

## Язык программирования Scala

1. [Почему мы выбираем Scala](#)
2. [Особенности языка Scala](#)
3. [Scala с точки зрения Java](#)
4. [Scala точки зрения функциональных языков](#)
5. [Сопоставление с образцом: подробности](#)
6. [Встроенный XML в Scala](#)
7. [Примеры кода Scala](#)
8. [Тестирование в Scala](#)
9. [Готовые продукты на Scala](#)
10. [Где, что делают на Scala](#)
11. [Рекомендуемая литература по языку Scala](#)
12. [Список литературы по языку Scala](#)

Влад Патрышев

### Почему мы выбираем Scala

Языки приходят и уходят; некоторые остаются. Scala - не первый язык, комбинирующий объектное ориентирование с функциональным программированием, но он появился в нужный момент, когда Java уже всем откровенно поднадоела. Теперь Scala вполне готова постепенно заменить Java, работая на той же виртуальной машине и предоставляя доступ к тем же Java-библиотекам.

Мартин Одерски в начале 2000-х разрабатывал параллельно Scala и дженерики для Java. Версия дженериков Java оказалась несколько недоделанной, если не сказать - ошибочной (невозможно указать вариантность параметров [11]). В Scala этот недостаток устранён: вариантность присутствует, и недоуменные вопросы <на счёт дженериков>, так часто возникающие при программировании на Java, сами собой отпадают.

Вариантность параметра типа - это указание на то, что происходит при замене типа на подтип: получим ли мы подтип или нет. Например, немодифицируемый список строк можно использовать как подтип немодифицируемого списка объектов (ковариантность); для модифицируемых такое явление не наблюдается (инвариантность); бывают и случаи, когда отношение подтипов меняет направление (контравариантность).

Так как Scala работает на привычной JVM, и даже компилируется в Java, использовать её можно практически везде, где есть Java версии не ниже 5 (т. е., не на Blackberry). Переход с Java на Scala можно начать с небольшого ознакомления с языком; нет необходимости всё глубоко изучать и всё радикально переписывать. Инфраструктура может оставаться той же самой; например, можно использовать тот же Tomcat, но часть сервлетов будет на Scala.

### Особенности языка Scala

Scala - язык, на котором можно писать в объектно-ориентированном стиле и в функциональном - по вкусу. Если вы привыкли программировать на Java, то при переходе на Scala поменяется не так уж много: в основном, выкинете лишние операции и будете наслаждаться новыми удобствами. Если вы любите программировать на Хаскеле, то обнаружите, что, по крайней мере внешне, многое из того, к чему вы привыкли, можно вполне разборчиво выразить на Scala. Система типов в Scala мощнее, чем в Хаскеле-98 (благодаря наличию higher-ranking types, т. е. дженериков высшего порядка).

### Scala с точки зрения Java

Итак, на Scala можно писать как на обычной Java со слегка изменённым синтаксисом. Всё есть объект: нет примитивных типов. (Читатель, конечно, тут же возмутится - а как же производительность? Стандартный ответ на это таков: JIT всё перемелет в эффективный код. К тому же, начиная с версии 2.8.0, используется специализация с помощью примитивных типов; за подробностями отсылаю к источникам.)

Все Java-классы доступны из Scala, и наоборот. Единственная сложность состоит в том, что коллекции в Java и в Scala очень сильно отличаются, и чтобы работать с Java-коллекцией на Scala так, как это принято на Scala, эту коллекцию надо конвертировать, обычно неявным способом. Но если вы будете писать код <как на Java>, то можно и не конвертировать.

Например, в компании Scala используется вперемешку с Java, и неявные преобразования из Scala-коллекций в Java-коллекции и обратно - обычное дело. Добавим ещё, что в Scala имеются такие же аннотации, что и в Java, что даёт возможность использовать и Guice, и Hibernate.

В Scala нет Java-массивов, а есть вместо него класс `Array`; поэтому квадратные скобки можно использовать для других целей. В Scala квадратные скобки используются для параметров типа, вместо угловых.

Аналогом понятия Java-интерфейса в Scala является типаж (`trait`); типаж может содержать частичную реализацию, и работать в качестве абстрактного класса - хотя класс может наследовать сразу несколько типажей. Это не ведёт к проблемам, типичным для множественного наследования, т. к. речь не идёт о классах, а о типажах.

Метод может принадлежать классу или объекту; функции также определяются внутри класса или объекта. Если вам нужна функция, не привязанная к экземпляру класса, то её можно определить в статическом объекте. По традиции Scala, к классу добавляется так называемый объект-компаньон (companion object) с тем же именем, что и класс:

```
class A {...} object A { def myStaticMethod(s:String) : Integer {...} }
```

Эти же объекты-компаньоны можно использовать в качестве фабрик.

Оператор `<==>` в Scala соответствует, буквально, на Java. `equals()` для ненулевых ссылок и равенству для `null`; для

буквального (ссылочного) равенства в Scala нужно использовать метод `eq`, определённый для класса `AnyRef`.

Тип переменной определяется через двоеточие: `var s : String`. Функции и методы определяются через `def`:

```
def square(n : Int) = n * n
```

Вот как в Scala выглядят шаблоны (generics):

```
var m : Map[String, List[Timestamp]] ... val s = m("today")(5)
```

(Здесь мы извлекаем список по ключу `"today"`, затем берём пятый элемент списка.)

В отличие от Java, шаблоны могут быть высших порядков (см. [9]):

```
trait Functor[F[_]] { def fmap[A, B](fa: F[A], f: A => B): F[B] }
```

Такая конструкция невозможна в Java (см., например, [8]). Невозможно написать вот такой интерфейс:

```
interface Functor<F<?> extends Collection<>> { public <A, B> F<B> fmap(A a, Function<A, B> f); }
```

Циклы можно писать практически как в Java; цикл `while` ничем не отличается, а цикл, известный под названием `for`, выглядит так:

```
def main(args: Array[String]) { for (arg <- args) println(arg + ": " + calculate(arg)) }
```

Можно писать и более вычурные циклы:

```
for (x <- expr1; if expr2; y <- expr3) ...
```

Это, конечно, эквивалентно следующему:

```
for (x <- expr1) { if (expr2) { for (y <- expr3) ... } }
```

Долгожданная новинка - замыкания (closures). Конечно, в Java тоже можно написать что-то вроде замыканий, используя интерфейсы и анонимные классы, но на практике всё это выливается в совершенно нечитаемый код. На Scala замыкания выглядят гораздо более естественно:

```
List("abc.scala", "xyz.py", "file.sh", "myclass.java").filter (name => name.endsWith(".java"))
```

Разумеется, Java-стиль программирования - это только верхушка айсберга; им удобно пользоваться, если вы только переходите с Java на Scala. По мере освоения вы обнаружите, что Scala способна на очень многое, практически невыразимое на Java.

Две новых особенности языка (относительно Java) удивят и кого-то порадуют, а кого-то и нет: неявные преобразования и миксины.

В Java мы привыкли, что `int` приводится к `long` и что в определённом контексте любой объект приводится к `String` - а в Scala этими преобразованиями можно управлять. В качестве примера возьмём строку, и превратим её, с помощью неявного преобразования, в объект, очень похожий на массив:

```
implicit def randomAccess (s:String) = new RandomAccessSeq[Char] { def length = s.length def apply(i: Int) = s.charAt(i) }
```

Теперь можно писать `val c = "This is a string"(2)` - строка будет неявно преобразована в `RandomAccessSeq[Char]`, а значение `c` будет вычислено и равно `'i'`. Ниже приводятся другие примеры неявных преобразований.

Теперь мы можем обращаться со строками, как если бы это были массивы:

```
// returns true if string contains a digit "abc123" exists (_.isDigit)
```

Миксины в Scala реализованы с помощью типажей. Миксины позволяют добавлять к классу функциональность, определённую не в непосредственном суперклассе, а где-то ещё; в Scala типаж может содержать определения методов. Вот простой пример:

```
class Human(name: String, sex: Sex) { def ping() { println("?!") } } class Russian(firstName: String, patronymic: String, lastName: String, sex: Sex) extends Human(firstName + " " + patronymic + " " + lastName, sex) { ... } trait Programmer { override def ping() { println("I'm busy!") } } def debug(byte[] binary) { println("omg...") } } class SovietProgrammer(firstName: String, patronymic: String, lastName: String, birthDate: Timestamp) extends Russian(firstName, patronymic, lastName, null) with Programmer { ... }
```

Здесь методы `debug()` и `ping` наследуются из типаж `Programmer`.

## Scala точки зрения функциональных языков

Несмотря на то, что Scala бежит на JVM и предоставляет все возможности <объектного программирования>, это вполне функциональный язык. Функция может передаваться в качестве параметра, может быть значением переменной,

может быть анонимным литералом, может возвращаться:

```
def myFun(x: Integer) = "Look, " + x + " * " + x + " = " + (x * x) val sameThing = (x: Integer) => "Look, " + x + " * " + x + " = " + (x * x)
List(1,2,3) map {x => x * x * x} List(0., 3.14159265358) map { sin(_) } def const[X, Y](y: Y) = (x: X) => y def c123 = const(123) val
x = c123("abc") // it is 123
```

В последних двух примерах мы взяли списки и применили к каждому элементу функцию, получая новый список. В Scala имеется масса традиционных для ФП операций над списками, например:

```
val list = "Clatto" :: "Verata" :: "Nicto" :: Nil // Same as List("Clatto", "Verata", "Nicto") val list = List("double", "double") :::
List("toi", "and", "trouble") // concatenation list.exists(x => x.toString.length == 4) // wtf method of looking for TRUE
list.drop(2).dropRight(4).filter(s => (s endsWith "y")) list.map(s => "\"" + s + "\"") // list comprehension
List(1,2).flatMap(List(123,_,456)) // lists built by List(123,_,456) are concatenated list.foreach(print) list.reverse list.head list.tail
list.last
```

К спискам, в частности, применимы свёртки:

```
val list = List(1,2,3,4,5) // folds the list, adding up the elements: list.foldLeft(0)(_+_)
```

Есть кортежи (ака n-ки); есть аналог data types, есть карринг - многое из того, ради чего люди переходят на Хаскель. Но так как язык этот в принципе-то императивный, то при вводе-выводе можно формально обойтись без монад.

Пример кортежа:

```
val x = (123, "abc", new Date)
```

Пример алгебраического типа:

```
abstract class Option[T] case class None[T]() extends Option[T] case class Some[T](value: T) extends Option[T]
```

**case class** - особый вид класса, в частности, пригодный для использования в конструкции switch. В таком классе много чего хорошего; казалось бы, почему не сделать все классы case-классами? Проблема в том, что сравнение с образцом становится проблематичным при наличии наследования; поэтому у таких классов наследование ограничено.

Пример карринга:

```
def sum(x: Int)(y: Int) = x + y sum(1)(2)
```

Здесь sum определяется как функция, которая принимает целочисленный аргумент ( *x* ) и возвращает функцию, которая принимает целочисленный аргумент ( *y* ). Если мы напишем `sum(5)`, то это и будет называться <карринг>.

Классы типов (type classes) реализованы с помощью типажей и полиморфизма, например:

```
trait Functor[F[_]] { def fmap[A, B](fa: F[A], f: A => B): F[B] }
```

Выше, в разделе <С точки зрения Java>, этот же пример приведён в качестве образца шаблона высшего порядка: на одну и ту же вещь можно по-разному смотреть.

Вместо переменных в Scala используется **val** - такие значения определяются один раз и не меняются. Можно использовать и **var** - тогда это будет обычная переменная, как в Java. **val** можно объявить ленивой:

```
lazy val x = buildBigThingThatTakesLong(parameters)
```

В этом случае значение вычисляется только в момент, когда оно требуется, в выражении или в качестве неленивого параметра.

В качестве альтернативы ленивой переменной можно использовать функцию без параметров:

```
def x = buildBigThingThatTakesLong(parameters)
```

Между ленивой переменной и функцией без параметров имеется существенное различие: ленивая переменная вычисляется один раз, а функция - каждый раз.

Ещё Scala замечательна сопоставлением с образцом (*pattern matching*; подробнее на с. ??) и частичными функциями.

Как задаётся частичная функция? Через сопоставление с образцом:

```
val second : List[Int] => Int = { case x::y::_ => y }
```

Это эквивалентно следующему коду:



```
val second = new PartialFunction[List[Int], Int] { def apply(xs: List[Int]) = xs match { case x :: y :: _ => y } def isDefinedAt(xs: List[Int]) = xs match { case x :: y :: _ => true case _ => false }
```

(Выражение `x::y::z::Nil` эквивалентно выражению `List(x, y, z)`).

Частичная функция может использоваться по-разному:

```
val pf : PartialFunction[X, Y] = ... val y = pf(x) // exception happens if pf is not defined on the value of x val yOpt : Option[Y] = pf.lift(x) // returns either Some(y) or None val b : Boolean = pf.isDefinedAt(x) val pfAlt : PartialFunction[X, Y] = ... val y = pf.orElse(pf1) // if (pf.isDefinedAt(x)) pf(x) else pf1(x)
```

Естественно, возникает вопрос: как же так, язык объектно-ориентированный; есть методы у объектов - как это всё сочетается с наличием обычных функций, не методов? Ну вот взять, например,

```
class A(i: Integer) { def f1(j: Integer, k: Integer) = i+j+k val f2 = (j: Integer, k: Integer) => i+j+k }
```

Здесь очевидно, что `f1` - метод, а `f2` - функция. Какая разница между `f1` и `f2`? Можно ли написать `val f3 = f1`? Нет, нельзя. Чтобы превратить метод класса в полноправную функцию (т. е., экземпляр типа `Function`), нужно, формально говоря, добавить подчёркивание после пробела:

```
val f3 = f1 _
```

Эта запись буквально означает, что мы определяем новое значение, имеющее тип <функция>, и эта функция имеет те же параметры, что и метод `f1`, и тот же тип результата; и для вычисления `f3` нужно подставить её параметры в `f1`. Так как `f1` вполне может использовать члены класса, в котором она определена, то мы получаем замыкание (closure). `f3` можно передавать в качестве параметра или возвращать; это теперь самостоятельная сущность.

Обычные списки в Скале ленивы: если функция возвращает список, то все его элементы должны быть вычислены. Но есть ленивые потоки (`Stream`). Вот, например, направляем ленивый поток целых чисел в решето Эратосфена и получаем ленивый поток простых чисел:

```
def sieve(s: Stream[Int]): Stream[Int] = Stream.cons(s.head, sieve(s.tail filter { _ % s.head != 0})) def primes = sieve(Stream from 2) primes take 100 foreach println
```

Очень легко создавать встроенные предметно-ориентированные языки (DSL), см. например, [5]:

```
val orders = List[Order]( // use premium pricing strategy new Order to buy(100 sharesOf "IBM") maxUnitPrice 300 using premiumPricing, // use default pricing strategy new Order to buy(200 sharesOf "GOOGLE") maxUnitPrice 300 using defaultPricing, // use custom pricing strategy new Order to sell(200 bondsOf "Sun") maxUnitPrice 300 using { (qty, unit) => qty * unit - 500 }
```

Как ни странно это выглядит, но всё это - Scala, а не макросы и не текст для парсера. Как и в предыдущих примерах, здесь мы пользуемся удобствами языка.

## Сопоставление с образцом: подробности

Одна из приятнейших особенностей Scala - сопоставление с образцом. После Java - сплошное удовольствие.

Сначала простой пример. Сопоставляем значения, как в обычном `switch/case`:

```
def matchMe (s: String): Int = s match { case "ichi" => 1 case "ni" => 2 case "san" => 3 case _ => -1 }
```

На самом деле Java с 2007 года обещает конструкцию `switch` и для строк: Скоро будет.

В Scala с помощью образцов можно сопоставлять не только значения, но и алгебраические типы данных, особенно учитывая, что они допускают деконструкцию:

```
trait Expr { case class Num(value : int) extends Expr case class Var(name : String) extends Expr case class Mul(lft : Expr, rgt : Expr) extends Expr } ... // Simplification rule: expr match { case Mul(x, Num(1)) => x case _ => e }
```

Сопоставление с образцом, вместе с регулярными выражениями и с экстракторами, даёт удобную возможность разбирать текст:

```
scala> val Entry = """"(w+)\s*=\s*(ld+)""".r Entry: scala.util.matching.Regex = (w+)\s*=\s*(ld+) scala> def parse(s: String) = s match { case Entry(x,y) => (x, y) } parse: (s: String)(String, Int) scala> parse("salary=120") res50: (String, Int) = (salary,120)
```

Это, конечно, если язык описывается регулярным выражением. Если же грамматика посложнее, то нужно писать парсер-комбинатор, как в нижеследующем примере.

// a regular tree, not binary,  $Tree[X] = X + Tree[X]^n$  **abstract class** Tree[X] // main class for a tree **case class** Leaf[X](v:X) **extends** Tree[X] **case class** Branch[X](kids:List[Tree[X]]) **extends** Tree[X] **class** TreeParser **extends** JavaTokenParsers { // A tree is either a leaf or a branch **private** def node : Parser[Tree[String]] = leaf | branch // a branch is a sequence of trees in parentheses; // postprocessing consists of calling a constructor **private** def branch : Parser[Branch[String]] = "("~repsep(node, ",")~")" ^^ { **case** "("~nodes~")" => Branch(List() ++ nodes) } // a leaf is a string wrapped in a constructor **private** def leaf : Parser[Leaf[String]] = string ^^ { **case** s => Leaf(s) } // a string is a string is a string **private** def string : Parser[String] = regex(""""w+""") // this is all that's exposed def read(input: CharSequence) = parseAll(node, input).get def parseTree(input: CharSequence) = (new TreeParser).read(input) scala> parse("(a,(b,(c,(d,e,f),g)))" res58: Tree[String] = Branch(List(Leaf(a), Branch(List(Leaf(b), Branch(List(Leaf(c), Leaf(d), Branch(List(), Leaf(f))))), Leaf(g))))

## Встроенный XML в Scala

Для определённого рода приложений (ну, скажем, для веба) тот факт, что XML является в Scala частью языка, освобождает от необходимости писать массу рутинного ненужного кода. Вот пример:

```
class Person(val id: Int, val firstName: String, val lastName: String, val title: String) { def toXML = { <person> <id>{id}</id> <firstName>{firstName}</firstName> <lastName>{lastName}</lastName> <title>{title}</title> </person> } }
```

## Примеры кода Scala

Любимый хаскельцами квиксорт, правда, в ленивом варианте (т. е., не имеющий практического смысла), т. к. списки не ленивы:

```
def qsort(list: List[Int]): List[Int] = list match { case Nil => Nil case x::xs => qsort(xs.filter(_ < x)) ::: x :: qsort(xs.filter(_ >= x)) }
```

Здесь мы берём список, и, если он пуст, возвращаем его же, а иначе берём первый элемент; сортируем элементы, которые меньше этого первого; сортируем элементы, которые больше этого первого; наконец, соединяем всё в нужном порядке.

Хитрый трюк с неявным преобразованием, позволяющий определять операторы на существующих классах:

```
> class Fact (n: Int) { > def fact(n: Int): BigInt = if (n == 0) 1 else fact(n-1) * n > def ! = fact(n) > } > implicit def int2fact(n: Int) = new Fact(n) > > println(42!) 140500611775287989854314260624451156993638400000000
```

Что тут у нас получилось? Мы вычисляем 42!; т. к. оператор ! определён на значениях типа Fact, то мы пробуем неявное преобразование Integer в Fact - а так как такое преобразование определено (int2fact), то его и используем. После чего к результату применяем оператор ! - и получаем в результате BigInt; его и печатаем в println().

## Тестирование в Scala

Для юниттестов имеется два пакета: scalatest ([12]) и scalacheck. Вот типичный пример теста для scalatest:

```
import org.scalatest.FlatSpec import org.scalatest.matchers.ShouldMatchers class StackSpec extends FlatSpec with ShouldMatchers { "A Stack" should "pop values in LIFO order" in { val stack = new Stack[Int] stack.push(1) stack.push(2) stack.pop() should equal (2) stack.pop() should equal (1) } it should "throw Exception on popping empty stack" in { val emptyStack = new Stack[String] evaluating { emptyStack.pop() } should produce [NoSuchElementException] }
```

В версии для scalacheck это может выглядеть примерно так:

```
val stackIsLIFO = forall { (stack: Stack[Int], x: Int, y: Int) => stack.push(x) stack.push(y) (y != stack.pop) | "stack top should pop" && (x != stack.pop) | "stack bottom lost" }
```

scalacheck для тестирования генерирует порядочное количество различных стеков, проверяя указанные свойства. Какие именно стеки генерируются, зависит от интеллекта (искусственного) - необходимый набор тестовых примеров выводится из условий и из определения.

Но можно писать и по-простому, как в JUnit:

```
def testParse_positive_5 { val stack = new Stack[Int] stack.push(1) stack.push(2) assert(2 == stack.pop()) assert(1 == stack.pop()) assert(stack.isEmpty) }
```

## Готовые продукты на Scala

Scala можно скачать с <http://www.scala-lang.org/downloads>; этот дистрибутив включает в себя компилятор, библиотеки, герп-интерпретатор, и документацию. Очень рекомендую прекрасный плагин для Scala в IntelliJ: этот плагин делает всё, что надо - рефакторинг, юниттесты, есть и отладчик. Версия 2.8 плагина для Eclipse тоже вполне работоспособна; существует также плагин для Netbeans.

Ну и наконец, Lift, ради которого уже стоит срочно браться за Scala, так как этот веб-фреймворк лет на десять обогнал большинство остальных.

Lift, фреймворк для создания веб-приложений, разумеется, опирается на всё, что было достигнуто в вебе за последние десять лет, но он идёт дальше. Прежде всего, это строгое разделение компонент MVC; затем, каждая форма уникальна, и невозможно передать одну и ту же информацию дважды или вернуться на ту же страницу: каждый экземпляр формы содержит уникальную метку. Так как Scala может содержать XML, не нужны отдельные JSP, это всё пишется прямо в коде - но презентация отделена от логики структурой приложения.

Вот что сказал Мартин Одерски, дизайнер Scala:

Образец HTML и соответствующего кода на Scala:

```
<table> <lift:Show.users> <tr> <td></f:first_name>David</f:first_name></td> <td></f:last_name>Pollak</f:last_name></td> </tr>
</lift:Show.users> </table> class Show { def users(xhtml: NodeSeq) = Users.findAll.flatMap(user => bind("f", xhtml, "first_name" -
> user.firstName, "last_name" -> user.nameName)) }
```

## Где, что делают на Scala

Самый сейчас передовой пример - это [foursquare.com](http://foursquare.com), небольшой нью-йоркский стартап, чья социальная сеть следит за вашим местоположением и даёт возможность пригласить всех в понравившийся вам ресторан и отследить, кто где находится. За полгода весь код переписан на ScalaLift.

Yammer: Artie, служба рассылки сообщений, написана на Scala.

Twitter: порядочное количество серверного кода переписано на Scala, а остальное пока на Ruby on Rails.

LinkedIn: там было порядочное количество Scala-программистов, но, пока писалась эта статья, они все, похоже, уже разбежались - как и из Гугла до того.

KaChing: желающие пишут на Scala, хотя основная масса кода на Java; проблем совместимости нет.

Есть ещё пара стартапов в Сан-Франциско, пишущих исключительно на Scala; но если вы пойдёте на [Dice](http://Dice), то обнаружите с десяток контор, где требуется Scala, даже в таких экзотических местах, как штат Теннесси.

## Рекомендуемая литература по языку Scala

David Pollack [10] - пособие для начинающих. Если вы серьёзно хотите изучить и использовать Scala, вряд ли вам нужна эта книга. Для серьёзного изучения лучше всего подойдёт

Martin Oderski, Lexi Spoon, Bill Venners [13]. Большая книга, но её не обязательно читать вдоль, можно и поперёк. [14] - перевод главы из этой книги.

Dean Wampler, Alex Payne [1]. В этой книге меньше страниц, но больше Scala, различных пикантных деталей. Хороша в качестве <второй книги> по этому языку.

[18], [16], [17] - три статьи Антона Панасенко, дельно описывающие те или иные аспекты языка.

[15] - русский перевод статьи <Обзор языка Scala> Мартина Одерски и множества других.

[9] - проект ScalaZ, Тони Морриса (Tony Morris). Читается как стихи Лорки в оригинале.

Ну и, наконец, для настоящих героев - книга Gregory Meredith [7]. Мередит - большой теоретик, и читать его непросто. Но стоит того: если вы научитесь рассуждать на его уровне, то уже ничего не страшно.

Derek Chen-Becker, Tyler Weir, Marius Danciu [6]. Небольшая книга по ScalaLift. Так как Lift развивается быстрее, чем печатаются книги, то лучше информацию черпать из следующих источников: [4], [3], [2].

## Список литературы по языку Scala

- [1] Dean Wampler Alex Payne. *Programming Scala: Scalability = Functional Programming + Objects*, <http://programming-scala.labs.oreilly.com/>. O'Reilly, 2009.
- [2] Derek Chen-Becker. Проект на Lift - <<Мелочь в Кармане>>. Проект, <http://github.com/tjweir/pocketchangeapp/tree/master/PocketChange>.
- [3] Marius Danciu David Pollak, Derek Chen-Becker and Tyler Weir. Starting with Lift. Веб-страница, [http://old.liftweb.net/docs/getting\\_started/mod\\_master.html](http://old.liftweb.net/docs/getting_started/mod_master.html).
- [4] David Pollack et al. Getting Started With Lift. Веб-сайт, [http://liftweb.net/getting\\_started](http://liftweb.net/getting_started).
- [5] Debashish Ghosh. Designing Internal DSLs in Scala. Блог, <http://debasishg.blogspot.com/2008/05/designing-internal-dsls-in-scala.html>.
- [6]



Tyler Weir Marius Danciu, Derek Chen-Becker. *The Definitive Guide to Lift: A Scala-based Web Framework*, <http://www.amazon.com/Definitive-Guide-Lift-Scala-based-Framework/dp/1430224215>. Apress, 2007.

[7] Gregory Meredith. *Pro Scala: Monadic Design Patterns for the Web*, <http://www.amazon.com/Pro-Scala-Monadic-Design-Patterns/dp/143022844X>. Apress, 2010?

[8] JP Moresmaugh. Java and higher order generics. Блог, <http://jpmoresmau.blogspot.com/2007/12/java-and-higher-order-generics.html>.

[9] Tony Morris. Проект scalaz. Проект в Google Code, <http://code.google.com/p/scalaz/>.

[10] David Pollack. *Beginning Scala*, <http://www.amazon.com/Beginning-Scala-David-Pollak/dp/1430219890>. Apress, 2009.

[11] David Rupp. Java generics broken? we report, you decide. Блог, <http://davidrupp.blogspot.com/2008/01/java-generics-broken-we-report-you.html>.

[12] Bill Venners. Scalatest. Проект, <http://www.scalatest.org/>.

[13] Bill Venners, Martin Odersky, and Lexi Spoon. *Programming in Scala: A Comprehensive Step-by-step Guide*, <http://www.amazon.com/Programming-Scala-Comprehensive-Step---step/dp/0981531601>. Artima, 2008.

[14] Лекси Спун, Бил Веннерс, Мартин Одерски. Первые шаги в Scala. RSDN, <http://www.rsdn.ru/article/scala/scala.xml>.

[15] Мартин Одерски и другие. Обзор языка программирования Scala. RSDN, <http://www.rsdn.ru/article/philosophy/Scala.xml>.


[16] Антон Панасенко. Scala: Actors (part 2). Блог, <http://blog.apanasenko.me/2009/12/scala-actors-part-2/>.

[17] Антон Панасенко. Scala: Functional Language (part 3). Блог, <http://blog.apanasenko.me/2009/12/scala-functional-language-part-3/>.

[18] Антон Панасенко. Scala: введение в мир FL JVM (part 1). Блог, <http://blog.apanasenko.me/2009/12/scala-fl-jvm-part-1/>.

[ТП](#) [ОКМ](#) [ТПОИ](#) [ДБИ](#) [АиЯ](#) [3GL - языки выс. уровня](#) [4GL - визуальные среды](#) [АСДП](#) [ООП](#) [ОСП](#)

**Знаете ли Вы**, что **Программный сниппет** (англ. snippet — фрагмент, отрывок) в практике программирования — небольшой фрагмент исходного кода или текста, пригодный для повторного использования. Сниппеты не являются заменой процедур, функций или других подобных понятий структурного программирования. Они обычно используются для более лёгкой читаемости кода функций, которые без их использования выглядят слишком перегруженными деталями, или для устранения повторения одного и того же общего участка кода. Интегрированные среды разработки (IDE) содержат встроенные средства для ввода конструкций языка. Например, в Microsoft Visual Studio, Borland Developer Studio, для этого необходимо ввести ключевое слово и нажать определённую клавишную комбинацию. В IDE Geany существует специальный файл snippets.conf (путь к файлу: /home/user/.config/geany) позволяющий создавать свои сниппеты. Другие программы, такие как Macromedia Dreamweaver и Zend Studio, позволяют использовать сниппеты в Веб-программировании.



РИЦАРИ ТЕОРИИ ЭФИРА

НОВОСТИ ФОРУМА

27.04.2016 - 07:59: [СОВЕСТЬ - Conscience](#) -> [Проблема государственного терроризма](#) - Карим\_Хайдаров.

25.04.2016 - 07:47: [СОВЕСТЬ - Conscience](#) -> [КОЛЛАПС МИРОВОЙ ФИНАНСОВОЙ СИСТЕМЫ](#) - Карим\_Хайдаров.

24.04.2016 - 21:11: [АСТРОФИЗИКА - Astrophysics](#) -> [Комета 67P/Чурюмова-Герасименко и проблема ее происхождения](#) - Евгений\_Дмитриев.

20.04.2016 - 12:33: [ЭКОЛОГИЯ - Ecology](#) -> [ЭКОЛОГИЯ ДЛЯ ВСЕХ](#) - Карим\_Хайдаров.

17.04.2016 - 22:31: [СОВЕСТЬ - Conscience](#) -> [РУССКИЙ МИР](#) - Карим\_Хайдаров.

09.04.2016 - 06:59: [АСТРОФИЗИКА - Astrophysics](#) -> [Сезонные колебания уровня вод морей и океанов](#) - Юсуп\_Хизиров.

28.03.2016 - 16:42: [СОВЕСТЬ - Conscience](#) -> [ПРАВОСУДИЯ.НЕТ](#) - Карим\_Хайдаров.

17.03.2016 - 11:20: [СЕЙСМОЛОГИЯ - Seismology](#) -> [Запасы воды под Землей](#) - Карим\_Хайдаров.

15.03.2016 - 16:15: [ЦИТАТЫ ЧУЖИХ ФОРУМОВ - Outside Quotings](#) -> [ВЫМИРАНИЕ ДИНОЗАВРОВ на www.nkj.ru](#) - Карим\_Хайдаров.

23.02.2016 - 20:34: [Беседка - Chatter](#) -> [Приливы и отливы](#) - Юсуп\_Хизиров.

19.02.2016 - 05:38: [ФИЗИКА ЭФИРА - Aether Physics](#) -> [Скорость распространения гравитации](#) - Карим\_Хайдаров.