2 * x  // Won't compile
```

## Определение списка Интов

```
trait IntList { ... }

class Cons(val head: Int, val tail: IntList) extends IntList { ... }

class Nil extends IntList { ... }
```

но что, если нам нужно определить список Boolean, Double и т. д.?

## Определение общего списка

```
trait List[T] {
  def isEmpty: Boolean
  def head: T
  def tail: List[T]
}

class Cons[T](val head: [T], val tail: List[T]) extends List[T] {
  def isEmpty: Boolean = false
}

class Nil[T] extends List[T] {
  def isEmpty: Boolean = true

  def head: Nothing = throw NoSuchElementException("Nil.head")

  def tail: Nothing = throw NoSuchElementException("Nil.tail")
}
```

Прочитайте Тип Параметрирование (Generics) онлайн:
https://riptutorial.com/ru/scala/topic/782/тип-параметрирование--generics-

# глава 56: Типы классов

## замечания

Чтобы избежать проблем с сериализацией, особенно в распределенных средах (например, Apache Spark ), наилучшей практикой является внедрение признака `Serializable` для экземпляров класса типов.

## Examples

### Класс простого типа

Класс типа - это просто `trait` с одним или несколькими параметрами типа:

```
trait Show[A] {
  def show(a: A): String
}
```

Вместо расширения класса типа для каждого поддерживаемого типа предоставляется неявный экземпляр класса type. Размещение этих реализаций в сопутствующем объекте класса типов позволяет неявное разрешение работать без какого-либо специального импорта:

```
object Show {
  implicit val intShow: Show[Int] = new Show {
    def show(x: Int): String = x.toString
  }

  implicit val dateShow: Show[java.util.Date] = new Show {
    def show(x: java.util.Date): String = x.getTime.toString
  }

  // ..etc
}
```

Если вы хотите гарантировать, что общий параметр, переданный функции, имеет экземпляр класса типа, используйте неявные параметры:

```
def log[A](a: A)(implicit showInstance: Show[A]): Unit = {
  println(showInstance.show(a))
}
```

Вы также можете использовать привязку к контексту :

```
def log[A: Show](a: A): Unit = {
  println(implicitly[Show[A]].show(a))
}
```

Вызовите вышеуказанный метод `log` как и любой другой метод. Он не сможет скомпилировать, если неявная реализация `Show[A]` не может быть найдена для `A` вы передаете в `log`

```
log(10) // prints: "10"
log(new java.util.Date(1469491668401L) // prints: "1469491668401"
log(List(1,2,3)) // fails to compile with
                 // could not find implicit value for evidence parameter of type
Show[List[Int]]
```

В этом примере реализуется класс `Show` type. Это обычный тип, используемый для преобразования произвольных экземпляров произвольных типов в `String` s. Несмотря на то, что у каждого объекта есть метод `toString` , не всегда ясно, полезен ли параметр `toString` . С помощью класса `Show` type вы можете гарантировать, что все, что передано в `log` имеет четко определенное преобразование в `String` .

## Расширение типа класса

В этом примере обсуждается расширение класса ниже.

```
trait Show[A] {
  def show: String
}
```

Чтобы сделать класс, который *вы* контролируете (и написан на Scala), расширьте класс типа, добавьте неявный его сопутствующий объект. Давайте покажем, как мы можем получить класс `Person` из этого примера, чтобы расширить `Show` :

```
class Person(val fullName: String) {
  def this(firstName: String, lastName: String) = this(s"$firstName $lastName")
}
```

Мы можем сделать этот класс расширить `Show` , добавив неявное к `Person` объекта компаньона «s:

```
object Person {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}
```

Сопутствующий объект *должен находиться в том же файле,* что и класс, поэтому вам нужен как `class Person` и `object Person` в том же файле.

Чтобы создать класс, который вы не контролируете или не написан в Scala, расширьте класс типа, добавьте неявный к сопутствующему объекту класса type, как показано в примере Simple Type Class .

Если вы не контролируете ни класс, ни класс типа, создайте неявное, как указано выше, и `import` его. Использование метода `log` в примере простого типа :

```
object MyShow {
  implicit val personShow: Show[Person] = new Show {
    def show(p: Person): String = s"Person(${p.fullname})"
  }
}

def logPeople(persons: Person*): Unit = {
  import MyShow.personShow
  persons foreach { p => log(p) }
}
```

## Добавление типов функций классов к типам

Внедрение Scala в классы типов довольно многословно. Один из способов уменьшить многословие - ввести так называемые «классы операций». Эти классы автоматически обертывают переменную / значение, когда они импортируются для расширения функциональности.

Чтобы проиллюстрировать это, давайте сначала создадим простой класс типа:

```
// The mathematical definition of "Addable" is "Semigroup"
trait Addable[A] {
  def add(x: A, y: A): A
}
```

Затем мы реализуем признак (создаем экземпляр класса типа):

```
object Instances {

  // Instance for Int
  // Also called evidence object, meaning that this object saw that Int learned how to be
added
  implicit object addableInt extends Addable[Int] {
    def add(x: Int, y: Int): Int = x + y
  }

  // Instance for String
  implicit object addableString extends Addable[String] {
    def add(x: String, y: String): String = x + y
  }

}
```

В настоящий момент было бы очень громоздко использовать наши Добавляемые экземпляры:

```
import Instances._
val three = addableInt.add(1,2)
```

Мы предпочли бы просто написать write `1.add(2)` . Поэтому мы создадим «Операционный класс» (также называемый «Класс Ops»), который всегда будет `Addable` тип, который реализует `Addable` .

```
object Ops {
  implicit class AddableOps[A](self: A)(implicit A: Addable[A]) {
    def add(other: A): A = A.add(self, other)
  }
}
```

Теперь мы можем использовать нашу новую функцию `add` как если бы она была частью `Int` и `String` :

```
object Main {

  import Instances._ // import evidence objects into this scope
  import Ops._       // import the wrappers

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

    println(1.add(5))
    println("mag".add("net"))
    // println(1.add(3.141)) // Fails because we didn't create an instance for Double

  }
}
```

Классы Ops могут автоматически создаваться макросами в библиотеке simulacrum :

```
import simulacrum._

@typeclass trait Addable[A] {
  @op("|+|") def add(x: A, y: A): A
}
```

Прочитайте Типы классов онлайн: https://riptutorial.com/ru/scala/topic/3835/типы-классов

# глава 57: функции

## замечания

Scala имеет первоклассные функции.

# Разница между функциями и методами:

Функция не является методом в Scala: функции являются значением и могут быть назначены как таковые. Методы (созданные с использованием `def` ), с другой стороны, должны принадлежать классу, признаку или объекту.

- Функции компилируются в класс, расширяющий признак (например, `Function1` ) во время компиляции и создаются при значении во время выполнения. Методы, с другой стороны, являются членами их класса, признака или объекта и не существуют вне этого.
- Метод может быть преобразован в функцию, но функция не может быть преобразована в метод.
- Методы могут иметь параметризацию типа, а функции - нет.
- Методы могут иметь значения параметров по умолчанию, тогда как функции не могут.

## Examples

### Анонимные функции

Анонимные функции - это функции, которые определены, но не назначены.

Ниже приведена анонимная функция, которая принимает два целых числа и возвращает сумму.

```
(x: Int, y: Int) => x + y
```

Полученное выражение может быть присвоено `val` :

```
val sum = (x: Int, y: Int) => x + y
```

Анонимные функции в основном используются в качестве аргументов для других функций. Например, функция `map` в коллекции ожидает в качестве аргумента другой функции:

```
// Returns Seq("FOO", "BAR", "QUX")
Seq("Foo", "Bar", "Qux").map((x: String) => x.toUpperCase)
```

Типы аргументов анонимной функции можно опустить: типы выводятся автоматически :

```
Seq("Foo", "Bar", "Qux").map((x) => x.toUpperCase)
```

Если есть только один аргумент, круглые скобки вокруг этого аргумента могут быть опущены:

```
Seq("Foo", "Bar", "Qux").map(x => x.toUpperCase)
```

# Подчеркивание сокращений

Существует еще более короткий синтаксис, который не требует имен для аргументов. Вышеприведенный фрагмент можно написать:

```
Seq("Foo", "Bar", "Qux").map(_.toUpperCase)
```

_ представляет анонимные аргументы функции позиционно. С анонимной функцией, имеющей несколько параметров, каждое вхождение _ будет ссылаться на другой аргумент. Например, два следующих выражения эквивалентны:

```
// Returns "FooBarQux" in both cases
Seq("Foo", "Bar", "Qux").reduce((s1, s2) => s1 + s2)
Seq("Foo", "Bar", "Qux").reduce(_ + _)
```

При использовании этой стенограммы любой аргумент, представленный позицией _ можно ссылаться только один раз и в том же порядке.

# Анонимные функции без параметров

Чтобы создать значение для анонимной функции, которая не принимает параметры, оставьте список параметров пустым:

```
val sayHello = () => println("hello")
```

## Состав

Состав функции позволяет использовать две функции и рассматривать их как одну функцию. Выраженная математическими терминами, заданная функцией `f(x)` и функцией `g(x)` , функция `h(x) = f(g(x))` .

Когда функция скомпилирована, она компилируется в тип, связанный с `Function1` . Scala предоставляет два метода в реализации `Function1` связанных с композицией: `andThen` и

`compose` . Метод `compose` соответствует приведенному выше математическому определению так:

```
val f: B => C = ...
val g: A => B = ...

val h: A => C = f compose g
```

`andThen` (думаю, `h(x) = g(f(x))` ) имеет более «DSL-подобное» чувство:

```
val f: A => B = ...
val g: B => C = ...

val h: A => C = f andThen g
```

Новая анонимная функция выделяется с закрытой над `f` и `g` . Эта функция связана с новой функцией `h` в обоих случаях.

```
def andThen(g: B => C): A => C = new (A => C){
  def apply(x: A) = g(self(x))
}
```

Если либо `f` либо `g` работает через побочный эффект, то вызов `h` вызовет все побочные эффекты `f` и `g` в порядке. То же самое относится к любым изменяемым изменениям состояния.

## Связь с PartialFunctions

```
trait PartialFunction[-A, +B] extends (A => B)
```

Каждый отдельный аргумент `PartialFunction` также является `Function1` . Это противоречит интуиции в формальном математическом смысле, но лучше подходит для объектно-ориентированного дизайна. По этой причине `Function1` не должен предоставлять постоянный `true` метод `isDefinedAt` .

Чтобы определить частичную функцию (которая также является функцией), используйте следующий синтаксис:

```
{ case i: Int => i + 1 } // or equivalently { case i: Int ⇒ i + 1 }
```

Для получения дополнительной информации ознакомьтесь с PartialFunctions .

Прочитайте функции онлайн: https://riptutorial.com/ru/scala/topic/477/функции

# глава 58: Функция более высокого порядка

## замечания

Scala делает все возможное, чтобы рассматривать методы и функции как синтаксически идентичные. Но под капотом они представляют собой разные понятия.

Метод является исполняемым кодом и не имеет представления значения.

Функция - это фактический экземпляр объекта типа `Function1` (или аналогичный тип другой арности). Его код содержится в методе его `apply` . Фактически, он просто действует как ценность, которая может быть передана.

Кстати, способность рассматривать функции как ценности - *это* именно то, что подразумевается под языком, поддерживающим функции более высокого порядка. Экземпляры функций - это подход Scala к реализации этой функции.

Фактическая функция более высокого порядка - это функция, которая либо принимает значение функции в качестве аргумента, либо возвращает значение функции. Но в Scala, поскольку все операции являются методами, более общее представление о методах, которые принимают или возвращают параметры функции. Таким образом, `map` , как определено в `Seq` может считаться «функцией более высокого порядка» из-за того, что ее параметр является функцией, но это не буквально функция; это метод.

## Examples

### Использование методов в качестве значений функций

Компилятор Scala автоматически преобразует методы в значения функций для передачи их в функции более высокого порядка.

```
object MyObject {
  def mapMethod(input: Int): String = {
    int.toString
  }
}

Seq(1, 2, 3).map(MyObject.mapMethod) // Seq("1", "2", "3")
```

В приведенном выше примере `MyObject.mapMethod` не является вызовом функции, а вместо этого передается для `map` в качестве значения. Действительно, для `map` *требуется* переданное ей значение функции, как видно из ее подписи. Подпись для `map List[A]` (

список объектов типа `A` ):

```
def map[B](f: (A) ⇒ B): List[B]
```

Часть `f: (A) => B` указывает, что параметр этого вызова метода является некоторой функцией, которая принимает объект типа `A` и возвращает объект типа `B` `A` и `B` произвольно определены. Возвращаясь к первому примеру, мы видим, что `mapMethod` принимает `Int` (что соответствует `A` ) и возвращает `String` (что соответствует `B` ). Таким образом `mapMethod` является допустимым значением функции для перехода к `map` . Мы могли бы переписать тот же код, как это:

```
Seq(1, 2, 3).map(x:Int => int.toString)
```

Это добавляет значение функции, которое может добавить ясность для простых функций.

## Функции высокого порядка (функция как параметр)

Функция более высокого порядка, в отличие от функции первого порядка, может иметь одну из трех форм:

- Один или несколько его параметров являются функцией, и она возвращает некоторое значение.
- Он возвращает функцию, но ни один из ее параметров не является функцией.
- Оба перечисленного: один или несколько его параметров являются функцией, и она возвращает функцию.

```
object HOF {
    def main(args: Array[String]) {
    val list =
List(("Srini","E"),("Subash","R"),("Ranjith","RK"),("Vicky","s"),("Sudhar","s"))
    //HOF
     val fullNameList= list.map(n => getFullName(n._1, n._2))

     }

    def getFullName(firstName: String, lastName: String): String = firstName + "." +
lastName
    }
```

Здесь функция map принимает функцию `getFullName(n._1,n._2)` в качестве параметра. Это называется **HOF (функция высшего порядка).**

## Аргументы ленивой оценки

Scala поддерживает ленивую оценку аргументов функции с помощью нотации: `def func(arg: => String)` . Такой аргумент функции может принимать обычный объект `String` или

функцию более высокого порядка с типом возвращаемого типа `String` . Во втором случае аргумент функции будет оцениваться при доступе к значениям.

См. Пример:

```
def calculateData: String = {
  print("Calculating expensive data! ")
  "some expensive data"
}

def dumbMediator(preconditions: Boolean, data: String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
    case false => None
  }
}

def smartMediator(preconditions: Boolean, data: => String): Option[String] = {
  print("Applying mediator")
  preconditions match {
    case true => Some(data)
    case false => None
  }
}

smartMediator(preconditions = false, calculateData)

dumbMediator(preconditions = false, calculateData)
```

вызов `smartMediator` возвращает значение None и печатает сообщение `"Applying mediator"` .

вызов `dumbMediator` возвращает значение None и выводит сообщение `"Calculating expensive data! Applying mediator"` .

Ленивая оценка может быть чрезвычайно полезна, когда вы хотите оптимизировать накладные расходы при расчете дорогостоящих аргументов.

Прочитайте Функция более высокого порядка онлайн:
https://riptutorial.com/ru/scala/topic/1642/функция-более-высокого-порядка

# глава 59: фьючерсы

## Examples

### Создание будущего

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

object FutureDivider {
    def divide(a: Int, b: Int): Future[Int] = Future {
        // Note that this is integer division.
        a / b
    }
}
```

Весьма просто, метод `divide` создает Будущее, которое будет разрешено с частным `a` над `b` .

### Потребление успешного будущего

Самый простой способ использовать *успешное* будущее - или, скорее, получить ценность в будущем - это использовать метод `map` . Предположим , что некоторый код вызывает `divide` метод `FutureDivider` объекта из «Создание будущего». Например , Что бы код должен выглядеть , чтобы получить частное `a` течение `b` ?

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(quotient)
        }
    }
}
```

### Потребление неудачного будущего

Иногда вычисление в Будущем может создать исключение, которое приведет к сбою Будущего. В примере «Создание будущего», если код вызова передал `55` и `0` методу `divide` ? Разумеется, это приведет к `ArithmeticException` после попытки разделить на ноль. Как это будет обрабатываться во потребляющем коде? На самом деле существует несколько способов устранения сбоев.

Обработать исключение с помощью `recover` и сопоставления шаблонов.

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
```

```
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

Обрабатывать исключение с `failed` проекцией, где исключение становится значением будущего:

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        // Note the use of the dot operator to get the failed projection and map it.
        eventualQuotient.failed.map {
            ex => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

## Объединение будущего

В предыдущих примерах были продемонстрированы индивидуальные особенности будущего, обработка успешных и неудачных случаев. Обычно, однако, обе функции обрабатываются гораздо более кратко. Вот пример, написанный более аккуратным и реалистичным образом:

```
object Calculator {
    def calculateAndReport(a: Int, b: Int) = {
        val eventualQuotient = FutureDivider divide(a, b)

        eventualQuotient map {
            quotient => println(s"Quotient: $quotient")
        } recover {
            case ex: ArithmeticException => println(s"It failed with: ${ex.getMessage}")
        }
    }
}
```

## Секвенирование и перемещение фьючерсов

В некоторых случаях необходимо вычислить переменную величину значений для отдельных фьючерсов. Предположим, что у вас есть `List[Future[Int]]` , но вместо этого нужно обработать `List[Int]` . Затем возникает вопрос, как превратить этот экземпляр `List[Future[Int]]` в `Future[List[Int]]` . Для этого существует метод `sequence` на объекте компаньона `Future` .

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.sequence(listOfFuture)
```

В общем случае `sequence` является широко известным оператором в мире функционального программирования , который преобразует `F[G[T]]` в `G[F[T]]` с ограничениями на `F` и `G` .

Существует альтернативный оператор, называемый `traverse` , который работает аналогично, но принимает функцию в качестве дополнительного аргумента. С помощью функции тождества `x => x` в качестве параметра она ведет себя как оператор `sequence` .

```
def listOfFuture: List[Future[Int]] = List(1,2,3).map(Future(_))
def futureOfList: Future[List[Int]] = Future.traverse(listOfFuture)(x => x)
```

Однако дополнительный аргумент позволяет изменять каждый будущий экземпляр внутри данного `listOfFuture` . Кроме того, первый аргумент не должен быть списком `Future` . Поэтому можно преобразовать пример следующим образом:

```
def futureOfList: Future[List[Int]] = Future.traverse(List(1,2,3))(Future(_))
```

В этом случае `List(1,2,3)` непосредственно передается в качестве первого аргумента, а функция тождества `x => x` заменяется функцией `Future(_)` чтобы аналогично обернуть каждое значение `Int` в `Future` . Преимущество этого заключается в том, что промежуточный `List[Future[Int]]` может быть опущен для повышения производительности.

## Объединить несколько фьючерсов - для понимания

*Понимание* - это компактный способ запуска блока кода, который зависит от успешного результата нескольких фьючерсов.

С `f1, f2, f3` три `Future[String]` , которые будут содержать строки `one, two, three` соответственно,

```
val fCombined =
    for {
        s1 <- f1
        s2 <- f2
        s3 <- f3
    } yield (s"$s1 - $s2 - $s3")
```

`fCombined` будет `Future[String]` содержащий строку `one - two - three` только все фьючерсы будут успешно завершены.

Обратите внимание, что здесь подразумевается неявный ExectionContext.

Кроме того, имейте в виду, что для понимания это просто синтаксический сахар для метода flatMap, поэтому построение объектов Future внутри тела исключило бы одновременное выполнение кодовых блоков, заключенных в фьючерсы, и привести к последовательному коду. Вы видите это на примере:

```
val result1 = for {
```

```
  first <- Future {
    Thread.sleep(2000)
    System.currentTimeMillis()
  }
  second <- Future {
    Thread.sleep(1000)
    System.currentTimeMillis()
  }
} yield first - second

val fut1 = Future {
  Thread.sleep(2000)
  System.currentTimeMillis()
}
val fut2 = Future {
  Thread.sleep(1000)
  System.currentTimeMillis()
}
val result2 = for {
  first <- fut1
  second <- fut2
} yield first - second
```

Значение, заключенное в `result1` объекта `result1` , всегда будет отрицательным, а `result2` - положительным.

Более подробную информацию о *понимании* и `yield` в целом см. По адресу http://docs.scala-lang.org/tutorials/FAQ/yield.html.

Прочитайте фьючерсы онлайн: https://riptutorial.com/ru/scala/topic/3245/фьючерсы

# глава 60: Частичные функции

## Examples

### Состав

Частичные функции часто используются для определения полной функции по частям:

```
sealed trait SuperType
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val pfA: PartialFunction[SuperType, Int] = {
  case A => 5
}

val pfB: PartialFunction[SuperType, Int] = {
  case B => 10
}

val input: Seq[SuperType] = Seq(A, B, C)

input.map(pfA orElse pfB orElse {
  case _ => 15
}) // Seq(5, 10, 15)
```

При этом использовании частичные функции предпринимаются в порядке конкатенации с `orElse` метода `orElse` . Как правило, предоставляется заключительная частичная функция, которая соответствует всем остальным случаям. В совокупности комбинация этих функций действует как общая функция.

Этот шаблон обычно используется для разделения проблем, когда функция может эффективно выполнять диспетчер для разрозненных путей кода. Это обычное явление, например, в методе приема Акка Акка .

### Использование с `collect`

Хотя частичная функция часто используется в качестве удобного синтаксиса для общих функций, путем включения окончательного подстановочного соответствия ( `case _` ) в некоторых методах их пристрастность является ключевым. Один очень распространенный пример в идиоматической Scala - это метод `collect` , определенный в библиотеке коллекций Scala. Здесь частичные функции позволяют общими функциями проверки элементов коллекции отображать и / или фильтровать их в одном компактном синтаксисе.

**Пример 1**

Предполагая, что мы имеем функцию квадратного корня, определенную как частичная

функция:

```
val sqRoot:PartialFunction[Double,Double] = { case n if n > 0 => math.sqrt(n) }
```

Мы можем вызвать его с помощью комбинатора `collect` :

```
List(-1.1,2.2,3.3,0).collect(sqRoot)
```

эффективно выполняя ту же операцию, что и:

```
List(-1.1,2.2,3.3,0).filter(sqRoot.isDefinedAt).map(sqRoot)
```

**Пример 2.**

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case class A(value: Int) extends SuperType
case class B(text: String) extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A(5), B("hello"), C, A(25), B(""))

input.collect {
  case A(value) if value < 10   => value.toString
  case B(text) if text.nonEmpty => text
} // Seq("5", "hello")
```

В приведенном выше примере есть несколько вещей:

- Левая часть каждого совпадения шаблонов эффективно выбирает элементы для обработки и включения в выход. Любое значение, которое не имеет подходящего `case` , просто опускается.
- Правая часть определяет применимую к делу обработку.
- Соответствие шаблону связывает переменную для использования в защитных выражениях (предложения `if` ) и в правой части.

## Основной синтаксис

Scala имеет специальный тип функции, называемый частичной функцией , которая расширяет нормальные функции, что означает, что экземпляр `PartialFunction` можно использовать везде, где ожидается `Function1` . Частичные функции могут быть определены анонимно с использованием синтаксиса `case` также используемого при сопоставлении шаблонов :

```
val pf: PartialFunction[Boolean, Int] = {
  case true => 7
}

pf.isDefinedAt(true) // returns true
pf(true) // returns 7
```

```
pf.isDefinedAt(false) // returns false
pf(false) // throws scala.MatchError: false (of class java.lang.Boolean)
```

Как видно из примера, частичная функция не обязательно должна быть определена во всей области ее первого параметра. Стандартный экземпляр `Function1` считается *полным* , что означает, что он определен для каждого возможного аргумента.

## Использование в качестве общей функции

Частичные функции очень распространены в идиоматической Scala. Они часто используются для их удобного синтаксиса на основе `case` для определения общих функций по признакам :

```
sealed trait SuperType // `sealed` modifier allows inheritance within current build-unit only
case object A extends SuperType
case object B extends SuperType
case object C extends SuperType

val input: Seq[SuperType] = Seq(A, B, C)

input.map {
  case A => 5
  case _ => 10
} // Seq(5, 10, 10)
```

Это сохраняет дополнительный синтаксис оператора `match` в обычной анонимной функции. Для сравнения:

```
input.map { item =>
  item match {
    case A => 5
    case _ => 10
  }
} // Seq(5, 10, 10)
```

Он также часто используется для выполнения декомпозиции параметров с использованием сопоставления шаблонов, когда кортеж или класс case передаются функции:

```
val input = Seq("A" -> 1, "B" -> 2, "C" -> 3)

input.map { case (a, i) =>
   a + i.toString
} // Seq("A1", "B2", "C3")
```

## Использование для извлечения кортежей в функции карты

Эти три функции карты эквивалентны, поэтому используйте вариацию, которую ваша команда считает наиболее читаемой.

```
val numberNames = Map(1 -> "One", 2 -> "Two", 3 -> "Three")

// 1. No extraction
numberNames.map(it => s"${it._1} is written ${it._2}" )

// 2. Extraction within a normal function
numberNames.map(it => {
    val (number, name) = it
    s"$number is written $name"
})

// 3. Extraction via a partial function (note the brackets in the parentheses)
numberNames.map({ case (number, name) => s"$number is written $name" })
```

Частичная функция **должна соответствовать всем входам** : любой случай, который не соответствует, генерирует исключение во время выполнения.

Прочитайте Частичные функции онлайн: https://riptutorial.com/ru/scala/topic/1638/частичные-функции

# глава 61: Черты

## Синтаксис

- черта ATrait {...}
- класс AClass (...) расширяет ATrait {...}
- класс AClass расширяет BClass с помощью ATrait
- класс AClass расширяет ATrait с помощью BTrait
- класс AClass расширяет ATrait с помощью BTrait с помощью CTrait
- класс ATrait расширяет BTrait

## Examples

### Сложная модификация с чертами

Вы можете использовать черты для изменения методов класса, используя черты в стиле стекирования.

В следующем примере показано, как можно складывать черты. Важное значение имеет упорядочение признаков. Используя различный порядок признаков, достигается различное поведение.

```
class Ball {
  def roll(ball : String) = println("Rolling : " + ball)
}

trait Red extends Ball {
  override def roll(ball : String) = super.roll("Red-" + ball)
}

trait Green extends Ball {
  override def roll(ball : String) = super.roll("Green-" + ball)
}

trait Shiny extends Ball {
  override def roll(ball : String) = super.roll("Shiny-" + ball)
}

object Balls {
  def main(args: Array[String]) {
    val ball1 = new Ball with Shiny with Red
    ball1.roll("Ball-1") // Rolling : Shiny-Red-Ball-1

    val ball2 = new Ball with Green with Shiny
    ball2.roll("Ball-2") // Rolling : Green-Shiny-Ball-2
  }
}
```

Обратите внимание, что `super` используется для вызова метода `roll()` в обоих признаках.

Только таким образом мы можем достичь штабелируемой модификации. В случаях суммируемой модификации порядок вызова метода определяется *правилом линеаризации* .

## Основные черты

Это самая базовая версия признака в Scala.

```
trait Identifiable {
  def getIdentifier: String
  def printIndentification(): Unit = println(getIdentifier)
}

case class Puppy(id: String, name: String) extends Identifiable {
  def getIdentifier: String = s"$name has id $id"
}
```

Поскольку ни супер класс не объявлен для признака `Identifiable` , по умолчанию она простирается от `AnyRef` класса. Поскольку определение `getIdentifier` не предусмотрено в `Identifiable` , класс `Puppy` должен его реализовать. Однако `Puppy` наследует реализацию `printIdentification` от `Identifiable` .

В REPL:

```
val p = new Puppy("K9", "Rex")
p.getIdentifier  // res0: String = Rex has id K9
p.printIndentification()  // Rex has id K9
```

## Решение проблемы алмаза

Проблема с алмазом или множественное наследование обрабатывается Scala с использованием Traits, которые аналогичны интерфейсам Java. Черты более гибкие, чем интерфейсы, и могут включать реализованные методы. Это делает черты похожими на mixins на других языках.

Scala не поддерживает наследование от нескольких классов, но пользователь может расширять несколько признаков в одном классе:

```
trait traitA {
  def name = println("This is the 'grandparent' trait.")
}

trait traitB extends traitA {
  override def name = {
    println("B is a child of A.")
    super.name
  }

}
```

```
trait traitC extends traitA {
  override def name = {
    println("C is a child of A.")
    super.name
  }
}

object grandChild extends traitB with traitC

grandChild.name
```

Здесь `grandChild` наследует как `traitB` и `traitC` , которые, в свою очередь, наследуют от `traitA` . На выходе (ниже) также показан порядок приоритета при первом разрешении реализации метода:

```
C is a child of A.
B is a child of A.
This is the 'grandparent' trait.
```

Обратите внимание, что когда `super` используется для вызова методов в `class` или `trait` , правило линеаризации входит в игру для решения иерархии вызовов. Порядок линеаризации для `grandChild` будет:

> grandChild -> traitC -> traitB -> traitA -> AnyRef -> Any

---

Ниже приведен еще один пример:

```
trait Printer {
  def print(msg : String) = println (msg)
}

trait DelimitWithHyphen extends Printer {
  override def print(msg : String) {
    println("-------------")
    super.print(msg)
  }
}

trait DelimitWithStar extends Printer  {
  override def print(msg : String) {
    println("*************")
    super.print(msg)
  }
}

class CustomPrinter extends Printer with DelimitWithHyphen with DelimitWithStar

object TestPrinter{
  def main(args: Array[String]) {
    new CustomPrinter().print("Hello World!")
  }
}
```

Эта программа печатает:

```
*************
-------------
Hello World!
```

Линеаризация для `CustomPrinter` будет:

> CustomPrinter -> DelimitWithStar -> DelimitWithHyphen -> Принтер -> AnyRef -> Любой

## Линеаризация

В случае стекируемой модификации Scala устраивает классы и черты в линейном порядке для определения иерархии вызовов метода, которая известна как *линеаризация* . Правило линеаризации используется *только* для тех методов, которые включают вызов метода через `super()` . Рассмотрим это на примере:

```
class Shape {
  def paint (shape: String): Unit = {
    println(shape)
  }
}

trait Color extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Color ")
  }
}

trait Blue extends Color {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Blue ")
  }
}

trait Border extends Shape {
  abstract override def paint (shape : String) {
    super.paint(shape + "Border ")
  }
}

trait Dotted extends Border {
  abstract override def paint (shape : String) {
    super.paint(shape + "with Dotted ")
  }
}

class MyShape extends Shape with Dotted with Blue {
  override def paint (shape : String) {
    super.paint(shape)
  }
}
```

Линеаризация происходит от *начала до фронта* . В этом случае,

1. Первая `Shape` будет линеаризована, которая выглядит так:

```
Shape -> AnyRef -> Any
```

2. Затем `Dotted` линеаризуется:

   ```
   Dotted -> Border -> Shape -> AnyRef -> Any
   ```

3. Следующий в строке - `Blue` . Обычно линеаризация `Blue` :

   ```
   Blue -> Color -> Shape -> AnyRef -> Any
   ```

   потому что в линеаризации `MyShape` до сих пор ( *Шаг 2* ) уже появился `Shape -> AnyRef -> Any` . Следовательно, он игнорируется. Таким образом, `Blue` линеаризации будет:

   ```
   Blue -> Color -> Dotted -> Border -> Shape -> AnyRef -> Any
   ```

4. Наконец, будет добавлен `Circle` и окончательный порядок линеаризации будет:

   > Круг -> Синий -> Цвет -> Пунктирный -> Граница -> Форма -> AnyRef -> Любой

Этот порядок линеаризации решает порядок обращения методов, когда `super` используется в любом классе или признаке. Первая реализация метода справа вызывается в порядке линеаризации. Если `new MyShape().paint("Circle ")` , он будет печатать:

```
Circle with Blue Color with Dotted Border
```

Более подробную информацию о линеаризации можно найти здесь .

Прочитайте Черты онлайн: https://riptutorial.com/ru/scala/topic/1056/черты

# глава 62: Экстракторы

## Синтаксис

- val extractor (extractValue1, _ / * игнорируется второе извлеченное значение * /) = valueToBeExtracted
- valueToBeExtracted match {case extractor (extractValue1, _) => ???}
- val (tuple1, tuple2, tuple3) = tupleWith3Elements
- объект Foo {def unapply (foo: Foo): Option [String] = Some (foo.x); }

## Examples

### Экстракторы кортежей

`x` и `y` извлекаются из кортежа:

```
val (x, y) = (1337, 42)
// x: Int = 1337
// y: Int = 42
```

Чтобы игнорировать значение, используйте _ :

```
val (_, y: Int) = (1337, 42)
// y: Int = 42
```

Чтобы распаковать экстрактор:

```
val myTuple = (1337, 42)
myTuple._1  // res0: Int = 1337
myTuple._2  // res1: Int = 42
```

Обратите внимание, что кортежи имеют максимальную длину 22 и, следовательно, будет работать от `._1` до `._22` (при условии, что кортеж имеет по крайней мере такой размер).

Экстракторы кортежей могут использоваться для предоставления символических аргументов для буквенных функций:

```
val persons = List("A." -> "Lovelace", "G." -> "Hopper")
val names = List("Lovelace, A.", "Hopper, G.")

assert {
  names ==
    (persons map { name =>
      s"${name._2}, ${name._1}"
    })
}
```

```
assert {
  names ==
    (persons map { case (given, surname) =>
      s"$surname, $given"
    })
}
```

## Экстракторы корпуса

Класс case - это класс с большим количеством стандартных шаблонов. Одно из преимуществ этого заключается в том, что Scala упрощает использование экстракторов с классами case.

```
case class Person(name: String, age: Int)  // Define the case class
val p = Person("Paola", 42)  // Instantiate a value with the case class type

val Person(n, a) = p  // Extract values n and a
// n: String = Paola
// a: Int = 42
```

На этом этапе, как `n` и `a` являются `val` s в программе и могут быть доступны как таковые: они, как говорят, были «извлечены» из p. Продолжение:

```
val p2 = Person("Angela", 1337)

val List(Person(n1, a1), Person(_, a2)) = List(p, p2)
// n1: String = Paola
// a1: Int = 42
// a2: Int = 1337
```

Здесь мы видим две важные вещи:

- Экстракция может происходить на «глубоких» уровнях: можно выделить свойства вложенных объектов.
- Не все элементы *должны* быть извлечены. Подстановочный _ символ указывает на то , что именно эта величина может быть что угодно, и игнорируется. Никакой `val` не создается.

В частности, это может упростить сопоставление по коллекциям:

```
val ls = List(p1, p2, p3)  // List of Person objects
ls.map(person => person match {
  case Person(n, a) => println("%s is %d years old".format(n, a))
})
```

Здесь у нас есть код, который использует экстрактор, чтобы явно проверить, что `person` является объектом `Person` и немедленно вытащить переменные, которые нас волнуют: `n` и `a` .

## Unapply - Пользовательские экстракторы

Пользовательское извлечение может быть записано путем реализации метода `unapply` и возврата значения типа `Option` :

```
class Foo(val x: String)

object Foo {
  def unapply(foo: Foo): Option[String] = Some(foo.x)
}

new Foo("42") match {
  case Foo(x) => x
}
// "42"
```

Возвращаемый тип `unapply` может быть чем-то иным, чем `Option` , если возвращаемый тип предоставляет методы `get` и `isEmpty` . В этом примере `Bar` определяется этими методами, а `unapply` возвращает экземпляр `Bar` :

```
class Bar(val x: String) {
  def get = x
  def isEmpty = false
}

object Bar {
  def unapply(bar: Bar): Bar = bar
}

new Bar("1337") match {
  case Bar(x) => x
}
// "1337"
```

Возвращаемый тип `unapply` также может быть `Boolean` , что является особым случаем, который не несет требований `get` и `isEmpty` выше. Однако обратите внимание в этом примере, что `DivisibleByTwo` - это объект, а не класс, и не принимает параметр (и, следовательно, этот параметр не может быть связан):

```
object DivisibleByTwo {
  def unapply(num: Int): Boolean = num % 2 == 0
}

4 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// yes

3 match {
  case DivisibleByTwo() => "yes"
  case _ => "no"
}
// no
```

Помните, что `unapply` идет в сопутствующем объекте класса, а не в классе. Приведенный

выше пример будет ясен, если вы поймете это различие.

### Инфраструктура Extractor Infix

Если класс case имеет ровно два значения, его экстрактор может использоваться в нотации infix.

```
case class Pair(a: String, b: String)
val p: Pair = Pair("hello", "world")
val x Pair y = p
//x: String = hello
//y: String = world
```

Любой экстрактор, который возвращает 2-кортеж, может работать таким образом.

```
object Foo {
    def unapply(s: String): Option[(Int, Int)] = Some((s.length, 5))
}
val a Foo b = "hello world!"
//a: Int = 12
//b: Int = 5
```

### Экстракторы регулярных выражений

Регулярное выражение с группируемыми частями может использоваться как экстрактор:

```
scala> val address = """(.+):(\d+)""".r
address: scala.util.matching.Regex = (.+):(\d+)

scala> val address(host, port) = "some.domain.org:8080"
host: String = some.domain.org
port: String = 8080
```

Обратите внимание, что когда он не сопоставляется, `MatchError` будет `MatchError` во время выполнения:

```
scala> val address(host, port) = "something not a host and port"
scala.MatchError: something not a host and port (of class java.lang.String)
```

### Трансформаторные экстракторы

Поведение экстрактора может использоваться для получения произвольных значений с их ввода. Это может быть полезно в сценариях, где вы хотите иметь возможность воздействовать на результаты преобразования в случае успешного преобразования.

Рассмотрим в качестве примера различные форматы имен пользователей, которые можно использовать в среде Windows :

```
object UserPrincipalName {
```

```
  def unapply(str: String): Option[(String, String)] = str.split('@') match {
    case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
    case _ => None
  }
}

object DownLevelLogonName {
  def unapply(str: String): Option[(String, String)] = str.split('\\') match {
    case Array(d, u) if u.length > 0 && d.length > 0 => Some((d, u))
    case _ => None
  }
}

def getDomain(str: String): Option[String] = str match {
  case UserPrincipalName(_, domain) => Some(domain)
  case DownLevelLogonName(domain, _) => Some(domain)
  case _ => None
}
```

На самом деле можно создать экстрактор, демонстрирующий оба поведения, путем расширения типов, которые он может сопоставить:

```
object UserPrincipalName {
  def unapply(obj: Any): Option[(String, String)] = obj match {
    case upn: UserPrincipalName => Some((upn.username, upn.domain))
    case str: String => str.split('@') match {
      case Array(u, d) if u.length > 0 && d.length > 0 => Some((u, d))
      case _ => None
    }
    case _ => None
  }
}
```

В общем, экстракторы - это просто удобная переформатирование шаблона `Option` , применительно к методам с именами, такими как `tryParse` :

```
UserPrincipalName.unapply("user@domain") match {
  case Some((u, d)) => ???
  case None => ???
}
```

Прочитайте Экстракторы онлайн: https://riptutorial.com/ru/scala/topic/930/экстракторы

# кредиты

| S. No | Главы | Contributors |
|---|---|---|
| 1 | Начало работы с языком Scala | 4444, Andy Hayden, Ani Menon, Community, David G., David Portabella, dk14, Donald.McLean, Gabriele Petronella, Grzegorz Oledzki, implicitdef, isaias-b, J Atkin, Jean, Jonathan, mammothbane, marcospereira, Marek Skiba, mdarwin, Nathaniel Ford, NeoWelkin, Nicofisi, Priya, rolve, Shoe, sschaef, Thomas Andrews, Tyler James Harden, Ven, Vogon Jeltz |
| 2 | Implicits | Andy Hayden, dimitrisli, Gábor Bakos, HTNW, implicitdef, ipoteka, Jose Antonio Jimenez Saez, Michael Zajac, Nathaniel Ford, nattyddubbs, Simon, spiffman, Suma, Timo, vsminkov |
| 3 | JSON | ipoteka, John, Muki, Nathaniel Ford, pedrorijo91, suj1th, void, Wogan, zoitol |
| 4 | Quasiquotes | gregghz |
| 5 | Scala.js | Camilo Sampedro |
| 6 | Scaladoc | Camilo Sampedro, Gábor Bakos, Nathaniel Ford |
| 7 | scalaz | chengpohi |
| 8 | Streams | jwvh, Nathaniel Ford, Oleg Pyzhcov |
| 9 | Var, Val и Def | Aamir, John Starich, jwvh, linkhyrule5, Nathaniel Ford, Shastick, Shuklaswag, stefanobaghino, ZbyszekKr |
| 10 | Аннотации | Gábor Bakos, Nathaniel Ford, Thomas Matecki |
| 11 | Библиотека непрерывности | dmitry, HTNW |
| 12 | В то время как петли | J Cracknell, Nathaniel Ford |
| 13 | Внедрение зависимости | Hoang Ong |
| 14 | Вывод типа | Gábor Bakos, Nathaniel Ford, suj1th |
| 15 | Динамический вызов | HTNW |

| | | |
|---|---|---|
| 16 | Для выражений | Andy Hayden, J Cracknell, jwvh, LivingRobot, Nathaniel Ford, ScientiaEtVeritas |
| 17 | Единые абстрактные типы методов (типы SAM) | Gábor Bakos, Gabriele Petronella, Nathaniel Ford |
| 18 | Если выражения | corvus_192, Nathaniel Ford, ScientiaEtVeritas |
| 19 | Интерполяция строк | Andy Hayden, Ayberk, Brian, implicitdef, J Cracknell, Nadim Bahadoor |
| 20 | Карринг | Adamos Loizou, alphaloop, Amr Gawish, dimitrisli, Luka Jacobowitz, Nathaniel Ford, rjsvaljean, Suma, vise890 |
| 21 | Класс опций | Bruce Lowe, CPS, earldouglas, evan.oman, Governa, John Starich, Matthew Scharley, Nathaniel Ford, R Pieters, ScientiaEtVeritas, suj1th, Tzach Zohar, Vasiliy Levykin |
| 22 | Классы и объекты | Aamir, Gábor Bakos, mdarwin, mirosval, MSmedberg, Nathaniel Ford, ScientiaEtVeritas, steve, Sudhir Singh, Tzach Zohar, vivek |
| 23 | Классы классов | Andy Hayden, Dan Simon, dk14, Gábor Bakos, HTNW, J Cracknell, keegan, made raka teja, Marc Grue, Nathaniel Ford, pedrorijo91, Rumoku, ScientiaEtVeritas, suj1th, Suma |
| 24 | Коллекции | Anton, Camilo Sampedro, deepkimo, Donald.McLean, doub1ejack, EdgeCaseBerg, Filippo Vitale, George, implicitdef, ipoteka, Jason, John Starich, Mr D, Nathaniel Ford, raam86, Shastick, Suma, Tundebabzy, Vasiliy Levykin |
| 25 | Комбинировщики Parser | Nathaniel Ford |
| 26 | Кортеж | corvus_192, evan.oman, Lawsy, Nathaniel Ford |
| 27 | Лучшие практики | corvus_192, ipoteka, Nathaniel Ford, RamenChef, Sarvesh Kumar Singh, Shuklaswag |
| 28 | макрос | gregghz, HTNW, Nathaniel Ford |
| 29 | Монады | ipoteka, Nathaniel Ford |
| 30 | Настройка Scala | Hristo Iliev, Matas Vaitkevicius, Nathaniel Ford, Rjk |
| 31 | Обрабатывающие устройства (меры) | Gábor Bakos |

| | | |
|---|---|---|
| 32 | Обработка XML | Nathaniel Ford, Rockie Yang, vsnyc |
| 33 | Обработка ошибок | Andy Hayden, Graham, John Starich, made raka teja, mnoronha, Nathaniel Ford, Simon, Suma, tacos_tacos_tacos, Tzach Zohar |
| 34 | Объем | Camilo Sampedro |
| 35 | Операторы в Scala | Gábor Bakos, Shaido, Suminda Sirinath S. Dharmasena |
| 36 | отражение | Sachin Janani |
| 37 | пакеты | Alex Javarotti, Nathaniel Ford, NetanelRabinowitz |
| 38 | Параллельные коллекции | Nathaniel Ford, Shuklaswag |
| 39 | Перегрузка оператора | corvus_192, implicitdef, inzi, mnoronha, Nathaniel Ford, Simon |
| 40 | Перечисления | Andy Hayden, Cortwave, Daniel Schröter, Gábor Bakos, implicitdef, ipoteka, Nathaniel Ford, phantomastray, Red Mercury |
| 41 | Пользовательские функции для улья | Camilo Sampedro |
| 42 | Программирование на уровне | J Cracknell |
| 43 | Работа с данными в неизменном стиле | Filippo Vitale |
| 44 | Работа с люлькой | Bianca Tesila, Nathaniel Ford, Rjk |
| 45 | Разница типов | acjay, J Cracknell, Reactormonk |
| 46 | Регулярные выражения | dmitry, J Cracknell, Nathaniel Ford |
| 47 | Рекурсия | Dmitry Bystritsky, Gábor Bakos, jilen, jwvh, michael_s, ScientiaEtVeritas, teldosas |
| 48 | Самостоятельные типы | Gábor Bakos, irundaia |
| 49 | Символьные литералы | ZbyszekKr |

| | | |
|---|---|---|
| 50 | синхронизированный | Gábor Bakos |
| 51 | Совместимость Java | Andrzej Jozwik, Dan Hulme, Gábor Bakos, mvn, the21st, thekingofkings |
| 52 | Соответствие шаблону | Ali Dehghani, Andrzej Jozwik, Andy Hayden, CPS, Dan Simon, Daniel Werner, Filippo Vitale, Gábor Bakos, implicitdef, insan-e, jilen, jozic, JRomero, Justin Bailey, Louis F., mammothbane, Matt, Nadim Bahadoor, Nathaniel Ford, Peter Neyens, Sergio, Shastick, Shoe, Simon, Suma, T.Grottker, user6062072, vdebergue, vsminkov, Yagüe |
| 53 | Тестирование с помощью ScalaCheck | Andrzej Jozwik |
| 54 | Тестирование с помощью ScalaTest | Nadim Bahadoor, Nathaniel Ford |
| 55 | Тип Параметрирование ( Generics) | akauppi, Andy Hayden, Eero Helenius, Nathaniel Ford, vivek |
| 56 | Типы классов | Arseniy Zhizhelev, Daniel C. Sobral, Gábor Bakos, gregghz, Nathaniel Ford, TomTom, Yawar |
| 57 | функции | Aravindh S, Archeg, Camilo Sampedro, ches, corvus_192, Dawny33, Gábor Bakos, Gabriele Petronella, implicitdef, ipoteka, Jean, jwvh, michael_s, Nathaniel Ford, raam86, rjsvaljean, ScientiaEtVeritas, Shastick, stefanobaghino, Sven Koschnicke, vise890, wheaties |
| 58 | Функция более высокого порядка | acjay, ches, Nathaniel Ford, nukie, Rajat Jain, Srini |
| 59 | фьючерсы | isaias-b, kevin628, Nathaniel Ford, nukie, Shastick |
| 60 | Частичные функции | acjay, Akash Sethi, David Leppik, dimitrisli, jwvh, Suma, Tzach Zohar |
| 61 | Черты | André Laszlo, Andy Hayden, Donald.McLean, Louis F., Nathaniel Ford, Rumoku, Sudhir Singh, Vogon Jeltz |
| 62 | Экстракторы | Andy Hayden, Dan Hulme, Dan Simon, Gábor Bakos, gilad hoch, Idloj, J Cracknell, jwvh, knutwalker, Łukasz, Martin Seeler, Michael Ahlers, Nathaniel Ford, Suma, W.P. McNeill |