Руководство по Scala для Java программистов

Версия 1.3
декабрь 10, 2010

Michel Schinz,
Philipp Haller,
Ржевский Дмитрий (пер.)

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 Введение

Этот документ дает возможность быстро ознакомиться с языком программирования и компилятором Scala.Этот документ предназначен для людей, которые уже имеют опыт в программировании и хотят посмотреть, что они могут сделать в Scala. При этом необходимо наличие основных знаний объектно-ориентированного программирования, особенно на Java.

2 Первый пример

Как первый пример мы будем использовать стандартную программу Hello world. Он**а** не очень интересная, но позволяет легко продемонстрировать использование инструментов Scala без особого знания языка. Вот как она выглядит:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!" )
  }
}
```

Структура этой программы должна быть знакома Java-программистам: она состоит из метода main который берет аргументы командной строки, массив строк, как параметр; тело этого метода состоит из одного вызова предопределенного метода println с дружественным приветствием как аргумент. Метод main не возвращает значение (это процедурный метод).В связи с этим не нужно декларировать тип возвращаемого результата.

Что наименее известно Java программистам - это декларация **object,** содержащая main метод. Такая декларация знакомит с широко известным *singleton объектом,* который является классом с одним экземпляром. Таким образом, приведённая выше декларация декларирует и класс с именем HelloWorld и экземпляр этого класса также с именем HelloWorld. Этот экземпляр создаётся по требованию (on demand), как только он начинает использоваться.

Проницательный читатель может заметить, что main метод не декларирован как static. Это потому что static-члены (методы или поля) не существуют в Scala. Вместо определения статических членов (методов и полей) программист Scala декларирует эти члены в singleton объектах.

2.1 Компилирование примера

Для компиляции примера, мы используем scalac, компилятор Scala. scalac работает как большинство компиляторов: берёт исходные файлы как аргумент, возможно несколько опций, и создаёт один или несколько объектных файлов. Объектные файлы, которые он создаёт - стандартные java класс файлы.

Если мы сохраним приведённую выше программу в файле HelloWorld.scala, мы сможем скомпилировать ее набрав следующую команду (знак больше ‘>’ -- приглашение командной строки, и не он должен набираться):

```
> scalac HelloWorld.scala
```

Данная команда сгенерирует несколько class файлов в текущей директории. Один из них будет называться HelloWorld.class и содержать класс, который может быть прямо запущен

командой scala, как это будет показано в следующей секции.

2.2 Запуск примера

Однажды скомпилированная программа Scala может быть запущена использовав команду scala. Её использование очень похоже на команду java для запуска java программ и принимает те же опции. Предыдущий пример может быть выполнен при помощи следующей команды, которая выдаёт ожидаемый результат:

```
> scala -classpath . HelloWorld

Hello, world!
```

3 Взаимодействие с Java

Одной из сильных сторон Scala это то что она очень легко взаимодействует с кодом Java. Все классы с пакета java.lang импортированы по умолчанию, в то время другие пакеты нужно явно импортировать.

Давайте посмотрим на пример, который демонстрирует это. Мы хотим получить и форматировать текущую дату в соответствии с соглашениями принятыми в конкретной стране, например для Франции[1].

Библиотека классов Java содержит мощные утилитные классы, такие как Date и DateFormat. Так как Scala гладко взаимодействует с Java, нет нужды реализовывать эквивалентные классы в библиотеке классов Scala - мы можем просто импортировать соответствующие Java пакеты:

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._
object FrenchDate {
    def main(args: Array[String]) {
        val now = new Date
        val df = getDateInstance(LONG, Locale.FRANCE)
        println(df format now)
    }
}
```

Выражение импорта в Scala выглядит очень похоже на Java эквивалент, при этом оно более мощное. Множество классов может быть импортировано с того же пакета, окружая их фигурными скобками, как показано в первой строке. Другое отличие - это когда импортируются все имена пакета или класса, используется символ подчёркивания (_) вместо звёздочки (*). Это потому что звёздочка - корректный идентификатор в Scala (например, имя метода), как мы увидим позднее.

Таким образом, выражение импорта на третьей строке импортирует всех членов класса DateFormat. Это делает статический метод getDateInstance и статическое поле LONG непосредственно видимыми.

Внутри main метода мы сначала создадим экземпляр Java-класса Date, который по умолчанию содержит текущую дату. Далее мы определяем формат даты**,** используя статический getDateInstance метод**,** который мы перед этим импортировали. И, наконец, мы печатаем текущую дату, отформатированную в соответствии с локализованным

---

1 Другие регионы, такие как франкоговорящая часть Швейцарии, используют те же соглашения.

DateFormat экземпляром. Последняя строка показывает интересное свойство синтаксиса Scala. Метод, получающий один аргумент, может быть использован с инфиксным синтаксисом. Вот это выражение:
df format now
немного другой, немного менее многословный способ написания выражения
df.format(now)
Это может показаться незначительной синтаксической деталью, но она имеет важные следствия одно из которых будет изучено в следующей главе.

Подводя итоги этой главы об интеграции с Java, нужно заметить что также возможно наследовать от Java классов и реализовать Java интерфейс напрямую в Scala.

4 Всё является объектом

Scala - чистый объектно-ориентированной язык в том смысле, что всё в нем является объектом, включая числа или функции. Это отличает его от Java т.к. Java проводит различие между примитивными типами (такие как boolean и int) и ссылочными типами, и не позволяет манипулировать функциями как значениями.

4.1 Числа являются объектами

Т.к. числа являются объектами, они также имеют методы. И действительно, такое арифметическое выражение как:
1 + 2 * 3 / x
содержит исключительно вызовы методов, потому что (как мы видели из предыдущей главы) оно эквивалентно следующему выражению:
(1).+(((2).*(3))./(x))
Это также значит что +, *, и т.д. правильные идентификаторы в Scala.
Скобки вокруг чисел во второй версии нужны, потому что лексер **Scala** использует правило самого длинного соответствия для токенов. Поэтому он разобьёт следующее выражение:
1.+(2)
на токены 1., +, и 2. Причина того**,** что данная токенизация выбрана в том, что 1. является более длинным правильным соответствием, чем 1. Токен 1. интерпретируется как литерал 1.0, делая его Double а не Int.Запись выражения как:

(1).+(2)

не позволяет 1 интерпретировать 1 как Double.

4.2 Функции являются объектами.

Наиболее удивительно для Java программистов то, что функции также являются объектами в Scala. По этой причине возможно передавать функцию как аргумент, хранить их в переменных и возвращать их от других функций. Эта возможность манипулировать функциями как значениями - один из краеугольных камней очень интересной парадигмы программирования, называемой *функциональным программированием*.

Как очень простой пример, почему использование функций как значений может быть полезно, давайте рассмотрим функцию таймера чья цель выполнять некоторые действия каждую секунду. Как мы передадим таймеру действие для выполнения? Вполне логично, как функцию. Это очень простой вид передачи функции должен быть знаком программистам: он часто используется в коде пользовательского интерфейса (UI), чтобы зарегистрировать функции обратного вызова (call back), вызываются, когда происходят некоторые события.
В следующей программе функция таймера, которая называется oncePerSecond, получает call-back функцию как аргумент. Тип этой функции записан () => Unit и это тип всех функций которые не получают аргументов и ничего не возвращают (тип Unit похож на void в C/C++). Функция main этой программы просто вызывает функцию таймера с call-back функцией, которая печатает предложение на терминал. Другими словами, эта программа бесконечно каждую секунду печатает предложение “time flies like an arrow”.

```
object Timer {
   def oncePerSecond(callback: () => Unit) {
      while (true) { callback(); Thread sleep 1000 }
   }
   def timeFlies() {
      println("time flies like an arrow...")
   }
   def main(args: Array[String]) {
      oncePerSecond(timeFlies)
   }
}
```

Обращаем внимание для того, чтобы напечатать строку мы используем предопределённый метод println вместо того, чтобы использовать его из System.out.

4.2.1 Анонимные функции

Пока программу легко понять, её можно немного улучшить. В начале обращаем внимание на то, что функция timeFlies определена только для того,чтобы быть переданной после в функцию oncePerSecond**.** Необходимость дать имя данной функции, которая используется только один раз, может показаться излишним, но, в действительности, было бы хорошо создать эту функцию только для того, чтобы передать её в oncePerSecond. В Scala возможно использовать анонимные функции*,* которые как раз и являются функциями без имени. Пересмотренная версия нашей программы-таймера, где используются анонимные функции вместо timeFlies, выглядит так:

```
object TimerAnonymous {
   def oncePerSecond(callback: () => Unit) {
      while (true) { callback(); Thread sleep 1000 }
   }
def main(args: Array[String]) {
   oncePerSecond(() =>
      println("time flies like an arrow..."))
   }
}
```

Присутствие анонимной функции в этом примере обнаруживается с помощью правой стрелки ‘=>’ , которая отделяет список аргументов функции от его тела. В этом примере список аргументов пуст, о чем свидетельствует пустая пара скобок слева от стрелки. Тело данной функции такое же, как и тело функции timeFlies, описанной выше.

5 Классы

Как мы видели выше, Scala – объектно-ориентированный язык и, таким образом, имеет концепцию классов[2].Классы в Scala описываются с применением синтаксиса близкого к синтаксису Java. Одна важная разница в том, что классы в Scala могут иметь параметры, как показано в следующем определении комплексных чисел:

```
class Complex(real: Double, imaginary: Double) {
def re() = real
def im() = imaginary
}
```

Этот complex класс получает два аргумента, которые являются реальной частью и мнимой частью комплексного числа. Аргумент должен быть передан, когда создаётся экземпляр класса Complex, например**:** new Complex(1.5, 2.3). Класс содержит два метода, названные re и im, которые дают доступ эти двум частям.

Стоит отметить, что возвращаемый тип этих двух методов не задаётся явно. Он будет выведен компилятором автоматически, который смотрит на правую часть данных методов и делает вывод, что оба возвращают значение типа Double.

Компилятор не всегда может определить тип, как он определил его здесь, и, к сожалению, нет простого правила узнать точно, когда он определит, а когда нет. На практике это обычно не проблема, потому что компилятор жалуется, когда он не может определить тип, который явно не задан. Как простое правило, начинающие Scala программисты могут попытаться опустить декларации типа, которые по их мнению легко установить из контекста, и посмотреть согласен ли компилятор. Через некоторое время программисты должны получить хорошее представление о том, когда можно опустить декларацию типов, а когда нужно указать их точно.

5.1 методы без аргументов

Маленькая проблема методов re и im заключается в том, что для того, чтобы вызвать их, нужно поставить пустую пару скобок после их имени, как в следующем примере.

```
object ComplexNumbers {
   def main(args: Array[String]) {
     val c = new Complex(1.2, 3.4)
   println("imaginary part: " + c.im())
  }
}
```

Было бы гораздо лучше иметь возможность доступа к real и imaginary частям как если бы они были полями, без написания пустой пары скобок. Это можно отлично выполнить в Scala, просто определив их как методы *без аргументов*. Такие методы отличаются от методов с нулевым количеством аргументов тем, что они не имеют скобок после их имени ни при их определении, ни при их использовании. Наш класс Complex можно переписать так:

```
class Complex(real: Double, imaginary: Double) {
   def re = real
   def im = imaginary
}
```

---

2 В завершение, важно отметить, что некоторыеобъектно-ориентированные языки не имеют концепции классов, но Scala - не один из них

5.2 Наследование и переопределение

Все классы в Scala наследуются от суперкласса. Когда суперкласс не задан, как в примере с Complex предыдущей главы, неявно используется scala.AnyRef.
Также возможно перекрывать методы, унаследованные от суперкласса в Scala. Тем не менее, необходимо указать, что метод перекрывает другой, используя override модификатор, для того, чтобы предотвратить случайное перекрытие. Например, наш Complex класс может быть доработан переопределением метода toString, унаследованного от Object.

```
class Complex(real: Double, imaginary: Double) {
   def re = real
   def im = imaginary
   override def toString() =
      "" + re + (if (im < 0) "" else "+") + im + "I"
}
```

6 Case -классы и поиск по шаблону.

Вид структуры данных которые часто появляются в программах — это дерево. Например интерпретаторы и компиляторы внутри представляют программы как дерево; XML документы - деревья; несколько видов контейнеров основаны на деревьях таких как красно-чёрные деревья.
Мы рассмотрим как такие деревья представляются и управляются в Scala через простую программу калькулятор. Задача этой программы - управление очень простыми арифметическими выражениями, составленных из сумм целочисленных констант и переменных. Два примера таких выражений: 1+2 и (x + x) + (7 + y).
Сначала мы решим как представить такие выражения. Наиболее естественный способ- дерево, где ноды являются операциями(здесь сложение) и листья значениями(константами и переменные).
В Java такие деревья будут представлены абстрактным суперклассом для деревьев, и одним конкретным классом на лист или вершину. В функциональных языках программирования можно использовать алгебраические типы данных для тех же целей. Scala предоставляет концепцию case-классов. которые есть что-то среднее среднее двумя. Здесь показано как их можно использовать для определения типа деревьев для нашего примера:

```
abstract class Tree
   case class Sum(l: Tree, r: Tree) extends Tree
   case class Var(n: String) extends Tree
   case class Const(v: Int) extends Tree
```

Факт того что классы Sum, Var и Const объявлены как case-классы значит что они отличаются от стандартных классов в нескольких аспектах:

• ключевое слово new не обязательно для создания экземпляра этих классов (то есть можно писать Const(5) вместо new Const(5)),
• getter функции автоматически определены для параметров конструктора (то есть возможно получить значение параметра конструктора v некоторого экземпляра с класса Const просто написав c.v),
• Даны определения методов по умолчанию для методов equals и hashCode, которые работают со структурой экземпляров, а не с их идентичностью
•Дано определени метода по умолчанию toString и печатает значение в "исходной форме" (то есть дерево для выражения x+1 печатается как Sum(Var(x),Const(1)))

Сейчас мы определили типы данных чтобы представить арифметические выражения, теперь мы можем начать определять операции для манипулирования над ними. Мы начнём с функции вычисления выражения в некотором *окружении*. Назначение окружения - дать переменным значения. Например выражение x+1 вычислится в окружении которое ассоциирует значение 5 с переменной x, напишем {x → 5}, даст результат 6.
Таким образом мы нашли способ представить окружения. Конечно мы можем использовать ассоциативные структуры данных такие как хэш таблицы, но мы также можем напрямую использовать функции! Действительно окружение ничего более чем функция, которая ассоциирует значение к имени(переменной). Окружение {x → 5} данное ранее может просто переписано на Scala:
{ case "x" => 5 }
Эта нотация определяет функцию, которая для данной строкй "x" как аргумент возвращает целое число 5, иначе падает с исключением. Перед написанием вычисляющей функции давайте дадим имя типу окружений. Мы можем конечно же использовать тип String => Int для окружений, упростит программу если мы дадим имя для такого типа и сделаем дальнейшие изменения легче. Это выполняется в Scala в следующей нотации:
type Environment = String => Int
С этого момента тип Environment может быть использован как псевдоним типа функций от String в Int.
Сейчас мы можем дать определение функции вычисления. Концептуально это очень просто: значение суммы двух выражений просто сумма значений этих выражений; значение переменной получается прямо из окружения; значение константы - сама константа. Выразить это на Scala не трудно:

```
def eval(t: Tree, env: Environment): Int = t match {
   case Sum(l, r) => eval(l, env) + eval(r, env)
   case Var(n)    => env(n)
   case Const(v)  => v
}
```

Здесь вычисляющая функция работает выполняя *поиск по шаблону(pattern matching)* в дереве t. Интуитивно значение данного выше определения нужно уточнить:

1. Первая проверка - является ли дерево суммой, если да связывается левое под-дерево к новой переменной названной l и правое под-дерево к переменной **r** и далее продолжит вычисление выражения после стрелки; выражение может использовать (и использует) переменные связанные шаблоном находящимся с левой стороны от стрелки. т.е. **l** и **r** .

Если первая проверка не прошла, т.е. дерево не является суммой, проверяется не является оно Var; если да, связывается с именем содержащемся в узле Var к переменной n и продолжает с правой частью.

Если вторая проверка не прошла, т.е.если переменная t не является ни Sum ни Var, проверяется является ли она Const, и если является, связывается значение содержащееся в узле Const с переменной v и и прдолжает с правой частью.

В конце если все проверки не прошли, вызывается исключение для сигнализации ошибки в выражении поиска по шаблону. Это может случиться только если декларировано большее количество подклассов класса Tree.

2. Мы видели что основная идея поиска по шаблона - попытаться найти значение по серии шаблонов, и как только значение по шаблону найдено, выделить и дать имена различным частям значения, чтобы наконец выполнить некоторый код который

использует эти именованные части

Опытный объектно-ориентированный программист может быть изумлён почему мы не определили eval как метод класса Tree и его подклассов. Мы действительно можем это сделать поскольку Scala разрешает определения методов в case-классах как и в обычных классах. Решая будем ли мы использовать поиск по шаблону или методы - вопрос вкуса, но это также имеет влияние на расширяемость:

- когда используются методы легко добавить новый вид узла т.к. Это может сделано всего лишь определяя подкласс Tree;с другой стороны добавление новой операции для управления деревом трудоёмко, т.к. Это требует изменений во всех подклассах Tree,
- когда применяется поиск по шаблону ситуация обратная: добавление нового типа узла требует модификацию всех функций которые делают поиск по шаблону в дереве, для для принятия во внимание нового узла; с другой стороны, добавление новой операции легко, просто определяя независимую функцию.

Для дальнейшего исследования поиска по шаблону давайте определим другую операцию в арифметическом выражении: символическая производная. Читатель может помнит следующие правила касающиеся этой операции:

1. производная суммы равняется сумме производных
2. производная некоторой переменной v равняется единице если v - переменная по которой берётся производная, иначе равняется нулю.
3. производная константы равняется нулю.

Эти правила могут переведены почти дословно в Scala код, получив следующее определение:

```
def derive(t: Tree, v: String): Tree = t match {
case Sum(l, r) => Sum(derive(l, v), derive(r, v))
case Var(n) if (v == n) => Const(1)
case _ => Const(0)
}
```

Эта функция знакомит с двумя концепциями касающихся поиска по шаблону. Во-первых case-выражение для переменных имеет ограничитель, выражение следующее за ключевым словом if. Ограничитель отменяет успешный поиск по шаблону если выражение не равняется true. Здесь это используется чтобы быть уверенным чтобы возвращаемая константа равна 1 только в случае если имя дифференцируемой переменной такое же как имя дифференцирующей переменной v. Вторая новая особенность поиска по шаблону используемая здесь - использование группового символа, записываемого как _, который является при поиске по шаблону любым значением,без задавания ему имени. Мы ещё не изучали всю мощь поиска по шаблону, но мы остановится здесь для того, чтобы чтобы сохранить документ коротким. Мы всё хотим увидеть как две вышеприведённые функции выполнятся на реальном примере. Для этих цели давайте напишем простую main функцию, которая выполнит несколько операций на выражении (x + x) + (7 + y): сначала она считает значения в окружении $\{x \to 5, y \to 7\}$, затем вычисляет производную по x и далее по y.

```
def main(args: Array[String]) {
val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
val env: Environment = { case "x" => 5 case "y" => 7 }
println("Expression: " + exp)
println("Evaluation with x=5, y=7: " + eval(exp, env))
println("Derivative relative to x:\n " + derive(exp, "x"))
println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

Выполнив эту программу, мы получим ожидаемый результат:
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))

Изучая вывод (программы) мы видим, что результат производной целесообразно упростить, перед показом пользователю. Определение простой упрощающей функции используя поиск по шаблону - интересная (но на удивление сложная) проблема, оставим как упражнение читателю.

7 Трэйты

Отдельно от наследования кода от суперкласса, классы Scala могу также импортировать код от одного или несколько трэйтов.
Возможно наболее лёгкий путь для Java программистов понять что такое трэиты является рассмотрение их как интерфейсы, которые также содержат код. В Scala когда класс наследуется от трэйта, он реализует интерфейс трэита и наследует весь код содержащийся в трэите.
Чтобы увидеть полезность трэйтов, давайте рассмотрим классический пример: упорядоченные объекты. Обычно полезна возможность сравнивать объекты данного класса межу собой, например чтобы отсортировать их. В Java, объекты которые можно сравнить реализуют Comparable интерфейс. В Scala мы можем сделать это немного лучше чем в Java, определяя эквивалент Comparable интерфейса как трайт, который мы назывём Ord.

Когда сравниваются объекты, 6 различных предикатов могут быть полезны: меньше, меньше или равно, равно, не равно, больше или равно, больше. Однако определение всех их трудоёмко, особенно ввиду четыре из них можно выразить через оставшихся два. Так через предикаты (к примеру) равно и меньше , можно выразить оставшиеся. В Scala все эти сведения можно отлично записать трэйтом со следующим определением:

```
trait Ord{
  def < (that:Any):Boolean
  def<=(that:Any):Boolean = (this < that) || (this == that)
  def> (that:Any):Boolean = !(this <= that)
  def>=(that:Any):Boolean = !(this < that)
}
```

Это определение создаёт новый тип Ord, который играет туже роль как Java Comparable интерфейс и реализацию по умолчанию трёх предикатов, в термина четвёртого обстрактного. Предикаты для равенства и неравенства не появились здесь т.к. они есть по умолчанию у всех объектов.

Тип Any который использовался выше - это тип, который является супертипом всех других объектов в Scala. Он может рассматриваться как более общая версия Object из Java, т.к. он также супертип основных типов таких как Int, Float, и т. д.

Исходя из этого, чтобы сделать объекты класса сравнимыми, достаточно чтобы

определить предикаты которые проверяют на равенство и меньше и внедрить приведённый выше Ord класс. Как пример, давайте определим класс Date представляющий даты в Григорианском календаре. Такие даты состоят из дня, месяца и года, которые представляюся целыми числами. Таким образом, мы начнём так:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d
  override def toString(): String = year + "-" + month + "-" + day
```

Важная часть здесь - объявление extends Ord, которая следует за именем класса и параметрами. Это декларирует что класс Date наследуется от трэйта Ord.

Далее мы переопределим метод equals, унаследованный от Object, так, чтобы он корректно сравнивал даты сравнивая их индивидуальные поля. Реализация по умолчанию метода equals непригодна, потому что она сравнивает объекты физически. Мы пришли к следующему определению:

```
override def equals(that: Any): Boolean =
  that.isInstanceOf[Date] && {
    val o = that.asInstanceOf[Date]
    o.day == day && o.month == month && o.year == year
  }
```

Этот метод использует предопределённые методы isInstanceOf и asInstanceOf. Первый из них, isInstanceOf, соответствует оператору Java's instanceof и возвращает true если, и только если, объект к которому он применён, является экземпляром данного типа. Второй из них
asInstanceOf соответствует оператору приведения типа в Java: если объект экземпляр данного типа, он виден таким, иначе кидается ClassCastException.

Наконец, последний методы для определения это предикат который проверяет на "меньше". Он использует другой предопределённый метод, error, который кидает исключение с данным сообщение ошибки.

```
def <(that: Any): Boolean = {
 if (!that.isInstanceOf[Date])
 error("cannot compare " + that + " and a Date")
 val o = that.asInstanceOf[Date]
    (year < o.year) ||
    (year == o.year && (month < o.month ||
    (month == o.month && day < o.day)))
}
```

Это заканчивает определение класса Date. Экземпляры этого класса могут быть видны как даты или как сравнимые объекты. Кроме того, они все определяют шесть предикатов сравнения упомянутые раньше: equals и < потому что они определены прямо в определении класса Date, и остальные потому что они унаследованы от трэйта Ord. Конечно, трэйты полезны в других ситуациях, но дальнейшее обсуждение применения вне области этого документа.

8 Генерики (Genericity)

Последняя характеристика scala которую мы исследуем в этом руководстве - генерики. Java программисты должны быть в курсе проблем которые исправлены в in Java 1.5.

Генерики - это возможность писать параметризованный типами код,.например, программист пишущий библиотеку для связанных списков встречается с проблемой решения какой тип дать элементам списка. Т.к. этот список предназначен использоваться во многих разных контекстах, невозможно решить какой тип элементов должен быть, к примеру Int. Это будет слишком прозвольным и поэтому черечур ограничивающим.
Java программисты прибигают к использованию Object, который супертип всех объектов. Это решение, тем не менее, далеко от идеала т.к. не работает для/ базовых типов (int,long, float, и т.д.) и подразумевает немалое количество динамического приведения типов, которое пишется программистом.
Scala делает возможным определить генерик классы (и методы) для того чтобы решить эту проблему. Давайте рассмотрим это на примере как можно простого контейнера классов: ссылка , которая может быть пустой или указывать на объект какого-нибудь типа.

```
class Reference[T] {
 private var contents: T = _
 def set(value: T) { contents = value }
 def get: T = contents
}
```

Класс Reference параметризован типом, называемым T, который тип его элемента. Этот тип используется в теле класса как тип содержащейся переменной, аргумент set метода, и тип возвращаемого результата в get методе.
Приведённый код примера вводит переменые на Scala, которая не требует дальнейшего объяснения. Тем не менее, интересно посмотреть, что начальное значение этой переменной _,которая представляет значение по умолчанию. Значение по умолчанию 0 для числовых типов, false для Boolean типа, () для типа Unit и null для всех объектных типов.
Чтобы использовать класс Reference нужно указать, какой тип использовать для параметра типа T, который содержится в Reference. Например для того чтобы создать и использовать Reference содержащую целое число нужно написать так:

```
object IntegerReference {
   def main(args: Array[String]) {
   val cell = new Reference[Int]
   cell.set(13)
   println("Reference contains the half of " + (cell.get * 2))
 }
}
```

Как мы видели в этом примере, не обязательно приводить тип возвращаемого значения метода get перед использованием как Integer. Также нельзя сохранить в этой конкретной ячейке что-нибудь отличное от целого числа, т.к. она декларирована как содержащая целое число.

9 Заключение

Этот документ дал быстрый обзор языка Scala и представил нескольно простых примеров. Интересующие читатели могут продолжить далее чтение сопровождающего документа **Scala By Example**, который содержит больше расширенных примеров, и и консультироваться в Scala Language Specification когда это нужно.