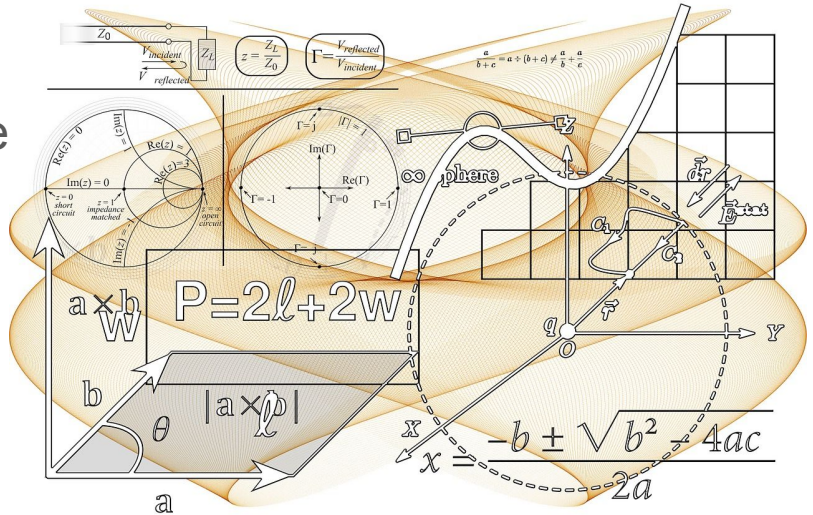
Scala School

Лекция 3: Функции и все о них (часть 1)

План лекции

- Класс Function
- Композиция функций
- Чистые функции
- Частичное применение, каррирование
- PartialFunction
- Хвостовая рекурсия



Класс Function

Анонимный класс

```
trait Animal {
  def sound(): Unit
}

scala> val cow = new Animal {
  def sound() = println("moo")
}
cow: Animal = $anon$1@29852487

scala> cow.sound()
moo
```

Класс Function

Функция - объект, имплементирующий `trait Function`

```
trait Function1[-T1, +R] extends AnyRef {
    def apply(v1: T1): R
}
```

Класс Function

```
scala> val myFun = new Function1[Int, Int] {
  override def apply(x: Int) = x + 1
}
myFun: Int => Int = <function1>

scala> myFun(3)
res13: Int = 4
```

Анонимные функции

(Также используется название function literal)

```
scala> val myFun = (x: Int) => x + 1
myFun: Int => Int = <function1>

// Синтаксический сахар для
new Function1[Int, Int] {
  override def apply(x: Int): Int = x + 1
}

scala> myFun(3)
res0: Int = 4
```

Анонимные функции

Можно создавать функции, имеющие от 0 до 22 аргументов

```
scala> val fun0 = () => "zero parameters"
fun0: () => String = <function0>

scala> val fun2 = (x: Int, y: Int) => x + y
fun2: (Int, Int) => Int = <function2>
```

Анонимные функции

Можно создавать анонимные функции, имеющие от 0 до 22 аргументов

А если больше?!

```
Error:(8, 265) implementation restricts functions to 22
parameters
```

Синтаксический сахар

Чтобы не писать тип `FunctionX[T1, T2, …]` в Scala существует отличный синтаксический сахар:

`(T1, T2, ..) => R` эквивалентно `FunctionX[A, B, .. , R]`

Например:

`(A, B, C) => R` эквивалентно `Function3[A, B, C, R]`

Функция как аргумент

В Scala функция может быть аргументом

```
scala> def applyTwice(f: Int => Int, x: Int): Int = f(f(x))
applyTwice: (f: Int => Int, x: Int)Int

scala> applyTwice(myFun, 0)
res16: Int = 2
```

Eta Expansion

Eta Expansion - создание объекта функции из метода

```
def plusTwo(x: Int) = x + 2


scala> val plusTwoEta = plusTwo _
plusTwoEta: Int => Int = <function1>
```

Eta Expansion

Eta Expansion - создание объекта функции из метода

```
def plusTwo(x: Int) = x + 2


// Что произошло за кадром?
val plusTwoEta = new Function1[Int, Int] {
  override def apply(x: Int) = plusTwo(x)
}
```

Eta Expansion

Когда передаем метод как параметр типа функция, происходит Eta Expansion этого метода в объект - функцию

```
def plusTwo(x: Int) = x + 2


scala> Some(4).map(plusTwo)
res17: Option[Int] = Some(6)
```

Функции и методы

- **Метод (def)**
  Нет объекта, по сути - Java-метод

- **Функция (A => B)**
  Есть объект типа функция
  Можно получить из метода (Eta Expansion)

http://stackoverflow.com/questions/2529184/difference-between-method-and-function-in-scala

Композиция функций

Функции высшего порядка

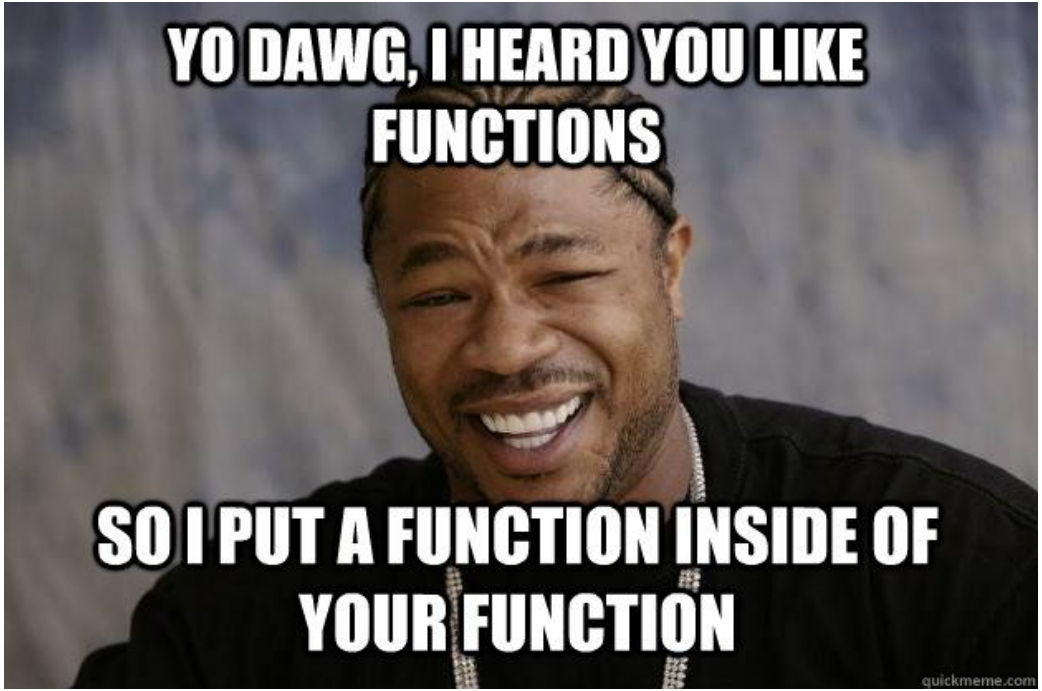
Функция высшего порядка (**higher order function**) - функция, обладающая хотя бы одним из свойств:

- Один или более аргументов - функции
- Результат - функция

Пример: `map`, `flatMap`, `filter`, ...

Функции первого порядка (**first order functions**) - все остальные функции

Композиция функций



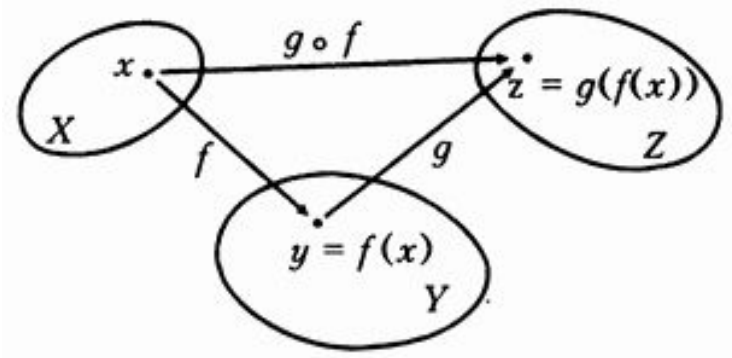
Композиция функций

Функции:

f(x): X => Y, g(y): Y => Z

Тогда их **композиция** g(f(x)) ~ gf - применение g к результату f

```
def f(x: String) = x + "f"
def g(x: String) = x + "g"
scala> f("a")
res19: String = af
scala> g(f("a")) // == g("af")
res18: String = afg
```



Композиция функций - compose

```
object FunctionCompositions extends App {
  def f(str: String) = str + " foo" // f(x)
  def g(str: String) = str + " bar" // g(x)




}
```

Композиция функций - compose

```
object FunctionCompositions extends App {
  def f(str: String) = str + " foo" // f(x)
  def g(str: String) = str + " bar" // g(x)

  val fooThenBar = g _ compose f _ // g(f(x)) == addBar(addFoo(s))
  println(fooThenBar("scala")) // scala foo bar



}
```

Композиция функций - andThen

```
object FunctionCompositions extends App {
  def f(str: String) = str + " foo" // f(x)
  def g(str: String) = str + " bar" // g(x)

  val fooThenBar = g _ compose f _ // g(f(x))
  println(fooThenBar("scala")) // scala foo bar

  val barThenFoo = g _ andThen f _ // f(g(x)
  println(barThenFoo("scala")) // scala bar foo
}
```

Композиция функций

```
object FunctionCompositions extends App {
  val inc = (x: Int) => x + 1 // Int => Int
  val toStr = (x: Int) => s"my int is $x" // Int => String




}
```

Композиция функций

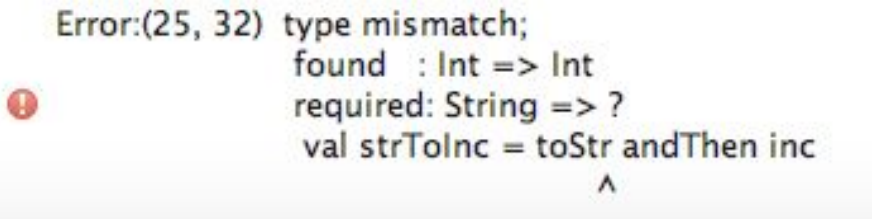
```
object FunctionCompositions extends App {
  val inc = (x: Int) => x + 1 // Int => Int
  val toStr = (x: Int) => s"my int is $x" // Int => String

  val incToStr = toStr compose inc
  println(incToStr(5)) // my int is 6



}
```

Композиция функций

```
object FunctionCompositions extends App {
  val inc = (x: Int) => x + 1 // Int => Int
  val toStr = (x: Int) => s"my int is $x" // Int => String

  val incToStr = toStr compose inc
  println(incToStr(5)) // my int is 6

  val strToInc = toStr andThen inc // ???
}
```

Композиция функций

```
object FunctionCompositions extends App {
  val inc = (x: Int) => x + 1 // Int => Int
  val toStr = (x: Int) => s"my int is $x" // Int => String

  val incToStr = toStr compose inc
  println(incToStr(5)) // my int is 6

  val strToInc = toStr andThen inc //
}
```



Чистые функции

Чистые функции

Чистая функция (**pure function**) - функция со свойствами:

1. **Детерминированная** - для одинакового набора входных значений возвращает одинаковый результат

```
System.currentTimeMillis() // Недетерминированная
```

Чистые функции

Чистая функция (**pure function**) - функция со свойствами:

1. **Детерминированная** - для одинакового набора входных значений возвращает одинаковый результат

```
System.currentTimeMillis() // Недетерминированная
```

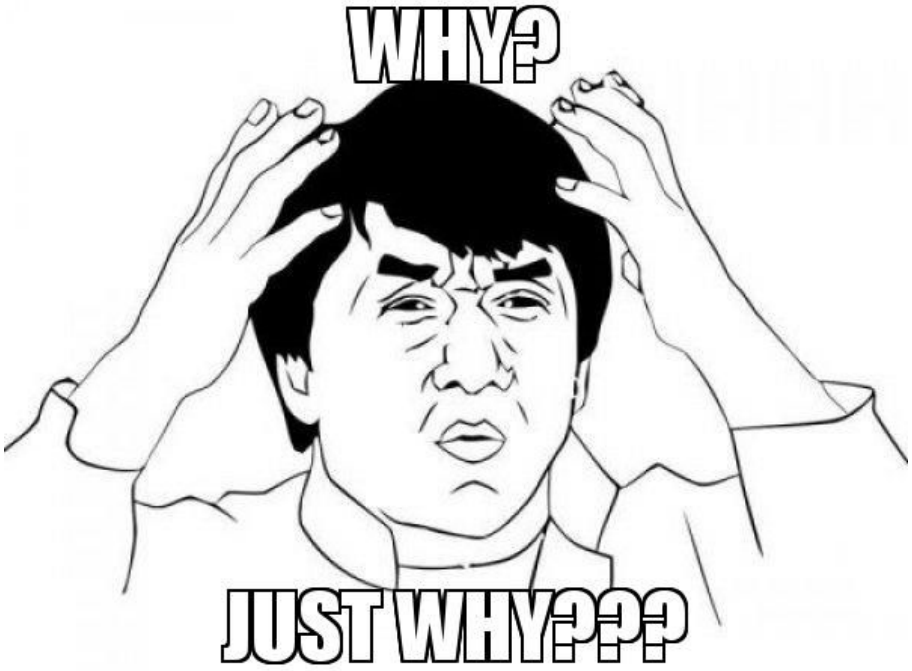
2. **Без побочных эффектов.** Не модифицирует глобальные переменные, входные параметры, не осуществляет ввод-вывод и т.д. (не изменяет внешнее состояние).

```
println(str)    Random.nextInt() // Есть побочные эффекты
```

Чистые функции

```
object PureFunctions {
  var x = 0

  def changeState(z: Int) = { // Not pure
    val res = z + x
    x += z
    res
  }
  def useState(z: Int) = z + x // Not pure
  def pure(x: Double, y: String) = sin(x) + y.length // Pure
}
```

Чистые функции - зачем?



Чистые функции - зачем?

- Воспроизводимый результат

Чистые функции - зачем?

- Воспроизводимый результат
- Легко распараллеливать

Чистые функции - зачем?

- Воспроизводимый результат
- Легко распараллеливать
- Мемоизация (memoization) - кеширование и повторное использование результата выполнения

Чистые функции - зачем?

- Воспроизводимый результат
- Легко распараллеливать
- Мемоизация (memoization) - кеширование и повторное использование результата выполнения
- Ленивое выполнение (lazy evaluation)

Частичное применение функций

Частичное применение функций

Частичное применение функции (**partial application**) - фиксирование части аргументов функции, в результате чего получается новая функция от меньшего числа аргументов

Частичное применение функции

```
object PartialApplication {
  val sumThree = (x: Int, y: Int, z: Int) => x + y + z
  val res = sumThree(1, 2, 3) // 6









}
```

Частичное применение функции

```
object PartialApplication {
  val sumThree = (x: Int, y: Int, z: Int) => x + y + z
  val res = sumThree(1, 2, 3) // 6

  val sumLastTwo = sumThree(1, _:Int, _:Int) // (Int, Int) => Int
  val res1 = sumLastTwo(2, 3) // 6



}
```

Частичное применение функции

```
object PartialApplication {
  val sumThree = (x: Int, y: Int, z: Int) => x + y + z
  val res = sumThree(1, 2, 3) // 6

  val sumLastTwo = sumThree(1, _:Int, _:Int) // (Int, Int) => Int
  val res1 = sumLastTwo(2, 3) // 6

  val sumLast = sumThree(1, 2, _:Int) // Int => Int
  val res2 = sumLast(3) // 6
}
```

Каррирование

Каррирование

Каррирование или **карринг** (***currying***) - преобразование функции от многих аргументов в последовательность функций от одного аргумента.

Каррирование

Каррирование или **карринг** (***currying***) - преобразование функции от многих аргументов в последовательность функций от одного аргумента.

```
def addC(x: Int)(y: Int) = x + y

addC(1)(2) // 3
```

Каррирование

Процесс напоминает **матрешку**:
Мы передаем в функцию аргумент,
получаем новую функцию от одного аргумента и так далее

Последняя функция в цепочке вернет значение исходной на заданном наборе аргументов

Каррирование в Scala

```
object Currying {
  def add(x: Int, y: Int) = x + y
  def addC(x: Int)(y: Int) = x + y // каррированная функция

  add(1, 2) // 3
  addC(1)(2) // 3

}
```

Каррирование в Scala

```
object Currying {
  def add(x: Int, y: Int) = x + y
  def addC(x: Int)(y: Int) = x + y // каррированная функция

  add(1, 2) // 3
  addC(1)(2) // 3

  val addOne: Int => Int = addC(1)
  addOne(2) // 3
  addOne(0) // 1
}
```

Тип каррированной функции

```
object CurryingType {
  def add(x: Int, y: Int, z: Int) = x + y + z
  val addF = add _ // (Int, Int, Int) => Int




}
```

Тип каррированной функции

```
object CurryingType {
  def add(x: Int, y: Int, z: Int) = x + y + z
  val addF = add _ // (Int, Int, Int) => Int

  def addCurried(x: Int)(y: Int)(z: Int) = x + y + z
  val addCurriedF = addCurried _ // Int => (Int => (Int => Int))

}
```

Каррирование функций

В Scala можно из функции в некаррированной форме получить каррированную функцию. Для этого используется метод `.curried`

```
object Currying {
  def add(x: Int, y: Int) = x + y
  val addCurried = (add _).curried // Int => (Int => Int) = <function1>

  add(1,2) // 3
  addCurried(1)(2) // 3
}
```

Декаррирование функций

Из каррированной функции можно получить функцию в некаррированном виде методом `Function.uncurried`

```
object Currying {
  def multCurried(x: Int)(y: Int) = x * y
  val mult = Function.uncurried(multCurried _)

  multCurried(3)(4) // 12
  mult(3, 4) // 12
}
```

Каррирование - зачем?

- В некоторых чисто функциональных языках (Haskell) каррирование - единственный способ получить функцию от нескольких переменных

Каррирование - зачем?

- В некоторых чисто функциональных языках (Haskell) каррирование - единственный способ получить функцию от нескольких переменных
- В computer science каррирование дает возможность применять теорию функций одной переменной к функциям многих переменных

Каррирование - зачем?

В Scala:

- Вывод типа

```
def foldLeft[B](z: B)(op: (B, A) => B): B
List("").foldLeft(0)(_ + _.length)

def foldLeft[B](z: B, op: (B, A) => B): B
List("").foldLeft(0, (b: Int, a: String) => a + b.length)
List("").foldLeft[Int](0, _ + _.length)
```

Каррирование - зачем?

В Scala:

- Вывод типа
- Более удобный синтаксис

```
def foo(x: String)(y: String => Unit) = y(x)


foo("vasia") { name =>
  println(s"hi, $name!")
}
```

Каррирование - зачем?

В Scala:

- Вывод типа
- Более удобный синтаксис
- Неявные аргументы (implicit arguments)
- ...

Каррирование и частичное применение

- **Каррирование**
  Результат - последовательность функций от одного аргумента

- **Частичное применение**
  Результат - функция от одного или нескольких аргументов

http://geekabyte.blogspot.ru/2016/12/exploring-difference-between-currying.html

Значение аргумента по умолчанию

Метод может иметь дефолтное значение аргумента (как в Python)

Пример: метод `.copy` у `case class`

```
case class MyClass(x: Int, y: Int)


scala> val a = MyClass(1, 2)
a: MyClass = MyClass(1,2)


scala> val b = a.copy(y = 3)
b: MyClass = MyClass(1,3)
```

Значение аргумента по умолчанию

Метод может иметь дефолтное значение аргумента (как в Python)

```
scala> def foo(x: String = "scala", y: String = "rules") = x + y.toUpperCase
foo: (x: String, y: String)String


scala> foo("java", "magic") // Можно вызвать как обычный метод
res9: String = javaMAGIC
```

Значение аргумента по умолчанию

```
scala> foo()
res6: String = scalaRULES

scala> foo("Java")
res7: String = JavaRULES

scala> foo(y = ".wtf")
res8: String = scala.WTF
```

Значение аргумента по умолчанию

Можно комбинировать обычные аргументы и аргументы со значением по умолчанию

```
scala> def bar(x: String, y: String = "Martin") = x + y
bar: (x: String, y: String)String

scala> bar("hello")
res11: String = helloMartin
```

Значение аргумента по умолчанию

Eta expansion не сохраняет дефолтные значения

```
scala> def foo(x: String = "scala", y: String = "rules") = x + y.toUpperCase
foo: (x: String, y: String)String

scala> val fooFun = foo _
fooFun: (String, String) => String = <function2>
```

Переменное количество аргументов

Если метод принимает несколько параметров одного типа, можно использовать специальный синтаксис - Varargs (как … в Java, *args в Python):

```
object VarArgs extends App {
  def foo(args: String*) = args.mkString(",")
  foo("x") // x
  foo("x", "y") // x,y



}
```

Переменное количество аргументов

Если метод принимает несколько параметров одного типа, можно использовать специальный синтаксис (Varargs):

```
object VarArgs extends App {
  def foo(args: String*) = args.mkString(",")
  foo("x") // x
  foo("x", "y") // x,y

  val seq = Seq("x", "y", "z") // Для Seq в varargs используется : _*
  foo(seq: _*) // x,y,z
}
```

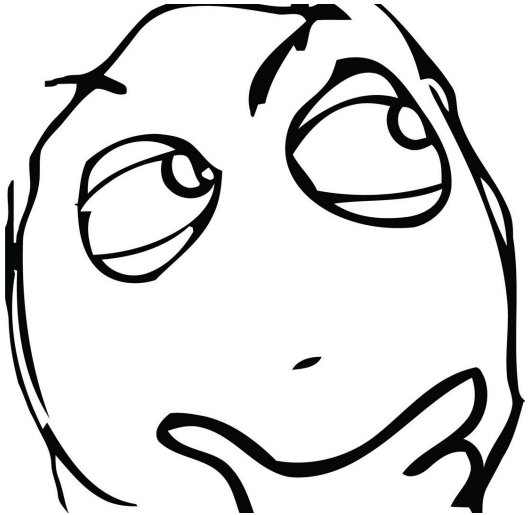
Переменное количество аргументов

Varargs-параметр:

- должен идти последним
- может быть только один
- не может иметь дефолтного значения

Переменное количество аргументов

В какую функцию превратится метод с varargs аргументом?



Переменное количество аргументов

В какую функцию превратится метод с varargs аргументом?

```
object VarArgs extends App {
  def foo(args: String*) = args.mkString(",")
  val fooFun = foo _ // Seq[String] = > String




}
```

Переменное количество аргументов

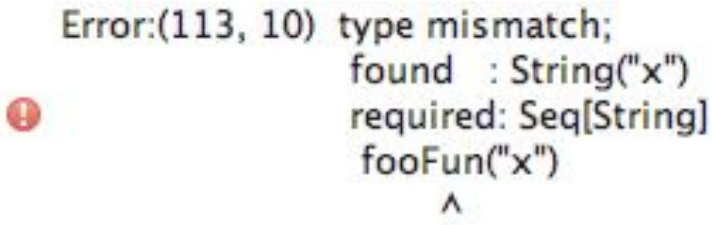
В какую функцию превратится метод с varargs аргументом?

```
object VarArgs extends App {
  def foo(args: String*) = args.mkString(",")
  val fooFun = foo _ // Seq[String] = > String

  fooFun("x")

}
```

Переменное количество аргументов

В какую функцию превратится метод с varargs аргументом?

```
object VarArgs extends App {
  def foo(args: String*) = args.mkString(",")
  val fooFun = foo _ // Seq[String] => String

  fooFun("x")

}
```



Переменное количество аргументов

В какую функцию превратится метод с varargs аргументом?

```
object VarArgs extends App {
  def foo(args: String*) = args.mkString(",")
  val fooFun = foo _ // Seq[String] = > String

  fooFun(Seq("x", "y", "z")) // x,y,z

}
```

Tupling

Tupling: для функции от нескольких аргументов метод `.tupled` создает функцию от одного аргумента - `Tuple`, содержащего аргументы исходной функции.

Tupling

Tupling: для функции от нескольких аргументов метод `.tupled` создает функцию от одного аргумента - `Tuple`, содержащего аргументы исходной функции.

```
object Tupling extends App {
  def foo(x: Int, y: Int) =  x + y
  foo(1, 2) // 3

  val fooTupled = (foo _).tupled // ((Int, Int)) => Int
  fooTupled((1, 2)) // 3
}
```

Untupling

Untupling: для функции от одного Tuple-аргумента метод `Function.untupled(f)` создает функцию от нескольких аргументов - элементов `Tuple`.

Untupling

Untupling: для функции от одного Tuple-аргумента метод `Function.untupled(f)` создает функцию от нескольких аргументов - элементов `Tuple`.

```
object Untupling extends App {
  def bar(x: (Int, Int)) = x._1 + x._2
  bar((1, 2)) // 3

  val barUntupled = Function.untupled(bar _) // (Int, Int) => Int
  barUntupled(1, 2) // 3
}
```

PartialFunction

PartialFunction

Функция работает для каждого аргумента определенного типа.

Частичная функция (PartialFunction) определена только для **некоторых** значений.

```
trait PartialFunction[-A, +B] extends (A) => B
```

PartialFunction

Функция работает для каждого аргумента определенного типа.

Частичная функция (PartialFunction) определена только для **некоторых** значений.

```
trait PartialFunction[-A, +B] extends (A) => B
```

Метод `.isDefinedAt(x)` проверяет, определена ли функция на аргументе x

```
def isDefinedAt(x: A): Boolean
```

PartialFunction - примеры

```
object PartialFunctions extends App {
  val foo: PartialFunction[Int, Int] = {
    case pos if pos > 0 => pos * pos
  }




}
```

PartialFunction - примеры

```
object PartialFunctions extends App {
  val foo: PartialFunction[Int, Int] = {
    case pos if pos > 0 => pos * pos
  }

  foo.isDefinedAt(4) // true
  foo(4) // 16



}
```

PartialFunction - примеры

```
object PartialFunctions extends App {
  val foo: PartialFunction[Int, Int] = {
    case pos if pos > 0 => pos * pos
  }

  foo.isDefinedAt(4) // true
  foo(4) // 16

  foo.isDefinedAt(-4) // false
  foo(-4)
}
```

PartialFunction - примеры

```
object PartialFunctions extends App {
  val foo: PartialFunction[Int, Int] = {
    case pos if pos > 0 => pos * pos
  }

  foo.isDefinedAt(4) // true
  foo(4) // 16

  foo.isDefinedAt(-4) // false
  foo(-4) // scala.MatchError: -4 (of class java.lang.Integer)
}
```

PartialFunction - примеры

примеры с map, foreach и тд

PartialFunction: orElse

У `PartialFunction` значение аргумента, не попавшее в область определения функции, можно передать “по цепочке” следующей функции `that` с помощью метода `.orElse(that)`

PartialFunction: orElse

У `PartialFunction` значение аргумента, не попавшее в область определения функции, можно передать “по цепочке” следующей функции `that` с помощью метода `.orElse(that)`

Результат метода `.orElse(that)` - новая функция:

- Если к аргументу можно применить исходную функцию, применяется она
- В противном случае при возможности выполняется следующая функция

PartialFunction: orElse

```
object PartialFunctions extends App {
  val foo: PartialFunction[Int, Int] = {
    case pos if pos > 0 => pos * pos
  }
  val bar: PartialFunction[Int, Int] = {
    case neg if neg < 0 => neg + neg
  }

  val chained = foo.orElse(bar) // PartialFunction[Int, Int]
```

PartialFunction: orElse

```
  chained.isDefinedAt( 4) // true
  chained(4) // 16
  chained.isDefinedAt(- 4) // true
  chained(-4) // -8



}
```

PartialFunction: orElse

```
chained.isDefinedAt(4) // true
chained(4) // 16
chained.isDefinedAt(-4) // true
chained(-4) // -8

chained.isDefinedAt(0) // false
chained(0) // scala.MatchError: 0 (of class java.lang.Integer)
```

PartialFunction: try/catch

Блок `catch` принимает `PartialFunction[Throwable, Any]`

```
def handleIae: PartialFunction[Throwable, Any] = {
  case iae: IllegalArgumentException => System.out.println("Got iae")
}

def handleNpe: PartialFunction[Throwable, Any] = {
  case iae: NullPointerException => System.out.println("Got npe")
}

try {
  // do something
} catch handleIae.orElse(handleNpe)
```

scala.util.control.Exception

`scala.util.control.Exception` содержит методы для обработки ошибок в функциональном стиле, альтернатива `scala.util.Try`

```
import scala.util.control.Exception._
```

scala.util.control.Exception

```
import scala.util.control.Exception._


def assertPos(x: Int): Int =
  if (x >= 0) x else throw new IllegalArgumentException(s"$x is negative")
```

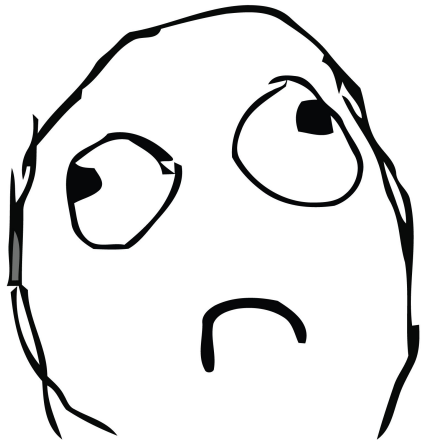
scala.util.control.Exception

```
import scala.util.control.Exception._

def assertPos(x: Int): Int =
  if (x >= 0) x else throw new IllegalArgumentException(s"$x is negative")

scala> catching(classOf[IllegalArgumentException]) opt assertPos(1)
res7: Option[Int] = Some(1)

scala> catching(classOf[IllegalArgumentException]) opt assertPos(-1)
res8: Option[Int] = None
```

scala.util.control.Exception

А если не поймали?

```
scala> catching(classOf[NullPointerException]) opt assertPos(-1)
java.lang.IllegalArgumentException: -1 is negative
```



scala.util.Exception

`catching` может принимать `PartialFunction[Throwable, T]`

```
def handleIae: PartialFunction[Throwable, Int] = {
  case iae: IllegalArgumentException =>
    System.out.println("Got iae")
    0
}

scala> catching(handleIae)(assertPos(-1))
Got iae
res35: Int = 0
```

scala.util.control.Exception

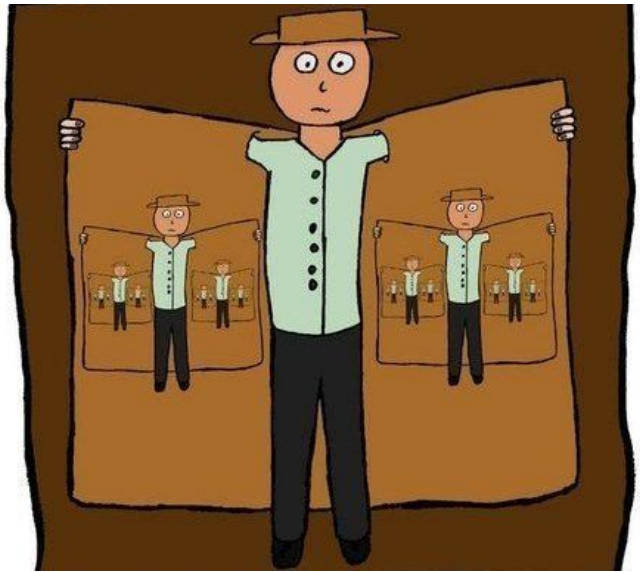
И многое другое:

- Оборачивание в `Try, Either, Option`
- Дефолтное значение
- `orElse`
- ...

Хвостовая рекурсия

Рекурсия

Рекурсия — вызов функции из неё же самой, непосредственно или через другие функции



Рекурсия

Рекурсия — вызов функции из неё же самой, непосредственно или через другие функции

```
object Recursion {
  def factorial(n: Int): Int = {
    if (n == 0) 1 else n * factorial(n - 1)
  }



}
```

Рекурсия

Рекурсия — вызов функции из неё же самой, непосредственно или через другие функции

```
object Recursion {
  def factorial(n: Int): Int = {
    if (n == 0) 1 else n * factorial(n - 1)
  }

  factorial(5) // 120
}
```

Хвостовая рекурсия

Хвостовая рекурсия — вид рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции.

Хвостовая рекурсия

```
object Recursion extends App {
  def factorialTailRec(n: Long): Long = {
    @tailrec
    def factorialAccum(acc: Long, n: Long): Long = {
      if (n == 0) acc else factorialAccum(n * acc, n - 1)
    }
    factorialAccum(1, n)
  }
}
```

Хвостовая рекурсия

Не любая рекурсия может быть представлена в виде хвостовой рекурсии

Хвостовая рекурсия - зачем?

- Нет необходимости хранить стек состояний
- Хвостовая рекурсия может быть заменена на итерацию
- Не любая рекурсия может быть приведена к виду хвостовой рекурсии

Хвостовая рекурсия

Scala - компилятор автоматически оптимизирует функции с хвостовой рекурсией.

Аннотация `@tailrec` указывает компилятору проверить, является ли метод хвостовой рекурсией.

Если метод не является хвостовой рекурсией, происходит ошибка компиляции.

Спасибо