Scala
для профессионалов

Павел Павлов

О себе

Занимаюсь системным программированием

- компиляторы, среды исполнения
- Excelsior JET - JVM with AOT compiler
- код на орбите (ГЛОНАСС)

Преподаю

- научное руководство студентами (НГУ)
- подготовка юниоров в компании

Участник Scala community

- Scala core committer (stdlib, compiler)
- Организовал ScalaNsk

Тема: Scala для профессионалов

Это не будет

- ещё одно “введение в Скалу”
- “мои любимые 2-3 фишечки в Scala”
- доклад для юниоров

Я не буду

- приводить код hello world, чисел Фибоначчи, ...
- вдаваться в несущественные детали
- гугл в помощь

Я расскажу

- чем является Scala
- зачем, кому и когда она нужна

Три аспекта

Идеи

- философия, концепции, вопросы дизайна языка

Инструменты

- языковые фичи, паттерны, библиотеки, тулы

Использование

- проникновение в индустрию, success stories, мнения пользователей, собственный опыт

Идеи

Кто такой профессионал?

*-программисты

- C++-программист
- Java-программист
- Ruby-программист
- html/js-программист
- 1C-программист

Звания и титулы

- Senior JavaEE Developer
- System Architect

Набор заклинаний

- XHTML, DHTML, XML, SQL, T-SQL, PL/SQL, COM, CSS, PHP, AJAX, J2EE, JBoss, Jira, FogBugz, IIS

Кто такой профессионал?

Профессиональный программист -
это инженер

Язык программирования -
главный инструмент программиста

Выбор языка

Объективные/инженерные факторы

- Пригодность для предметной области
- Эксплуатационные характеристики:

  производительность, масштабируемость, maturity, переносимость, resource requirements, ...
- Производственные характеристики:

  распространённость, скорость разработки, развитость экосистемы, требования к квалификации разработчиков, learning curve, ...
- Взаимодействие с 3rd-party/legacy кодом

Субъективные & внешние факторы

- Корпоративные стандарты, инерция, пиар
- Личные предпочтения

Java

Ruby

C#

Erlang

JavaScript

Clojure

Python

C

Haskell

PHP

C++

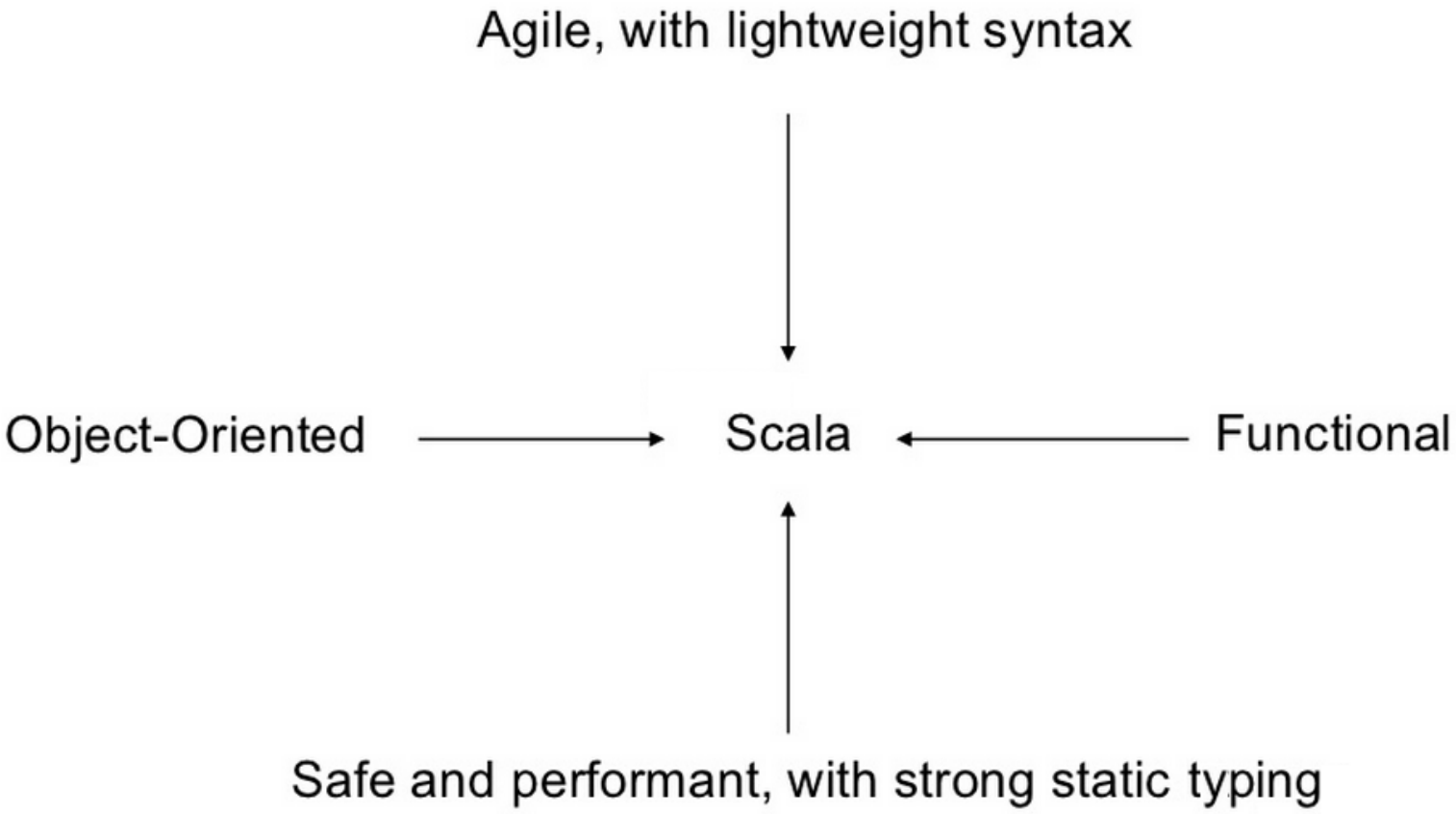
Lua

Go

Groovy

One ring to rule them all.

Scala is a Unifier



Язык Scala: цели

- Язык общего назначения
- Альтернатива/замена Java
- Внедрение FP в mainstream
- Преодоление "застоя" в OOP

Целевая аудитория:

1. Java-программисты (народные массы)

   Лучший язык, FP
2. Ruby/Python(/JS)-программисты

   Статическая типизация, производительность
3. Функциональщики

   Использование FP в mainstream

```
def scala = oop + fp
```

Scala = OOP + FP

Как скрестить ужа с ежом?

- дешёвая китайская имитация

  lambdas in C++, Java 8, ...
- мухи отдельно, котлеты отдельно
- Scala way: алхимический сплав
  - больше, чем сумма слагаемых
  - требуется переосмыслить основы

Технически, Scala = OO kernel + syntactic sugar

  - кодируем функции объектами

Синтаксис: основы

```
val x: Int = 42
val ys = List(1, 2, 3)              // y: List[Int]
var z = "hello"                     // z: String

def max(a: Int, b: Int): Int = if (a > b) a else b
def inc(x: Int) = x + 1
val inc = { x: Int => x + 1 }       // inc: Int => Int

ys map inc                          // List(2, 3, 4)
ys map { x => x + 1 }
ys map { _ + 1 }
ys filter { _ % 2 != 0 }            // List(1, 3)
ys reduce max                       // max(1, max(2, 3))
ys reduce { (x, y) => x + y }       // 1 + 2 + 3
```

Классы и объекты

```
abstract class Animal {
  def name: String
}
class Person(firstName: String, lastName: String)
    extends Animal {
  val name = firstName + " " + lastName
}
class Employee(firstName: String, lastName: String,
               val age: Int, var salary: Double)
    extends Person(firstName, lastName)

object Main extends App {
  val p = new Employee("John", "Doe", 20, 300.0)
  println(p.name + " is + p.age + " years old")
}
```

Methods and operators

```
List(1, 2, 3).map(inc)    // List(2, 3, 4)
List(1, 2, 3) map inc     // <- операторный синтаксис

abstract class Set[T] {
  def contains(x: T): Boolean
  def - (x: T): Set[T]
  def !@#%^&*-+ : Int     // <- пожалуй, так делать не стоит
}

def foo(s: Set[Int]) {
  if (s contains 42) println(s - 42)
}

1 + 3*5     =>  1.+(3.*(5))
```

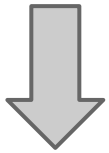
Вызов и индексация

```
val inverse = { x: Int => 1.0 / x }
inverse(1)                  // =>  inverse.apply(1)
val a = Array(0.33, 0.5)    // =>  Array.apply(0.33, 0.5)
a(1) /*0.5*/                // =>  a.apply(1)
a(0) = 1.0                  // =>  a.update(0, 1.0)

def foo(f: Int => Double) {
  println(f(1))             // =>  println(f.apply(1))
}
foo(inverse)      // 1.0
foo(a)            // 0.5

Array[T] is a function (Int => T)!
```

Функции как объекты

Функциональные типы:

```
(A => B)      // Function1[A, B]
trait Function1[A, B] {
  def apply(x: A): B
}

val inc = { x: Int => x + 1 }
```



```
val inc = new Function1[Int, Int] {
  def apply(x: Int): Int = { x + 1 }
}

inc(4)        // inc.apply(4)
```

Scala: OOP done right

Есть много устоявшихся мифов об ООП, которые не соответствуют действительности

Они отравляют нам жизнь: заставляют писать лишний код, плодить кривые дизайны, делать больше работы для достижения худшего результата

Я разберу четыре из них

Мифы ООП: 1

Getters & setters provide encapsulation

O, RLY?

На самом деле:

Они просто не нужны

Scala: uniform access principle

Uniform Access Principle

Все услуги, предлагаемые модулем должны быть доступны через единую нотацию, которая не раскрывает, реализованы ли они посредством хранения либо вычисления.

Bertrand Meyer

UAP: fields, methods and properties

```
class Foo(val r1: Int) { // <- auto-generated getter
  var rw1: Int  // <- auto-generated getter & setter

  def r2: Int = ???
  private[this] var _field = 0
  def rw2: Int = _field
  def rw2_=(x: Int) { _field = x }
}

val v: Foo = ...
println(v.r1 + v.r2 + v.rw1 + v.rw2)
v.rw1 = 42
v.rw2 = 43  // v.rw2_=(43)
```

Мифы ООП: 2

Object = identity + state + behavior

O, RLY?

java.lang.String: достаточно ООПен?

- identity is misleading & (almost) useless

```
if (s1 == s2) ...
```

- no *mutable* state => state = data
- behavior is just a function(data)

В чём дело?

Мифы ООП: 2

java.lang.String is *immutable* object

FP: Algebraic types (ADT)

Immutable objects = ADT + encapsulation

Scala:

- Favor immutability, support for ADTs
- Behavior of '==' is defined by the class

Object = encapsulated code & data

identity & state are optional

Мифы ООП: 3

Базовый класс не должен зависеть от своих наследников

O RLY? Visitor pattern? Double dispatching?

На самом деле:

- Есть закрытые и открытые иерархии
- Открытые иерархии - классический ООП
- Закрытые - знаем всех своих наследников
- Две принципиально разные модели, выражать одну через другую - неестественно
- В мире ФП есть подходящие средства: алгебраические типы (ADT) & pattern matching

Case classes

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

```
class Person(val name: String, val age: Int)
    extends Serializable {
  override def equals(other: AnyRef) = ...
  override def hashCode = ...
  override def toString = ...
  def copy(name: String = this.name, age: Int = this.age) = ...
}
object Person extends (String, Int) => Person {
  def apply(name: String, age: Int) = new Person
  def unapply(p: Person): Option((String, Int)) =
    Some((p.name, p.age))
}
```

Case classes as ADT

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

val p = Person("John", 20)
println(p)                      // Person(John,20)
val q = p.copy(name="Sam")
q.copy(age = q.age+1) == Person("Sam", 21)   // true

def foo(x: Any) = x match {
  case Person(name, age) => s"$name of $age years"
  case _ => x.toString
}

foo(p)          // "John of 20 years"
foo(123)        // "123"
```

ADT in action

```
sealed class Expr
case class Var(name: String) extends Expr
case class Num(value: Int) extends Expr
case class Neg(arg: Expr) extends Expr
case class Add(arg1: Expr, arg2: Expr) extends Expr

def optimize(expr: Expr): Expr = expr match {
  case Neg(Neg(x))                => optimize(x)
  case Add(x, Num(0))             => optimize(x)
  case Neg(Num(x))                => Num(-x)
  case Add(x, Neg(y)) if x == y => Num(0)
  case Add(Num(x), Num(y))        => Num(x + y)
  case Neg(x)                     => Neg(optimize(x))
  case Add(x, y)                  => Add(optimize(x), optimize(y))
  case _                          => expr
}
```

Мифы ООП: 4

Multiple inheritance is a sin

Классический миф:

- все об этом “знают”
- но никто не помнит, откуда это знание взялось
- все вроде бы в курсе, почему
- мало кто в точности понимает, в чём именно дело

Мифы ООП: 4

История мифа:

- MI, как оно было реализовано в C++ (~20 лет назад) создавало серьёзные проблемы
- В связи с этим в Java MI было сильно урезано
- Java появилась в 1995 году
- PR-машина (“Java лучше C++”) породила миф
- Ограничения MI в Java привели к появлению workaround’ов и “костыльных” паттернов
- В итоге в Java 8 таки появилось MI поведения
- Сейчас нам говорят, что это ok
- 20 лет нам врали?

Итоги: миф не развенчан, старые проблемы не решены, костыли в языке (default, различие классов и интерфейсов)

Мифы ООП: 4

Multiple inheritance is a sin

Что на самом деле?

Проблемы MI

A. Diamond problem
- shared state vs. duplicated state
- object initialization
- correct super calls

B. Sibling problem
- name clash
- mixing APIs

Корень проблемы: мы недостаточно ясно понимаем, что такое MI (как отношение и как действие) и что при этом происходит

Последние 20 лет наука на месте не стояла!

Multiple inheritance done right

- trait (mixin) - почти то же самое, что и class
- MI (действие) - это сборка класса из частей
  - aka mixin composition
- no state duplication
- порядок важен!
  - “`extends A with B`” != “`extends B with A`”
- super определяется динамически
  - correct super calls
  - stackable modifications
- no name clash - берётся последний

Traits: пример

```
trait Ordered[A] {
  def compare(that: A): Int

  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
}

class Money extends SomeTrait with Ordered[Money] {
  def compare(that: Money) = ...
}
```

Модульность: Java

- Как разделить систему на части?
- Как управлять зависимостями между модулями?
- Как абстрагироваться от деталей реализации?
- Как собрать систему из частей?

Use your favorite DI framework!

Что нам даёт DI framework:

- инкапсуляция, абстракция, композиция объектов, определение интерфейсов, сокрытие реализации, полиморфизм

Это же классическая ОО задача!

Java просто не справляется

Модульность: Scala

ООП может использоваться для модульности

- Модуль - это trait
- Композиция модулей - это MI
- Вся система - это объект
- Зависимости (импорт модулей) - через self types
- Абстракция: abstract types (в доп. к методам и полям)

Преимущества:

- Всё контролируется статически
- Нет boilerplate, дублирования кода
- Не нужен внешний DI framework
- Можно полностью избавиться от глобалов

Ключевые слова: cake pattern, self type, abstract type

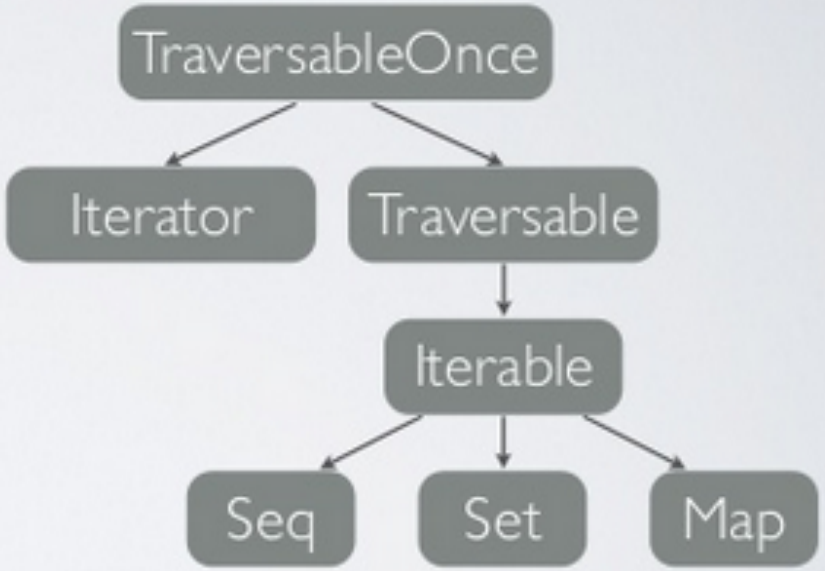
Коллекции & итерация

Коллекции - первая по важности вещь после собственно языковых фич

- Они вездесущи
- Дизайн стандартной библиотеки коллекций влияет на каждую программу, причём в значительной степени
- Они формируют словарь, с помощью которого программист выражает свои мысли

Scala Collections

- Seq
  - IndexedSeq, Buffer, …
- Set
  - HashSet, BitSet, …
- Map
  - HashMap, TreeMap, …



Scala Collections: API

- functional-style
- collect, count, exists, filter, find, flatMap, fold, forall, foreach, groupBy, map, max/min, partition, reduce, splitAt, take, to, ...
- примеры:

```
val people: Seq[Person] = ...
val (minors, adults) = people partition (_.age < 18)
```

- параллельные коллекции: .par, tuning (thread pools etc.)

Итерация: внешняя и внутренняя

for comprehension

- обобщённый цикл for
  - с поддержкой фильтров и pattern matching
- преобразователь потоков данных
- язык запросов
- монадический комбинатор
- всего лишь синтаксический сахар

for loop: basic form

```
val files: Seq[File] = ...

for (file <- files) {
  println(file.getName)
}

files foreach { file =>
  println(file.getName)
}

val names = for (file <- files) yield file.getName
val names = files map { _.getName }
```

for loop: filters & nested loops

```
def content(f: File): Seq[String] = ???

for {
  file <- files
  if !file.getName.startsWith(".")
  line <- content(file)
  if line.nonEmpty
} println(file + ": " + line)

files withFilter (!_.getName.startsWith(".")) foreach { file =>
  content(file) withFilter (_.nonEmpty) foreach { line =>
    println(file + ": " + line)
  }
}
```

for loop: filters & nested loops

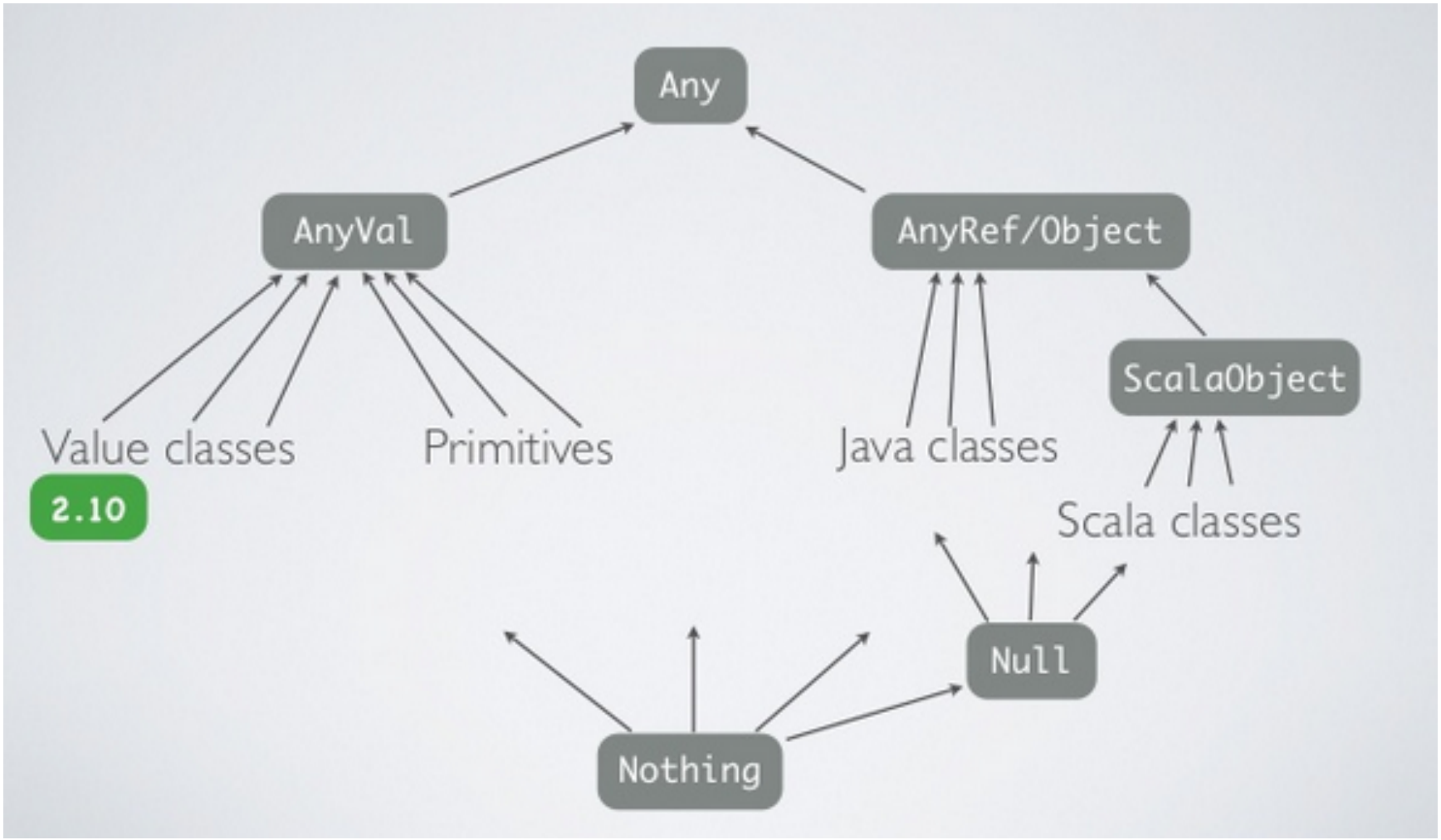
```
def content(f: File): Seq[String] = ???

val lines = for {
  file <- files
  if !file.getName.startsWith(".")
  line <- content(file)
  if line.nonEmpty
} yield file + ": " + line

files withFilter (!_.getName.startsWith(".")) flatMap { file =>
  content(file) withFilter (_.nonEmpty) map { line =>
    file + ": " + line
  }
}
```

Инструменты

Hello, REPL!

```
scala> val repl = Map('R' -> "Read", 'E' -> "Eval",
     |                'P' -> "Print", 'L' -> "Loop")
repl: immutable.Map[Char,String] = Map(...)

scala> for ((k, v) <- repl) println(s"$k is for $v")

R is for Read
E is for Eval
P is for Print
L is for Loop
```

Types hierarchy



Типы

- Основные типы: primitives, Unit, tuples, functions, Option
- Составные типы:

```
A with B with ... { refinement }
```

- Отношение соответствия типов: `A <: B`
- Тип определяет интерфейс

Типы

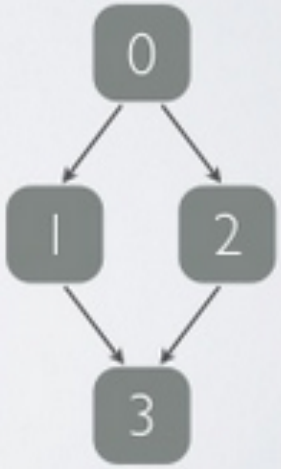
- value & reference types
- top & bottom types
- compound types
- параметризация (generics)
  - variance: co- & contra-
  - upper & lower bounds
- higher-kinded types
- abstract types
- self types
- dependent types
- structural types (typesafe duck typing)

Implicits

- Extension methods
- Automatic adaptors
- DSLs
- Typeclass pattern (C++ concepts)

Internal DSLs

```
class DominatorsSuite extends FunSuite with ShouldMatchers
                         with GraphBuilderDSL {
  test("diamond") {
    calcDominatorsOver(0 -> (1 || 2) -> 3)
    idom(1) should be (0)
    idom(2) should be (0)
    idom(3) should be (0)
  }
}
```



Macros

```
def assert(cond: Boolean, msg: Any) = macro Asserts.assertImpl

object Asserts {
  def raise(msg: Any) = throw new AssertionError(msg)
  def assertImpl(c: Context)
      (cond: c.Expr[Boolean], msg: c.Expr[Any]): c.Expr[Unit] =
    if (assertionsEnabled)
      reify { if (!cond.splice) raise(msg.splice) }
    else
      reify { () }
}

assert(c, "msg")     =>   if(!c) raise("msg")
```

ARM

automatic resource management

```
withFile ("~/.bashrc") { f =>
  for (l <- f.lines) {
    if ("#".r.findFirstIn(l) != None)
      println(l)
  }
}
```

Parallelism & concurrency

```
val people: Seq[Person] = ...
val (minors, adults) = people partition (_.age < 18)
```

Parallelism & concurrency

```
val people: Seq[Person] = ...
val (minors, adults) = people.par partition (_.age < 18)
```

Parallelism & concurrency

```
val people: Seq[Person] = ...
val (minors, adults) = people.par partition (_.age < 18)

actor {
  receive {
    case people: Seq[Person] =>
      val (minors, adults) = people partition (_.age < 18)
      School ! minors
      Work ! adults
  }
}
```

Futures

Future

- Хранилище для значения, которое будет получено в будущем
- Получение значение может быть выполнено асинхронно и не блокировать программу
- Значение может не быть получено вовсе

Futures

```
val f: Future[List[String]] = future {
  session.getRecentPosts
}

f onSuccess {
  case posts =>
    for (p <- posts) println(p)
}

f onFailure {
  case t =>
    println("An error has occurred: " + t.getMessage)
}
```

Combining futures

```
val usdQuote = future { connection.getCurrentValue(USD) }
val chfQuote = future { connection.getCurrentValue(CHF) }

val purchase = for {
  usd <- usdQuote
  chf <- chfQuote
  if isProfitable(usd, chf)
} yield connection.buy(amount, chf)

purchase onSuccess {
  case _ => println(s"Purchased $amount CHF")
}
```

Akka

- Toolkit & runtime for building concurrent, distributed & fault tolerant applications
- Actors
- Remoting: location transparent
- Supervision & monitoring
- Software Transactional Memory
- Dataflow concurrency

Tools

IDEs: IDEA, Eclipse, NetBeans

+ Emacs, vim, Sublime, TextMate

Building: SBT

but also ant, Maven, etc...

Testing: ScalaTest, Specs, ScalaCheck

but also JUnit, EasyMock, Mockito, etc...

Web frameworks: Lift, Play!

but also Struts, Spring MVC, etc...

+ all that Java stuff

Libraries & frameworks

DB query & access: Slick (LINQ-like)

UI: ScalaFX

Text processing: parser combinators (stdlib)

GPGPU: ScalaCL

...and many others

Использование

Что говорят умные люди?

“I can honestly say if someone had shown me the Programming Scala book by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy.“

James Strachan, creator of Groovy

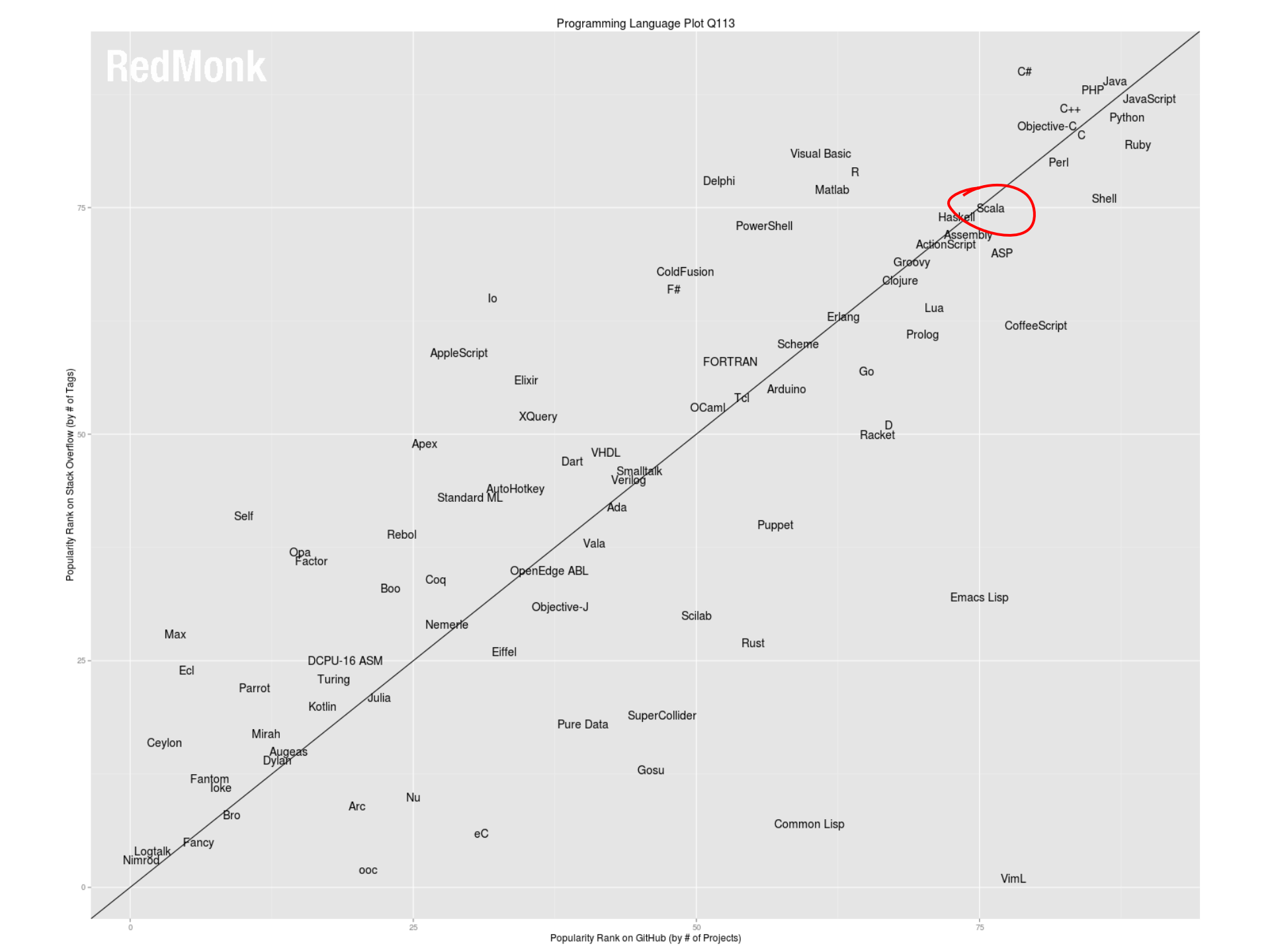
“If I were to pick a language to use today other than Java, it would be Scala.”

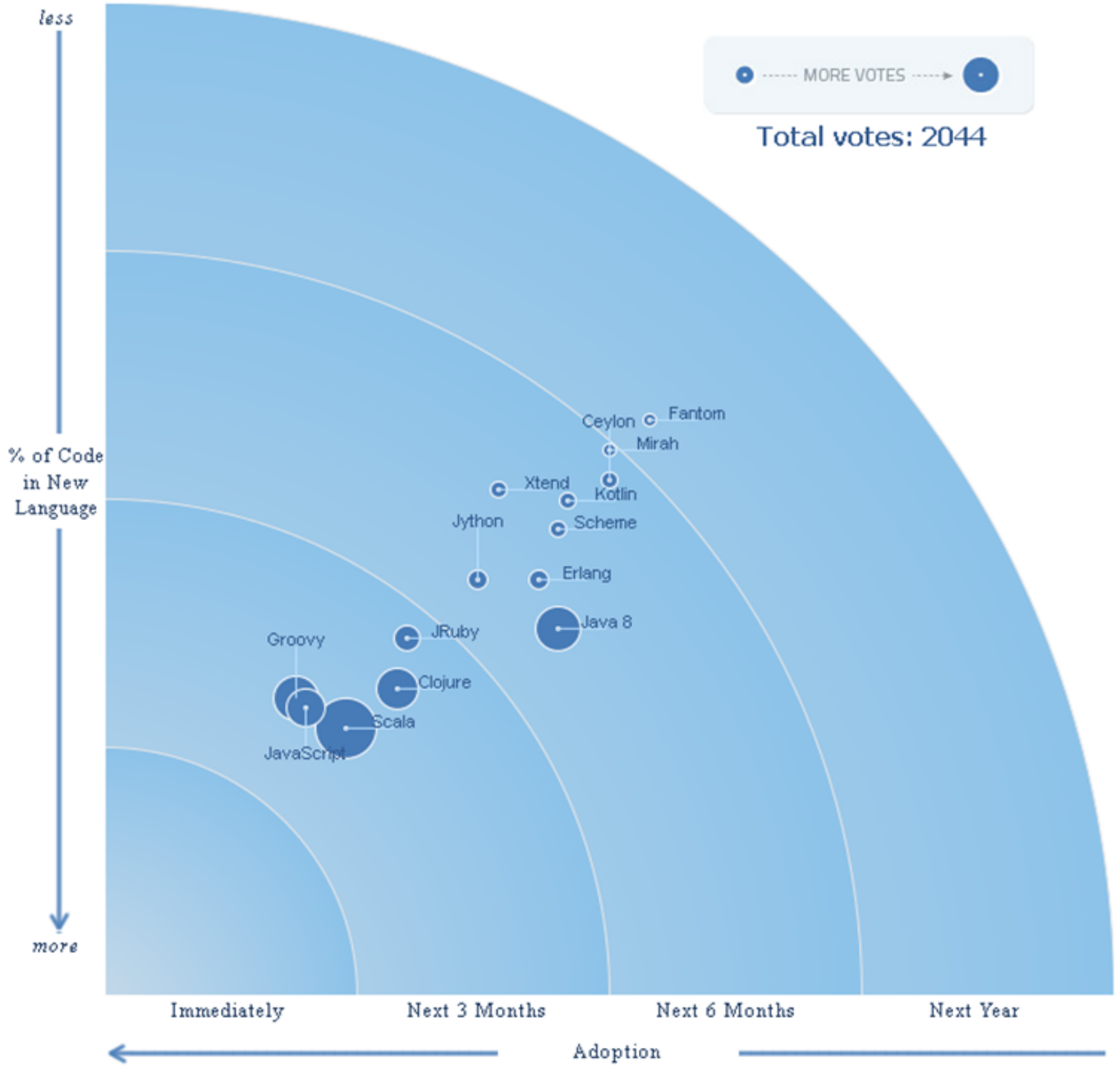
James Gosling

Популярность Scala

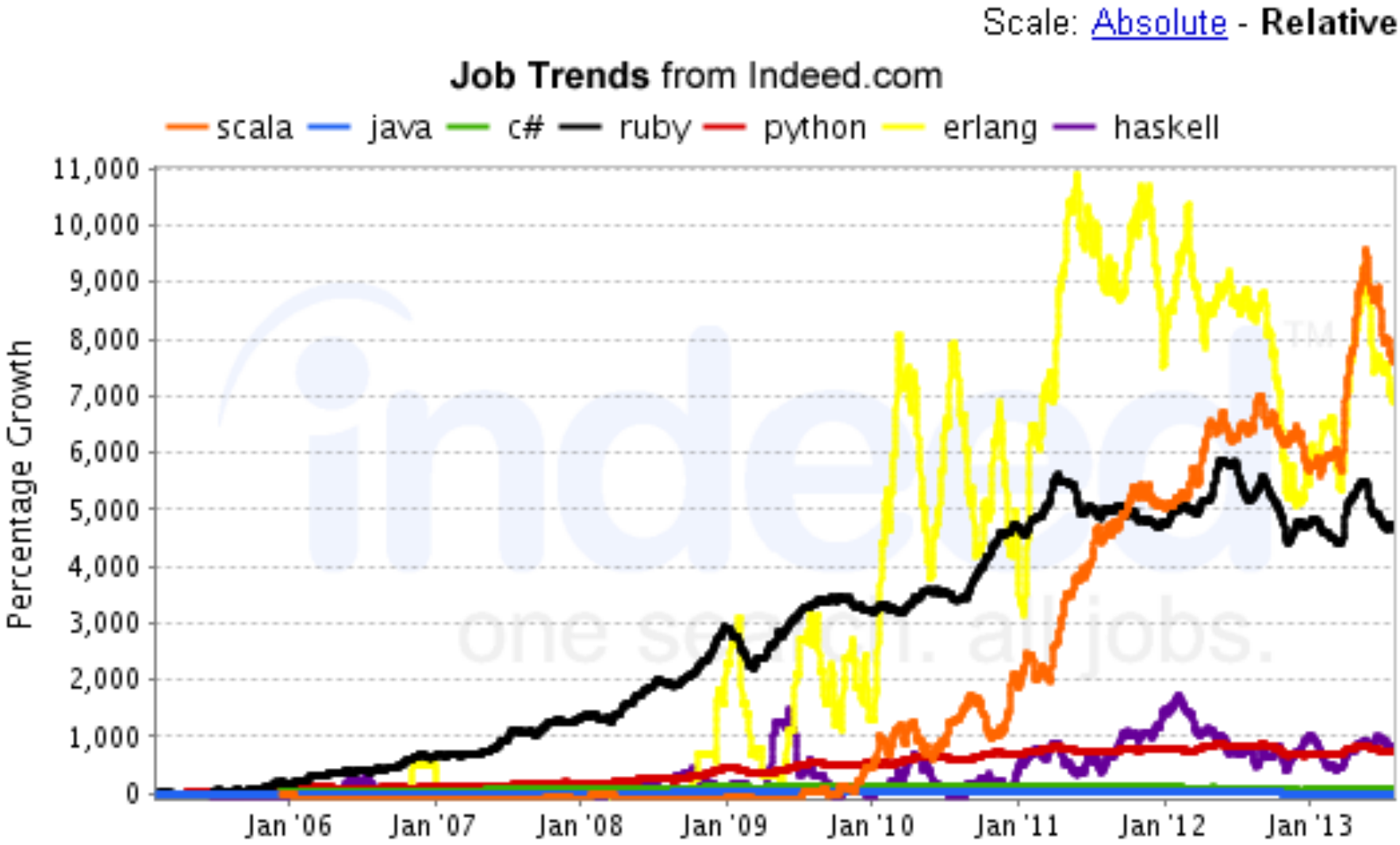
- 11 место - RedMonk Programming Language Rankings (StackOverflow, GitHub)
- 36 место - TIOBE index

  (поисковые запросы)

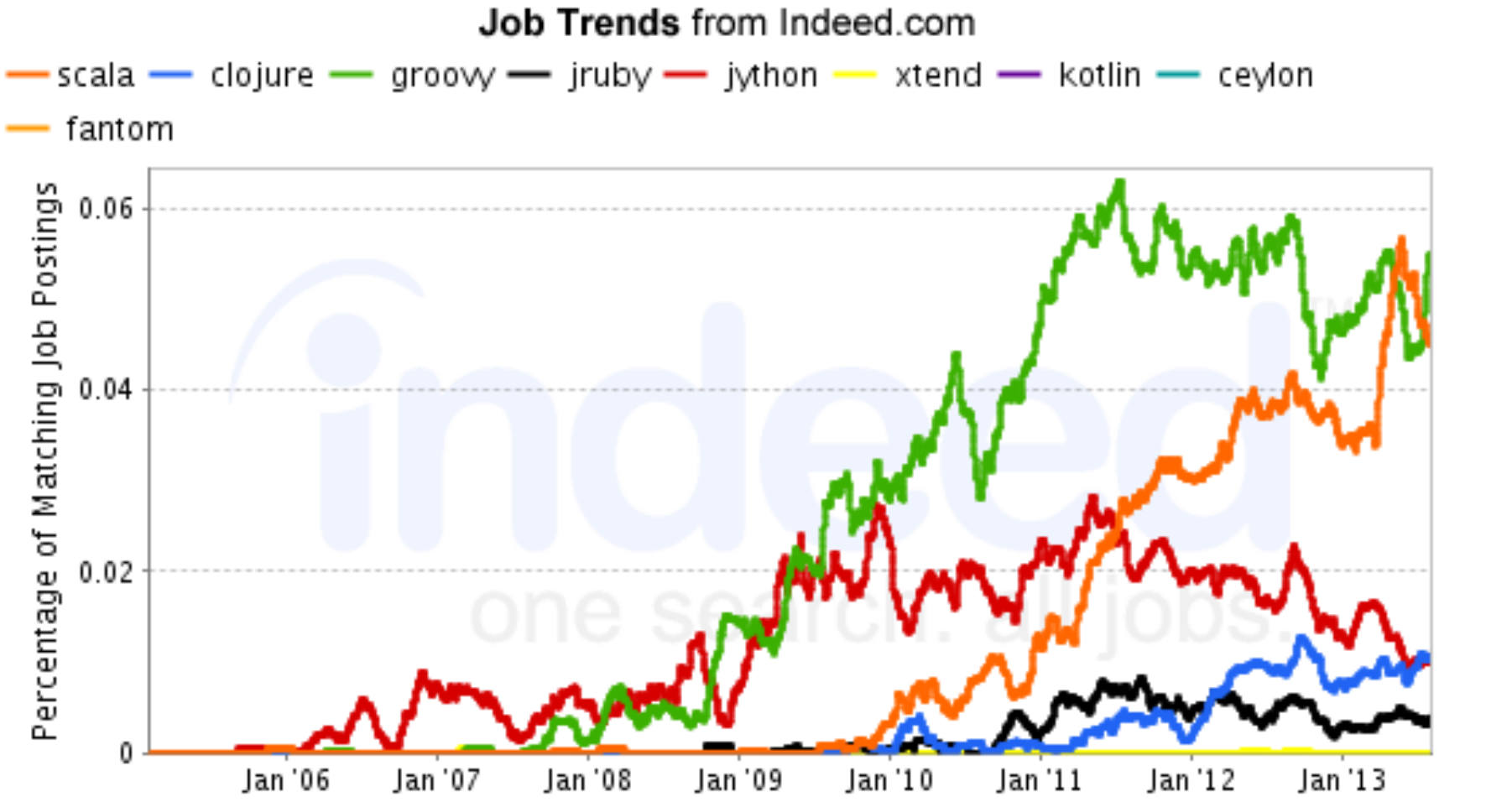

Programming Language Plot Q113
RedMonk
C#
Java
PHP
JavaScript
C++
Python
Objective-C
C
Ruby
Visual Basic
Perl
R
Delphi
Matlab
Shell
Scala
Haskell
PowerShell
Assembly
ActionScript
ASP
Groovy
ColdFusion
Clojure
F#
Io
Lua
Erlang
CoffeeScript
Prolog
Scheme
AppleScript
FORTRAN
Go
Elixir
Arduino
Tcl
OCaml
XQuery
D
Racket
Apex
VHDL
Dart
Smalltalk
Verilog
AutoHotkey
Standard ML
Ada
Self
Puppet
Rebol
Vala
Opa
Factor
OpenEdge ABL
Coq
Boo
Emacs Lisp
Objective-J
Scilab
Nemerle
Max
Rust
Eiffel
DCPU-16 ASM
Ecl
Turing
Parrot
Julia
Kotlin
SuperCollider
Pure Data
Mirah
Ceylon
Augeas
Dylan
Gosu
Fantom
Ioke
Nu
Arc
Bro
Common Lisp
eC
Fancy
Logtalk
Nimrod
ooc
VimL
Popularity Rank on Stack Overflow (by # of Tags)
Popularity Rank on GitHub (by # of Projects)
0
25
50
75



http://www.infoq.com/research/next-jvm-language


Scale: Absolute - Relative
Job Trends from Indeed.com
scala
java
c#
ruby
python
erlang
haskell
Percentage Growth
11,000
10,000
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
Jan '06
Jan '07
Jan '08
Jan '09
Jan '10
Jan '11
Jan '12
Jan '13

Scale: **Absolute** - Relative



Scala Today


IBM


SONY
make.believe


imageworks


twitter


CLOUD
FOUNDRY


LinkedIn


Novell


outside.in


xerox


EDF


NASA


SIEMENS


foursquare


EXCELSIOR


UBS


TOMTOM


amazon


Office
DEPOT
Taking Care
of Business


OPOWER
ENERGY EFFICIENCY DELIVERED


guardian.co.uk


Bank of America


yammer


CREDIT SUISSE

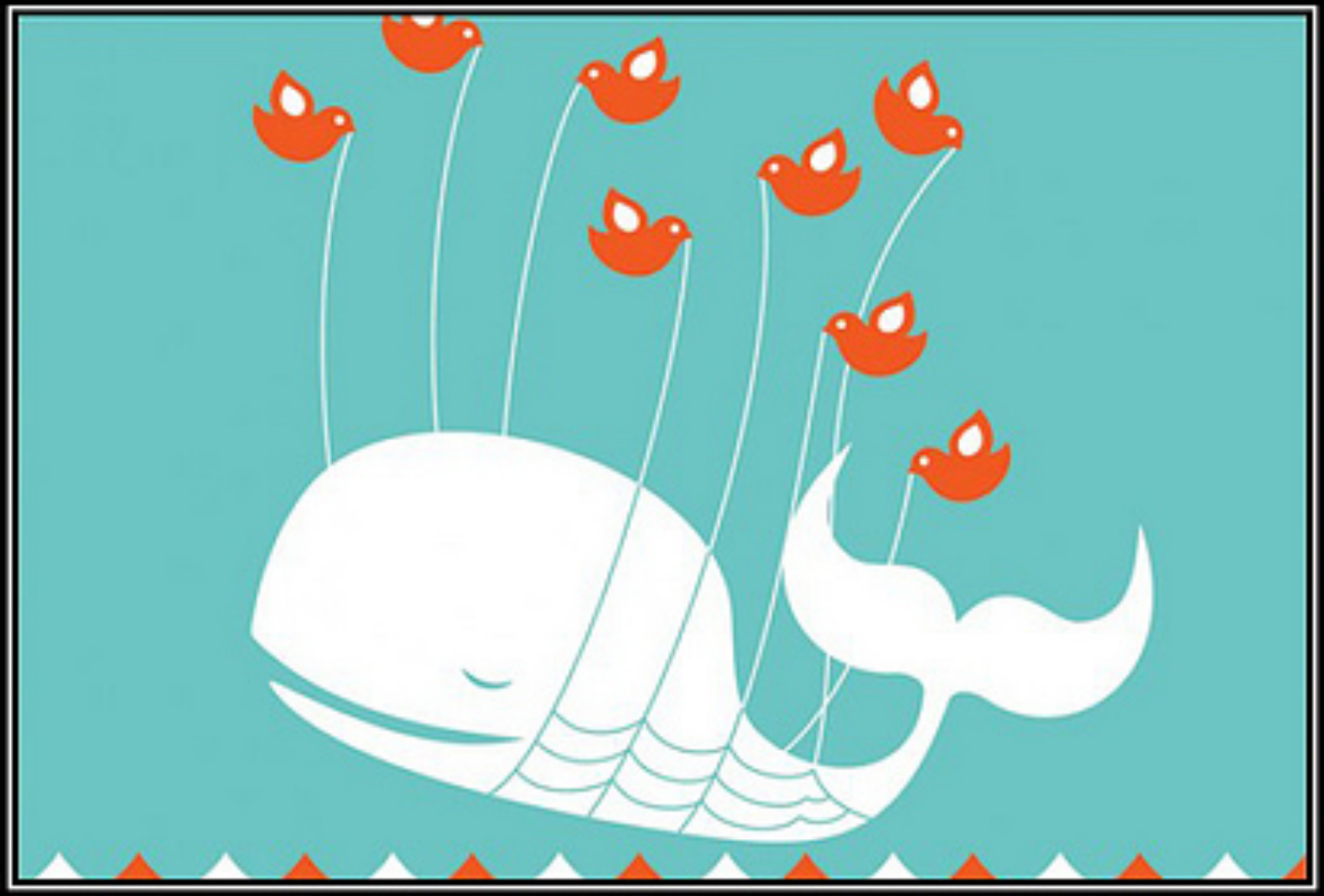

Thatcham


Autodesk


NETFLIX

Success story: Twitter

- 2006: начало - всё на Ruby on Rails
- 2007-2008: взрывной рост популярности



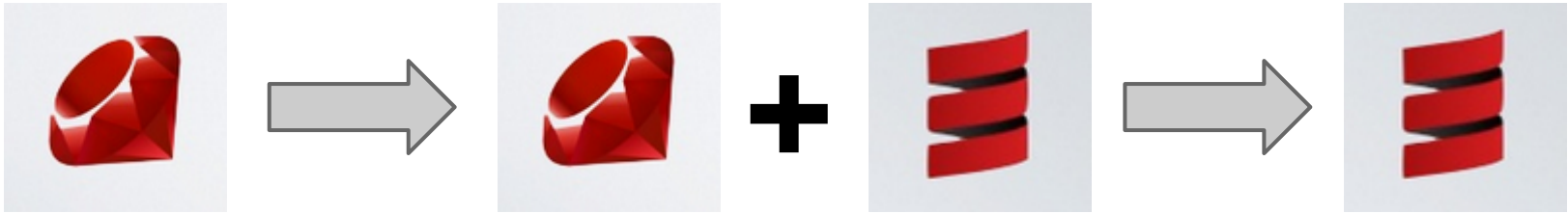
FAIL WHALE

Twitter: Failure is an option. At least once a day, or whenever you need it.

Success story: Twitter

- 2006: начало - всё на Ruby on Rails
- 2007-2008: взрывной рост популярности

Решение:



Excelsior: наш опыт

- Оптимизирующий компилятор
- Новое ядро, implemented from scratch
- R&D "по-взрослому"

EXCELSIOR

Excelsior: наш опыт

- Оптимизирующий компилятор
- Новое ядро, implemented from scratch
- R&D "по-взрослому"
- Команда: ≤5 человек

EXCELSIOR

Excelsior: наш опыт

- Оптимизирующий компилятор
- Новое ядро, implemented from scratch
- R&D "по-взрослому"
- Команда: ≤5 человек
  - 4 из них не знали Скалу

EXCELSIOR

Excelsior: наш опыт

- Оптимизирующий компилятор
- Новое ядро, implemented from scratch
- R&D "по-взрослому"
- Команда: ≤5 человек
  - 4 из них не знали Скалу
- Время: 1.5 года * 4 человек
  - всё работает, релиз был в апреле 2013

EXCELSIOR

Excelsior: наш опыт

- Оптимизирующий компилятор
- Новое ядро, implemented from scratch
- R&D "по-взрослому"
- Команда: ≤5 человек
  - 4 из них не знали Скалу
- Время: 1.5 года * 4 человек
  - всё работает, релиз был в апреле 2013
  - это мировой рекорд

EXCELSIOR

Novosibirsk Scala Enthusiasts

www.meetup.com/ScalaNsk

www.twitter.com/ScalaNsk

EXCELSIOR

Q / A

www.meetup.com/ScalaNsk

www.twitter.com/ScalaNsk

EXCELSIOR

Q / A

Case classes and pattern matching

- Pattern matching - обобщённый switch/case
- Case classes - реализация ADT

Преимущества Scala:

- Case class - больше чем ADT: интерфейсы, методы, данные
- Pattern matching customizable by user: абстракция логики от структуры данных, DSLs

Extractor objects

```
object PowerOfTwo {
  import java.lang.Long.{numberOfLeadingZeros => nlz}
  def apply(i: Int): Long = 1L << i
  def unapply(x: Long): Option[Int] =
    if (((x & (x-1)) == 0) && (x > 0))
      Some(63 - nlz(x)) else None
}

def optimize(e: Expr) = e match {
  case Mul(x, PowerOfTwo(n)) => Shl(x, n)
  case Div(x, PowerOfTwo(n)) => Shr(x, n)
  case Rem(x, PowerOfTwo(n)) => And(x, PowerOfTwo(n) - 1)
}
```

Pattern matching everywhere

```
def foo(x: Int) = (x, x*x*x)   // Int => (Int, Int)

val xs = List(1,2,3) map foo   // List((1,1),(2,8),(3,27))
val (n, c) = xs.last           // n = 3, c = 27

for ((n, c) <- xs) println(c/n)  // 1, 4, 9


val zs = xs collect { case (x, y) if x < y => y - x }
println(zs)                    // List(6,24)
```

Partial functions

```
class List[T] {
  def collect[U](pf: PartialFunction[T,U]): List[U] = {
    val buf: Buffer[U] = ...
    for(elem <- this) if(pf isDefinedAt elem) buf += pf(elem)
    buf.toList
  }
}

val zs = xs collect { case (x, y) if x < y => y - x }
println(zs)                          // List(6,24)
```

Type variance: определение

Пусть есть Foo[T], и A <: B (A != B).

Есть три возможности:

1. Foo[A] <: Foo[B] - Foo ковариантен по T
2. Foo[B] <: Foo[A] - Foo контравариантен по T
3. Иначе, Foo инвариантен по T

Вариантность задаётся при объявлении класса:

```
class Foo[+T] / class Foo[-T] / class Foo[T]
```

ко- / контра- / ин-

Type variance: примеры

```
trait InChannel[+T] { def get: T }

val in: InChannel[String] = ...
val x = in.get // x: String
val in2: InChannel[Any] = in
val x2 = in2.get // x2: Any

trait OutChannel[-T] { def put(x: T) }

val out: OutChannel[Any] = ...
out.put(42)
val out2: OutChannel[String] = out
out2.put("hello")
```

Type variance: контроль типов

```
class Cell[+T] {  // covariant
  def get: T      // ok
  def put(x: T)   // compilation error
}

class Cell[-T] {  // contravariant
  def get: T      // compilation error
  def put(x: T)   // ok
}

class Cell[T] {   // invariant
  def get: T      // ok
  def put(x: T)   // ok
}
```

Type variance: ещё примеры

```
trait Function1[-T, +R] {   // contravariant argument(s)
  def apply(x: T): R        // covariant result
}

class List[+T] {    // immutable collections are covariant
  def head: T
  def tail: List[T]
}

class Array[T] {  // mutable collections are invariant
  def apply(idx: Int): T
  def update(idx: Int, x: T)
}
```

Параметризация: пример

```
abstract class List[+A] {
  def head: A
  def tail: List[A]
  def isEmpty: Boolean
}

final case class Cons[+A](val head: A, val tail: List[A])
    extends List[A] {
  def isEmpty = false
}
case object Nil extends List[Nothing] {
  def head = throw new NoSuchElementException("Nil.head")
  ...
}
```

Upper & lower type bounds

```
def max[A <: Ordered[A]](xs: List[A]): A = ???

val accounts: Ordered[Money] = ???
max(accounts)



abstract class List[+A] {
  def ++[B >: A](that: List[B]): List[B] = ???
}

val strs = List("foo", "bar")
val nums = List(13, 27)
val all = strs ++ nums     // List[Any]
```

Compound Types

```
trait Cloneable {
 def clone: Cloneable = ???
}
trait Resetable {
  def reset(): Unit
}

def cloneAndReset(obj: Cloneable with Resetable): Cloneable = {
  val cloned = obj.clone
  obj.reset()
  cloned
}
```

Синтаксический сахар

- бесскобочные методы
- auto-generated accessors
- операторы = методы,
- операторный синтаксис
- apply, update
- for loop
- case classes, unapply
- implicits

Для чего? UAP, DSLs, extension methods, FP->OOP mapping, pattern matching