Программирование на Scala

для любопытных

Содержание

Введение

Язык программирования Scala реализован на базе Java и включает практически всю его функциональность, в частности, доступна библиотека Java. Похоже, что многое заимствовано из такого замечательного языка, как Ruby, хотя об этом нигде прямо не упоминается. Реализованы в Scala и оригинальные идеи. Язык заслуживает самой высокой оценки и представляет несомненный интерес для каждого программиста, опытного или начинающего. Scala относится к объектно- ориентированным языкам (ООП), но, как и Ruby, его можно использовать и при программировании в

процедурном стиле. Кроме того, он обладает возможностями функционального языка настолько полно, что имеется литература по изучению функционального стиля программирования на базе Scala. Язык имеет строгую типизацию, статическую — проверка типов на этапе компиляции. Система типов высокоразвитая и её можно отнести, пожалуй, к наиболее сложной среди всех современных языков. Практически можно запрограммировать всё, что угодно, используя только основные и простые средства языка Scala. Но, если освоить и применять все его «тонкости», то можно составлять очень экзотичные программы с замысловатым кодом. Существует довольно богатая литература по программированию на Scala, есть и русские переводы, но обычно это тяжеловесные книги с пространными рассуждениями и с педантичным описанием мельчайших деталей, иногда совершенно несущественных. Кстати, этот недостаток и вообще характерен для книг по программированию. Я намерен попытаться изложить тему «своими словами», без излишних подробностей, ведь иногда одна только строчка программного кода способна заменить целую страницу нудных рассуждений.

Глава 1 Начало работы

Для установки Scala на компьютер надо скачать и запустить установочный файл, после чего можно работать с помощью командной строки. Для создания исходного текста подходит любой редактор. Мне, например, очень нравится Geany, имеющий подсветку синтаксиса и позволяющий управлять компиляцией и запуском программ на Scala.

Scala предлагает три способа выполнения программ:
1.Интерактивный режим (Repl) из командной строки.
2.Запуск Scala-скриптов, записанных в отдельных файлах, в интерпретаторе.
3.Компиляция программы в class-файл, подобно тому, как это делается в Java.

Поскольку в любом, даже малом фрагменте программы на Scala требуется задавать типы, сразу покажу, как это делается в простейших случаях; дальше мы рассмотрим это детально.

x:Int – переменная ***x*** имеет тип ***Int*** (целое число)
a:List[String] – ***a*** представляет список, элементы которого имеют тип String (строка). В квадратных скобках задаётся так называемый параметр типа, мы ещё вернёмся к нему позже, а пока это надо просто запомнить.
f(x:Double):Double – функция ***f*** принимает аргумент типа ***Double*** и возвращает результат типа ***Double***. На Haskell это выглядело бы так:
f :: Double → Double
Ну, и так далее.

1.1 Интерактивный Scala (Repl)

Для старта Repl надо в командной строке напечатать команду: ***scala***. Появится приглашение *scala>*. Теперь можно печатать строки программного кода, которые будут немедленно выполняться с выдачей результата:
*scala>**1 + 1***
ответ будет выведен в виде:
res0:Int = 2
Это значит, что полученный результат получил идентификатор ***res0***, его можно будет использовать в следующих строчках кода, как обычную переменную. Тип этой переменной ***Int***, а её значение равно ***2***. Тип результата Repl указывает всегда.
*scala>**res0 * 8 → res1:Int = 16*** (дальше всегда вместо слов «получим» или «будет равно», и тому подобное, будем для краткости пользоваться такой стрелкой ***(→)*** ; при копировании кода примеров для выполнения этот текст вместе со стрелкой надо удалить; не будем впредь также писать приглашение *scala>)*.
val x = "Hello World" → x:String = Hello World
Теперь ***x*** имеет тип ***String*** – строковая переменная, её возможное значение представлено набором произвольных знаков, заключённых в кавычки. Таким образом, Scala способен определять (выводить) тип переменной по значению.
Идентификатор ***x***, задан нами. Стандартное (зарезервированное, ключевое) слово ***val*** указывает, что ***x*** неизменяемая *(immutable)* переменная, дальше мы не можем присвоить ей другое значение, например, написать ***x = “David”***, будет сообщение об ошибке. При этом можно сколько угодно заново объявлять переменную ***x:***

val x = “pattern” → x:String = pattern
и даже можно объявить новую переменную ***x*** изменив её тип:
val x = 45.33 → x:Double = 45.33
Аналогично объявляются переменные со словом ***var***:
var x = "Привет, мир!"
var x1 = x.length → x1:Int = 12
Запись ***x.length*** означает, что для переменной ***x*** вызван стандартный (встроенный в язык) метод ***length****;* иначе можно сказать, что вызвана стандартная функция ***length***, которой передано в качестве аргумента значение переменной ***x***. Метод ***length*** умеет определять
длину (количество знаков) переменной типа *String*. Ключевое слово ***var*** указывает, что ***x*** и ***x1*** – изменяемые переменные *(mutable),* дальше мы можем писать, например, ***x1 = 125***. Нельзя, однако, написать ***x1 = "Hello"***, тип можно изменить только объявив переменную заново. В одном объявлении можно инициировать сразу несколько переменных:
val x, y, z = 10 → x = 10, y = 10, z = 10

Отметим попутно, что точка применяется в двух различных случаях:
1) ***x.length*** – вызов метода ***length*** для переменной ***x***,
2) ***p.f()*** - вызов метода ***f()***, принадлежащего объекту ***p***.

Во всех примерах выше мы использовали способность Scala автоматически определять (выводить) тип по результату. Можно, однако, задавать тип принудительно:
val x:Double = 77 → x: Double = 77.0
Переменная ***x*** получила тип ***Double***, хотя справа от знака равенства целое число. Таким образом, Scala изменила тип согласно предписанию. Но это возможно не всегда. Так, если мы напишем:
val x:Double = "82.65",
то получим сообщение об ошибке:
error: type mismatch; - не соответствует тип
found : String("82.65") - получили значение типа String
required: Double - а требовался тип Double
val x:Double = "82.65"- ошибочный текст выводится повторно
Тип ***String*** Scala не может автоматически трансформировать в ***Double***, это придётся делать с помощью специального метода

***toDouble**:*
val x:String = "82.65"
val y = x.toDouble → y: Double = 82.65

Кроме стандартных функций можно использовать библиотечные функции языка Java. Запросить доступ в эту библиотеку можно командой
import java.util._ → import java.util._ (_ - знак подчёркивания, о нём позже)
Этот ответ подтверждает, что доступ получен и теперь мы можем применять разные средства библиотеки, например:
val d = new Date → d:java.util Date = Tue Apr 19 13:05:52 MSK 2016
Переменная ***d*** получила специальный тип. Я думаю, тут всё ясно без слов.

Для того, чтобы выйти из интерактивного режима надо напечатать команду ***:quit*** или просто ***:q***.

1.2 Скрипты Scala

Если программа имеет размер в несколько строк и более, использовать режим Repl становится неудобно и тогда лучше создать скрипт, представляющий собой отдельный файл с записанным в нём программным кодом (программисты обычно называют его исходной *(source)* программой). В некоторых языках программирования в скрипте должен быть объявлен метод со стандартным именем ***main***, определяющий точку входа в программу. На Scala это не всегда обязательно; без метода ***main*** выполняться программа будет строчка за строчкой сверху вниз (как на Ruby). Код программы надо напечатать на каком-либо текстовом редакторе и записать в файл с произвольным именем, например ***prob.scala*** (расширение имени - ***scala*** обязательно). Для выполнения программы в интерпретаторе надо в командной строке перейти в ту папку, где расположен файл и напечатать команду ***scala prob.scala***.

При использовании *Geany* скрипт можно запускать прямо из редактора. Возможно, что в этом случае окно командной строки после вывода результата будет сразу закрываться. Чтобы избежать этого, в самом конце запрограммируйте строку ввода, например:
val xxx = io.StdIn.readLine("Нажмите клавишу Enter")

При составлении примеров программ бывает нужно вводить какие-то данные с клавиатуры. Не вдаваясь в подробности сразу укажу, что для ввода текста надо выполнить команду:
val x = io.StdIn.readLine
В этом месте ваша программа остановится и будет ждать ввода. Можно, например, ввести число ***23.087***. Тогда переменная ***x*** станет равна этому числу, но представленному в текстовой форме (тип *String*). Для перевода в тип *Double* придётся опять воспользоваться методом ***toDouble***:
val y = x.toDouble
Можно программировать и так:
val x = io.StdIn.readLine.toDouble
А так можно ввести с клавиатуры список (один элемент списка):
val x:List[String] = List(io.StdIn.readLine)
Чтобы ввести список со многими элементами, придётся организовать цикл; но об этом позже.
readLine позволяет задать приглашение для ввода:
val x = io.StdIn.readLine("Ваше имя? ")
На экране появится приглашение: ***Ваше имя?*** И программа будет ждать ввода с клавиатуры. Кроме ***readLine*** можно также использовать ***readInt, readDouble*** и так далее, и тогда не надо заботиться о дополнительном преобразовании типов.

1.3 Первая программа

Традиционная Hello World программа на Scala предельно краткая:
pruntln(“Hello World!”)
Оператор ***println()*** - сокращённая запись для ***System.out.println()*** - модуля из библиотеки Scala, загружаемого каждой Scala — программой, об этом не надо заботиться. ***println()*** выводит на экран аргумент, заданный в скобках. Если аргумент не текстовый, например, целое число, то преобразование в текст происходит автоматически. Если внутри какого-то текста надо вывести какие-то результаты вычислений, то на Ruby, например, применяется так называемая интерполяция. Scala избавляет нас от такой необходимости и выполняет эту «интерполяцию» автоматически:
println("argsin(0.7) = " + math.asin(0.7) + " радиан") →
argsin(0.7) = 0.775397496610753 радиан

В этом контексте знак плюс является знаком конкатенации — объединяет строковые литералы в одну строку. Можно где угодно присоединять к тексту данные любого типа при помощи знака конкатенации (+). Scala выполнит перевод этих данных в текст (интерполяцию) автоматически. Например:
val x = "Foo" + 2.5 → x: String = Foo2.5
Запись ***math.asin(0.7)*** означает, что для числа ***0.7*** вызывается метод (функция) ***asin***, принадлежащая библиотеке ***math*** (можно писать ***Math***). Кстати, термины «метод» и «функция» в общем-то равнозначны, но в Scala имеется одна важная деталь, в контексте которой это разные вещи; позже мы рассмотрим это.

Можно, конечно, программировать и так:
val x = "argsin(0.7) = " + " " + math.asin(0.7) + " radian" → x имеет mun String
print(x) → argsin(0.7) = 0.7753974966107530 radian

После выведенных данных оператор ***println()*** выполняет перевод на новую строку. Есть также и оператор ***print()***, который на новую строку не переводит и, значит, следующий вывод будет в той же строке. О форматированном выводе поговорим позже.

Программа, печатающая целые числа в цикле может выглядеть так:
for { i <- 1 to 10 } print(i + " ") → 1 2 3 ...10
Оператор цикла ***for*** на *Scala* имеет очень большие возможности, дальше мы их рассмотрим подробно. Выражение в фигурных скобках означает, что переменная цикла ***i*** последовательно принимает значения от ***1*** до ***10*** (слово ***to*** стандартное*)* . Вместо фигурных скобок здесь можно применить и круглые. (Обратите внимание на аргумент оператора ***print*** – к целому числу присоединяется пробел*).*

А вот так может выглядеть вложенный цикл:
for {i <- 1 to 3; j <- 1 to 3} print(i * j + " ") → 1 2 3 2 4 6 3 6 9

1.4 Компилирование Scala – программ

Выполнение скриптов из командной строки с помощью команды типа ***scala prob.scala*** использует режим интерпретатора. Но программы Scala можно и компилировать для получения файла с неким промежуточным кодом, что даёт известные преимущества: скорость выполнения, сокрытие исходного кода программы и так

далее. Иногда компиляция вообще необходима. Файлы Scala компилируются аналогично программам на Java. Программы, позволяющие компиляцию, имеют некоторые особенности. В частности, они уже не могут быть в процедурном (императивном) стиле, как те примеры, что были выше, а должны быть ООП. В результате компиляции создаётся файл типа JVM – class, а имя файла совпадает с именем класса. Объясняется это тем, что используются средства языка Java. Компилятор требует, чтобы в исходном файле был объявлен хотя бы один класс, трейт или объект. Компиляция выполняется командой:
scalac prob.scala (обращаю внимание: не ***scala***, а ***scalac***).
Одной командой можно откомпилировать более одного файла:
scalac prob1.scala prob2.scala
Есть также возможность применять «быстрый» компилятор ***fsc***:
fsc prob.scala. ***fsc*** компилятор после окончания компиляции указанного файла ожидает команды на компиляцию другого файла. Достигается некоторая экономия времени.

В программе для компиляции могут быть какие угодно классы и должен присутствовать так называемый объект содержащий метод ***main***, объявленный так, как в следующем примере (если что-то пока непонятно, не обращайте внимания, позже мы во всём разберёмся):

```
class A  {
        def f(x:Int) = x * x
}
object B  {
        def main(args:Array[String]) {
        val p = new A
        println(p.f(5))  → 25
        }
}
```

Метод ***main*** определяет точку входа в программу и позволяет вводить аргументы из командной строки (в примере эта возможность не использована). Аргумент метода ***main*** должен быть указан так, как в примере и никак иначе. Здесь ***args*** — массив с элементами типа ***String*** (идентификатор массива может быть любым). Файл ***prob.scala*** надо скомпилировать командой
scalac prob.scala

в результате чего будут созданы (в той же директории) файлы под именами, совпадающими с именами классов и объектов программы. Запустить на выполнение надо файл с именем объекта, содержащего метод ***main***. У нас это будет файл с именем ***B***. Для запуска надо выполнить команду:

scala B

Сразу же получим результаты.

Покажем, как можно использовать аргумент метода *main*:

```
object Ob {
    def main(a:Array[String]) = {print(a(0) + " " + a(1))
    }
}
```

Программу с этим текстом компилируем и запускаем *class*-файл такой командой:

scala Ob "x = " 12.5 → x = 12.5

Массив мы обозначили буквой ***a***, тогда в теле функции ***main*** можно вызывать элементы этого массива, используя индекс в круглых скобках: ***a(0), a(1)*** и так далее. Значения элементов задаём в командной строке сразу после имени файла, отделяя друг от друга пробелом. Кстати, элемент ***a(1)*** задан числом с плавающей запятой, компилятор сам трансформировал его в строку.

Имеется возможность не создавать явно служебный метод ***main***, а использовать некий специальный трейт ***App***, который сделает за нас всё, что надо и такой способ считается предпочтительнее. В этом случае файл ***prob.scala*** должен иметь вид:

```
class A {
    def f(x:Int) = x * x
}
object B extends App {
    val p = new A
    println(p.f(5)) → 25
}
```

Служебное слово ***extends*** указывает, что объект ***B*** наследует трейту ***App***. Все действия по компиляции и запуску остаются прежними. Можно применять и аргументы командной строки, только идентификатор массива теперь всегда будет ***args.***

Глава 2 Базовый синтаксис

Как и в языке Java, многие синтаксические конструкции языка C присутствуют в Scala. Есть, разумеется, и существенные различия. Например, в Scala не обязательно, хотя и допустимо, заканчивать строки кода точкой с запятой. Отдельные выражения, расположенные на одной строке, необходимо отделять друг от друга точкой с запятой.

2.1 Комментарии

Комментарий, располагающийся на одной строке должен начинаться с двух прямых слэшей (//). С этого знака и до конца строки весь текст игнорируется компилятором. Комментарии, занимающие несколько строк ограничиваются знаками: */……./*.

Замечание: не следует, по моему мнению, слишком увлекаться этими комментариями, а писать их только, когда они действительно необходимы. Вообще, надо стремиться к максимальной краткости и лаконичности текста. В любом учебнике по программированию советуют, например, идентификаторы всех переменных, методов, объектов, классов и так далее выбирать такими, чтобы это были реальные имена тех величин, о которых идёт речь. Например, если в программе присутствует такая величина, как скорость автомобиля, то идентификатор для неё должен быть что-нибудь вроде *rateCar* или *rate-car* и тому подобное. Какой-то смысл в этом есть, поскольку всегда будешь знать, о чём идёт речь. Очевидно, что основные программные объекты, предназначенные для многократного пользования, надо называть собственными уникальными именами: модули, классы, объекты и так далее. Однако, длинные слова загромождают программу и в ней, наоборот, становится труднее ориентироваться. А если программист слабо знает английский язык, то такая манера и вообще не имеет никакого смысла. Когда я вижу, например, такое объявление класса:
class OptionalUserProfileInfo { … }
мне становится как-то скучновато. Я бы советовал никогда не выбирать длинных идентификаторов, а скорость автомобиля обозначать одной буквой *v*, ну, или если уж важно указать, что это скорость именно автомобиля, то — *va*. Самое главное — стремиться к тому, чтобы в программе была ясной структура и логика алгоритма, и чем меньше текста, тем лучше.

2.2 Базовые типы

Я уже упоминал, что система типов в Scala весьма многообразная и сложная. Но основные базовые типы практически те же, что и в Java. Не приводя подробных комментариев просто перечислим их с примерами:

Int: ***1, 882, -5*** (есть также и тип ***Integer***, почти тоже самое)
Boolean: true, false – всего только два значения.
Float: ***78.3, 34f*** (или ***F***)
Double: ***1.0, 54d, 0.78e3*** (можно ***D*** и ***E***) — длинные с плавающей запятой. По умолчанию Scala использует тип ***Double***, а не *Float*, как во многих других языках.
Long: ***4234876543267823 5L*** (или ***l***) — длинные целые
BigInt: ***536823460*** – большие (но меньше, чем ***L***) целые числа.
Char: ***'4', '?', 'z'***
String: ***"Anna"***
Scala также поддерживает много-строчные строковые литералы, заключаемые в тройные кавычки:
“““Hello
multiline
world”””
Есть ещё и другие базовые типы, но нам пока достаточно этих. А так могут выглядеть *XML* выражения с вложенным (или встроенным) Scala кодом:
Foo
***
{(1 to 3).map(i => {i})}
***
(Веб-программирование на Scala не будем рассматривать в этой книге, это отдельная большая тема).

2.3 Классы

В файле Scala можно объявлять любое число классов. Все они по умолчанию имеют общий доступ ***(public)***; фактически в Scala по умолчанию всё ***public***. Для объявления применяется ключевое слово ***class***, после которого надо указать имя класса с заглавной буквы. Дальше в круглых скобках может быть список аргументов; если их нет, то скобки ставить не требуется (хотя допустимо). Тело класса заключается в фигурные скобки.

```
class Prob   {
      println("В Scala нет проблем для работы с кириллицей")
}
```

val p = new Prob → сразу будет выведен указанный текст
Мы создали экземпляр (объект) класса с идентификатором ***p***. Для этой цели применяется стандартный метод ***new***. Все операторы в классе немедленно выполняются в момент создания экземпляра.

Иногда класс не имеет тела:
class Foo
Или с аргументом:
class Foo(name:String)
Указывать тип аргумента требуется обязательно. Теперь создать экземпляр можно так:
val q = new Foo("Peter")
Объявим класс ***Baz*** с аргументом ***name*** и запрограммируем проверку аргумента с выдачей сообщения при наличии ошибки:

```
class Baz(name: String) {
        if (name == null) throw new Exception("Name is null")
}
val p = new Baz(null) → java.lang.Exception: Name is null
```

То-есть, если значение аргумента равно специальному значению ***null***, в момент создания экземпляра будет инициировано исключение (предупреждение об ошибке). Организация подобных исключений — отдельная тема. Директива ***throw new Exception*** стандартная, заимствована из Java. Текст предупреждения можно задать любым. Очевидно, что подобным образом можно запрограммировать самые разнообразные проверки.

В языке Ruby аргументы класса использовались бы для инициации так называемых переменных экземпляра. Для этого там применяется специальный конструктор. В Scala конструктор тоже имеется, но он создаётся автоматически, нам не надо о нём заботиться. О конструкторах в Scala можно найти подробную информацию в руководствах, но на практике это понятие можно вообще не использовать.

В Scala всё, исключая методы, является экземплярами классов. Поскольку примитивные типы - представители классов, то их имена тоже начинаются с заглавной буквы: ***Int, Long, Double, Float, Boolean, Char, Short, Byte***. Все они подклассы (дочерние классы) класса ***AnyVal***. (Собственно говоря, в Scala понятия класс и тип фактически равнозначны — как только мы создаём класс, сразу получаем соответствующий тип). Есть также базовый тип ***Unit***, соответствующий ***void*** в других языках. Если инициировать переменную так:
val v = (), то она будет иметь тип ***Unit***. Есть методы, возвращающие ***Unit***:

val x = print(“Hello”) → x:Unit = ()

Этот метод выводит на печать текст, но ничего явно не возвращает.

Имеется ещё тип ***Any*** – некоторый заменитель всех реальных типов и коренной тип в классовой иерархии Scala. В неопределённой ситуации мы можем присвоить переменной тип *Any* и тогда в контексте эта переменная может принять любой тип, например, ***Int***.

val x:Any = 12 → x:Any = 12

val s:Any = "Hello" → s: Any = Hello

Нет возможности тип ***Any*** сразу преобразовать в какой-либо другой тип, кроме как в ***String*** с помощью метода ***toString***:

val y = x.toString → y:String = 12

Теперь можно получить другие типы:

val z = y.toInt → z:Int = 12

val z = y.toDouble → z:Double = 12.0

Можно, конечно, делать и так:

val z = x.toString.toInt → z:Int = 12

Имеется также метод ***asInstanceOf***, применяемый для изменения типов. Он способен трансформировать в том числе и тип ***An****y*:

val x:Any = 55

val y = x.asInstanceOf[Int] → y: Int = 55

Нужный нам тип указывается в квадратных скобках (параметр типа). Похожий метод ***isInstanceOf*** позволяет проверить тип переменной:

y.isInstanceOf[Double] → false

y.isInstanceOf[Int] → true

В Scala есть разновидности классов: абстрактный класс, *case*-класс; мы встретимся с ними по ходу дела.

Можно внутри классов создавать другие (вложенные) классы без ограничения глубины. Полезно это, прежде всего, для разграничения областей видимости (для исключения конфликтов имён). Покажем на примере:

```
class A  {
        def f(x:Int)=x
        class B {
                def f(x:Int) = x*x
        }
        val r = new B
```

```
}
val p = new A
println(p.f(3)) → 3
print(p.r.f(7)) → 49
```

В классе ***A*** и во вложенном классе ***B*** объявлены методы под одним и тем же именем ***f***. При вызове их из внешней области конфликта имён не возникает, так как приходится применять полные (квалифицированные) имена; у нас: ***p.f(3)*** и ***p.r.f(7)***.

Можно создавать экземпляры вложенных классов и так:

```
class A {
        def f(x:Int)=print(x)
        class B{
                def f(x:Int)=x*x
        }
}
val p = new A
val r = new p.B → экземпляр вложенного класса
print(r.f(7)) → 49
```

Метод ***getClass*** позволяет узнать, какому классу принадлежит данный субъект:

```
25.getClass →x:Class[Int] = int
3.045.getClass → y:Class[Double] = double
```

2.4 Объекты (синглетон, singleton)

Можно создавать много экземпляров класса (иначе их ещё называют объектами) с помощью метода ***new***. Кроме того, Scala позволяет создавать некие индивидуальные экземпляры, которые тоже называются объектами, или синглетонами (singleton). Они похожи на классы, только не могут иметь аргументов. Вместо слова ***class*** используется слово ***object***:

```
object Ob {
        val x = 2.5
        def f(a:Int) = a * a
}
print(Ob.x) → 2.5
print(Ob.f(7)) → 49
```

В теле объекта мы создали поле ***x***, и метод ***f,*** на которые можно ссылаться, используя имя объекта вместо имени экземпляра для

класса. Ранее мы уже использовали синглетон в примере компилируемой программы с методом ***main***.

Синглетоны могут также применяться в качестве объектов-компаньонов для классов. Чтобы создать объект-компаньон надо объявить синглетон с тем же именем, что и у класса. Размещаться они должны в одном файле с исходным кодом. Класс и его компаньон могут обращаться к полям и методам друг друга. В приводимом примере класс *A* имеет объект-компаньон *A*:

```
class A  {
        val id = A.n()  → n() - метод компаньона A
        var b = 1.0
        def f(a:Double) {b += a}  → f() - метод класса A
}
object A {
        var x = 0
        def n() = {x += 1; x }
}
val p = new A
println(p.id) → 1
p.f(2.3)
print(p.b) → 3.3
```

Обратите внимание, что метод ***f*** не имеет знака равенства перед телом метода. При таком варианте метод ничего не возвращает (точнее возвращает тип ***Unit***), метод ***f*** только меняет значение переменной ***b***. В функциональном программировании сказали бы, что метод ***f*** вызывается только ради побочного эффекта.

Посмотрите также на строчку

```
val id = A.n()
```

Здесь мы переменную ***id*** приравняли методу ***n()*** из синглетона ***A***. Позже мы узнаем, как и почему это возможно.

Синглетоны в Scala используются очень широко и с большой пользой, хотя по существу они почти не отличаются от обычных экземпляров класса. Мне пока не очень ясно, что могут дать объекты-компаньоны сверх того, что позволяют делать трейты.

2.5 Трейты (trait)

Кроме классов и синглетонов Scala позволяет создавать так называемые трейты *(trait)* аналогичные интерфейсам в Java или

миксинам в Ruby. Программный код из трейта может многократно использоваться, для уменьшения дублирования кода. В Ruby применяется термин «подмешивание» миксина в разрабатываемую программу. Можно использовать этот термин и применительно к трейтам. Объявляется трейт аналогично классу, но с ключевым словом ***trait***:

trait Dog {тело трейта}

Для использования трейта ***Dog*** в классе ***Fizz*** применяется такой синтаксис:

class Fizz(name:String) extends Dog

Теперь можно в классе ***Fizz*** использовать весь код из трейта ***Dog***. Слово ***extends*** ключевое, оно же применяется и при наследовании классов (об этом позже). Класс в Scala может наследовать только одному суперклассу (нет множественного наследования), но может наследовать произвольному числу трейтов. Если есть наследование одновременно классу и трейту, или нескольким трейтам, то применяется ещё одно ключевое слово: ***with:***

class Fizz(name:String) extends Bar(name) with Dog

Определим трейт ***Tr***, содержащий метод:

```
trait Tr {
    def f(x:Double, y:Double):Double = x * y
}
```

Мы объявили в теле трейта метод ***f*** , который принимает два аргумента ***x*** и ***y*** типа ***Double*** и возвращает результат тоже типа ***Double***. Покажем, как можно использовать трейт ***Tr***:

```
class Foo extends Tr  {
    def fi() = 4
}
val p = new Foo
println(p.fi()) → 4
print(p.f(3.0, 7.0)) → 21.0
```

В классе *Foo* мы объявили метод ***fi***, не имеющий аргументов (пустые скобки можно было опустить). Тип возвращаемого результата ***Int*** мы не указали, программа сама выводит его. В созданном нами экземпляре класса ***p*** доступны как метод самого класса - ***fi***, так и метод, унаследованный из трейта ***Tr - f.***

Приведём ещё один пример (искусственный) с более сложной схемой связи между классами, трейтами и объектами. Служебное

слово ***override*** используется для переопределения (перегрузки) методов.

```
trait Cat {
                    def meow(): String
}
trait Fuzzy extends Cat {
                    override def meow(): String = "Meeeeeeow"
}
class Yep extends Fuzzy
object Dude extends Yep {
                    override def meow() = "Dude looks like a cat"
}
object OtherDude extends Yep {
                    def two(x: Yep) = meow() + ", " + x.meow
}
 print(OtherDude.two(Dude)) → meeeeeow, Dude looks a cat
```

Обратите внимание на то, что метод ***two***, объявленный в объекте ***OtherDude***, принимает переменную ***x*** типа ***Yep***. Следовательно, создавая класс, мы, как указывали выше, создаём новый тип данных, который может применяться наравне с базовыми типами. Параметру метода ***two x*** мы передаём значение ***Dude***, то-есть синглетон вместе с методом ***meow***, кстати, перегруженным.

Попробуйте самостоятельно проанализировать этот код. Конечно, на практике такие сложные схемы связей встречаются не часто.

Трейты тоже могут наследовать, как другим трейтам, так и классам. При этом такие классы становятся суперклассами для всех классов, в которые подмешивается данный трейт. Попробуйте разные комбинации наследования самостоятельно.

С помощью ключевого слова ***final*** класс можно объявить финальным и тогда его уже нельзя будет унаследовать. Финальными можно объявлять также поля и методы, чтобы их нельзя было переопределять.

2.6 Пакеты

Пакеты — это именованные модули кода, они предназначены для управления пространствами имён в больших программах. Имя пакета может быть произвольным. Объявляются пакеты в самой

первой строке кода модуля с помощью ключевого слова ***package:***
package foo
Обычно пакет располагается в отдельном файле, но можно и в один файл поместить несколько пакетов. Кроме того, пакеты могут быть вложенными и можно даже создавать цепочки пакетов. Конфликтность имён исключается за счёт использования полных квалифицированных имён. Пакеты содержат классы, объекты и трейты, но в них не может быть полей и методов.

Пакеты могут быть импортированы с помощью ключевого слова ***import***, после чего на них можно ссылаться в текущей области имён.
import foo._
Эта команда импортирует весь контент пакета ***foo,*** на что указывает групповой символ ***(_)***. Вместо него можно указывать конкретные имена импортируемых классов, объектов, трейтов, если не нужен весь код пакета:
import math.sin → будет загружен только модуль ***sin*** из пакета ***math***
Можно импортировать сразу больше, чем один модуль и тогда их список надо перечислить через запятую в фигурных скобках.
import math.{sin, cos} → загружены ***sin, cos***
Одновременно можно изменить имя импортируемого модуля:
import math.{sin => sinus} → теперь можно писать: ***sinus(1.5)***
Фактически операция ***import*** служит только для укорачивания длинных имён. Написав ***import math***, дальше можем вызывать математические функции по имени: ***sin(x)*** вместо ***math.sin(x)***. Используя только длинные квалифицированные имена, можно обойтись вообще без операции ***import***.

Есть некоторые особенности компиляции при использовании пакетов. Пусть пакет находится в файле ***rab2.scala****:*

```
package pac {
    class C  {
        def pr = println("Hello")
    }
}
```

Основную программу запишем в файл ***rab.scala****:*

```
import pac._
class B {
```

```
        def f(x:Int) = x*x
}
object A  {
        def main(args:Array[String]) {
                val p = new B
                println(p.f(6)) → 36
                val q = new C
                q.pr → Hello
        }
}
```

Скомпилируем оба файла:

scalac rab2.scala rab.scala

После этого, как обычно, на выполнение надо запускать файл с именем объекта, содержащего метод ***main****:*

scala A

Инструкция ***import*** может располагаться в любом месте, не только в начале файла. Тогда область видимости импортируемого пакета простирается только до конца блока, в котором расположена инструкция ***import***. Это позволяет уменьшать количество одновременно импортируемых имён и уменьшать вероятность конфликта имён.

2.7 Объявление метода

Мы уже неоднократно объявляли методы в рассмотренных примерах: ключевое слово ***def***, имя метода, список аргументов в круглых скобках с обязательным указанием их типов, тип возвращаемого методом результата (не всегда обязательно), знак равенства и тело метода в фигурных скобках. Тип результата можно не объявлять, если он может быть однозначно выведен программой из контекста. Способность к выводу (определению) типов есть мощное и полезное свойство Scala, но при его использовании следует проявлять осторожность и всегда указывать тип явно, если есть какая-то неопределённость. Посмотрим на такой пример:

```
def f(a:Int, b:Boolean):String = if (b) a.toString else "Ложь"
print(f(23, true)) → 23
```

Функция ***f*** принимает аргументы ***a*** и ***b***, имеющие тип ***Int*** и ***Boolean*** соответственно, а возвращает результат типа ***String***. В теле

функции мы применили условный оператор ***if – else***, позже рассмотрим его более подробно. Если для аргумента ***b*** задать значение ***true***, переменная ***a*** стандартным методом ***toString*** будет приведена к типу ***String***, а если — ***false***, то будет возвращён текст ***”Ложь”***, то-есть в любом случае получим результат типа ***String***.

Имеется возможность при объявлении метода указывать, что аргументы или результат имеют неопределённый тип ***Any***:

def f[Any](p: Any): List[Any] = p :: Nil

Метод ***f*** принимает один аргумент типа ***Any*** и возвращает список с элементами типа ***Any***. В правой части использован оператор ***::*** (фактически это тоже метод), который добавляет элемент ***p*** к списку в его начало. Вместо списка стоит специальное значение ***Nil***, представляющее в данном контексте пустой список (вместо ***Nil*** можно явно указать ***List()***). Для того, чтобы метод ***f*** был способен определять (выводить) конкретный тип результата из контекста вместо неопределённого ***Any***, после имени метода надо поставить ***[Any]*** (называется параметр типа). Без этого результат так и будет иметь тип ***Any***. В итоге метод ***f*** даёт такие результаты:

f(56) → List[Int] = List(56) - (тип ***Int)***

f("Bob") → List[String] = List(Bob) - (тип ***String)***

f(true) → List[Boolean] = List(true) - (тип ***Boolean)***

f(43, "Peter", false) → List[(Int, String, Boolean)] = List((43, Peter, false))

Значит, метод ***f*** действительно принимает и выводит любые типы.

Посмотрим, что будет, если не указать параметр типа:

def f(p: Any): List[Any] = p :: Nil

f(43, "Peter", false) → List[Any] = List((43,Peter,false))

Теперь все элементы списка имеют неопределённый тип ***Any***. Неопределённый тип можно задать также применив произвольный идентификатор типа; например, часто используют обозначение ***T***. Отличие его от ***Any*** в том, что ***Any*** это базовый тип языка — супертип для всех других типов, в то время, как ***T*** – просто неопределённый тип.

Теперь проанализируйте самостоятельно, как работает такой метод:

def f[T](p: T):T = p

или такой:

def f[T](p: T):String = p.toString

Пусть у нас есть такой простейший класс:
class A { def f(x:Double) = x * x }
Создадим экземпляр:
val p = new A
Обычно метод вызывается так:
p.f(3.0) → 9.0
Если у метода несколько аргументов, то такой способ вызова единственный. Но если аргумент только один, точку и скобки можно опускать:
p f 3.0 → 9.0
Это, конечно, синтаксический сахар, привожу только потому, что в примерах можете встретить подобный вызов — лучше быть в курсе.

У методов могут быть аргументы переменной длины — наборы с произвольным числом членов, что указывается знаком звёздочка ***(*)*** у типа аргумента. Надо только иметь ввиду, что если в списке аргументов метода несколько членов, то такие аргументы должны стоять последними. В правой части аргументы переменной длины рассматриваются, как коллекции вида ***Seq*** – частный вид массива.
def f(c: Int*): Int = c.reduceLeft((a, b) => a max b)
Этот метод принимает аргумент переменной длины ***c*** с членами типа ***Int*** и возвращает целое число. К коллекции *c* применяется стандартный метод ***reduceLeft*** – разновидность итератора, которому в круглых скобках (можно в фигурных) передаётся анонимная функция. В других языках она может называться ***lamdla, closure, proc*** или ***block*** (будем для краткости чаще использовать термин блок, хотя этим словом обозначается также и любой фрагмент кода, заключённый в фигурные скобки). Итератор последовательно передаёт блоку все элементы коллекции, присваивая их значения вспомогательным переменным ***a*** и ***b***. В блоке из каждой пары этих переменных выбирается наибольшая стандартным методом ***max*** и таким образом отыскивается максимальный элемент из набора:
f(23, 5, 74, 2, 8) → 74
Если в примере ***max*** заменить на ***min*** получим наименьший элемент.
Обратите внимание — функции передаётся не список, а набор значений в скобках. Нельзя вместо этого набора передать функции

какую-нибудь готовую коллекцию, например ***List*** или ***Range***. Есть, правда, возможность поступить, например, так:

f(1 to 5: _*) → 5

Здесь знак ***(_)*** указывает на то, что ранг (о нём дальше) надо трактовать, как набор. Без такого синтаксиса не обойтись при использовании рекурсивной функции с переменным числом аргументов:

```
def sum(a:Int*):Int = {
        if (a.length == 0) 0
        else a.head + sum(a.tail:_*)
}
print(sum(1,2,3,4)) → 10
```

Здесь к набору переменной длины ***a*** применяются методы ***length***, ***head*** и ***tail***, при этом хвост ***a.tail*** является готовой коллекцией — списком и его можно передать функции ***sum*** только таким способом. Если в этом примере пока что-то непонятно, вернитесь к нему после чтения раздела о коллекциях.

В блоке ***((a, b) => a max b)***, как и в других аналогичных случаях, явные идентификаторы переменных блока ***a*** и ***b*** Scala позволяет заменять знаками подчёркивания (_) - *placeholder* (часто его называют «групповой символ»):

```
def f(c: Int*): Int = c.reduceLeft(_ max _)
```

При этом отпадает необходимость и в знаке =>. Компилятор «знает», что надо подставлять на место этих *placeholder.*

На самом деле в Scala есть методы ***max*** и ***min***, применяемые к коллекциям:

```
val x = List(3,1,7,4,2,5).max → x: Int = 7
```

В параметрах переменной длины можно совместно использовать элементы разных типов, если указать неопределённый тип аргументов метода:

```
def s[T](a: T*): String = a.foldLeft("")((x, y) => x + y.toString)
```

Здесь мы к исходной коллекции ***a*** применили ещё один итератор - ***foldLeft***, который параметру ***x*** задаёт исходное значение, указанное, как аргумент итератора - ***("")*** (то-есть у нас пустая строка), а параметру ***y*** последовательно передаёт все элементы коллекции. Затем переменная ***y***, какой бы тип у неё не был, методом ***toString*** приводится к типу ***String*** и с помощью знака конкатенации ***(+)*** соединяется с ***x***. В результате все элементы

объединяются в одну непрерывную строку:
s(23, “Anna”, true) → 23Annatrue
Перепишем метод с применением *placeholder*:
```
 def s[T](a: T*): String = a.foldLeft("")(_ + _.toString)
```
Переменное число аргументов допустимо использовать в цикле ***for:***
```
def sum(a:Int*) = {
        var r = 0
        for (x <- a)  r += x
        r
}
print(sum(1,2,3)) → 6
```
Мы уже применяли в рассмотренных примерах перегрузку (переопределение) методов. Приведём ещё один пример:
```
 class A {  def f(x:Double, y:Double) = x + y  }
 class B extends A { override def f(x:Double, y:Double) = x * y }
val p = new A
println(p.f(3.8, 5.3)) → 9.1
val q = new B
 print(q.f(3.8, 5.3)) → 20.14
```
В дочернем классе ***B*** мы переопределили метод ***f*** из родительского класса ***A***, в результате метод вместо сложения стал выполнять умножение двух переменных типа ***Double***. Для перегрузки метода надо после ключевого слова ***override*** определить новый метод с тем же именем. Перегрузка позволяет изменять функциональность методов родительского класса без изменения его кода. Есть, однако, ограничения. Например, если мы попытаемся изменить тип аргументов метода ***f***, получим сообщение об ошибке:
class B extends A { override def f(x:Int, y:Int) = x * y } → ошибка
Нельзя изменить и количество аргументов перегружаемого метода.

Другие возможности перегрузки методов покажем на следующем примере:
```
abstract class Base {
 def thing: String
}
class One extends Base {
 def thing = "Moof"
}
```

```
class Two extends One {
 override val thing = (new java.util.Date).toString
}
class Three extends One {
 override lazy val thing = super.thing + (new java.util.Date).toString
}
val p = new Two
println(p.thing) →  Sun May 01 13:06:58 MSK 2016
val q = new Three
lazy val r = q.thing
Thread.sleep(10000)
print(r) → MoofSun May 01 13:07:08 MSK 2016
```

Во первых: при перегрузке методов абстрактного класса модификатор ***override*** можно опускать (класс ***One***). Абстрактные классы объявляются со словом ***abstract***, а если класс не имеет тела, то он всегда абстрактный, даже и без ключевого слова. Такие классы имеются в других языках и в Scala используются довольно часто.

Во вторых: методы, не имеющие аргументов и полей можно перегружать, используя ***val*** вместо ***def*** .

В третьих: при перегрузке можно метод делать «ленивым» ***(lazy)***. Такой метод не вычисляет результат до тех пор, пока этот результат не будет как-то использоваться, например, выводиться на печать. Мы сделали задержку с помощью ***Thread.sleep(10000)*** и можем убедиться, что переменная ***r*** была вычислена на 10 секунд позже, то-есть только тогда, когда потребовалось выводить её на печать. Время в этих операторах измеряется в миллисекундах.

Попутно мы показали, что вызов метода родительского класса (суперкласса) можно делать с помощью ключевого слова ***super*** (***super.thing*** в классе ***Three***).

Scala позволяет использовать в идентификаторах символы юникода. Например, если среда программирования позволяет ввести знак √, то его можно применить для вычисления квадратного корня:

```
def √(x:Double) = Math.sqrt(x)
print(√(9.0)) → 3.0
```

В Scala любое выражение или блок возвращают последнее вычисленное значение. Например:

val x = io.StdIn.readInt → если здесь ввести ***4,***
val y = {val a = 0.3; val b = x/2.0; a * b}
print(y) → то будет выведено ***0.6*** – результат умножения ***a*b***.
Обратите внимание на операцию деления ***x/2.0***. Здесь смешанные типы: ***x*** типа ***Int*** делится на ***2.0*** типа ***Double***. В Scala это допустимо и результат будет всегда иметь тип ***Double.*** Если делимое и делитель имеют тип ***Int***, то и результат будет иметь тип ***Int*** – целая часть частного:
7/2 – результат равен ***3***.
Легко допустить ошибку, надо быть внимательным.

Методы (функции) тоже всегда возвращают последнее вычисленное значение в теле метода. Однако, можно опускать знак равенства при объявлении метода. Тогда он будет возвращать пустое значение типа ***Unit***:

```
var x:Double = 1.5
def f {x = math.sin(x)}
val y = f → y: Unit = ()
print(x) → 0.9974949866040544
```

Метод ***f***, не имеющий знака равенства, возвращает ***Unit*** и его можно использовать только для изменения переменной ***x***, объявленной со словом ***var***.

2.8 Аргументы метода «по умолчанию» *(default)* и именованные аргументы

```
def f(x:Int, y:Double = 23.06) =  {
      println(x, y)
}
 f(2, 88.0)              → 2, 88.0
f(33)                    → 33, 23.06
f(y = 878.33, x=33)      → 33, 878.33
```

Здесь мы в списке аргументов функции ***f*** переменную ***y*** инициировали числом с плавающей запятой ***23.06***. Теперь при вызове функции переменной ***y*** можно передать новое значение ***(88.0)***, а можно ничего не передавать (умолчать), тогда этот параметр будет равен тому, что указано при определении функции. Надо только иметь ввиду, что аргументы по умолчанию всегда должны стоять в конце списка, после не инициированных аргументов. В третьем примере вызова метода ***f*** мы указали имена

параметров, введённые при определении метода, то-есть сделали аргументы «именованными». Теперь можно не соблюдать порядок расположения аргументов, что очень удобно, если их в списке много.

Аргументы по умолчанию и именованные аргументы могут быть и у классов:

```
class A(x:Int, y:Int = 7) {
        def f = print(y + x)
}
val p = new A(2)
p.f → 9
val p = new A(2, 4)
p.f → 6
val p = new A(y = 3, x = 8)
p.f → 11
```

2.9 Операторы

Все операторы в Scala на самом деле являются методами и их можно вызывать, как обычные методы. Арифметические операторы ***(+), (-), (*), (/), (%)*** записанные в виде ***x + y, x * y*** и так далее, называются инфиксными — знак оператора между величинами. В виде методов их можно вызывать так: ***x.+(y), x.*(y)*** и так далее, а обычная их форма — просто «синтаксический сахар». Всё справедливо и для операторов сравнения ***(==), (!=), (>), (<), (>=), (<=)***. Вместо ***x == y*** можно писать ***x.==(y)*** (или ***y.==(x)***, если нравится). Разумеется всё аналогично и для логических операторов ***(|| - и), (&& - или), (! - не)***. Только оператор «***не***» унарный и вызов, как метода для него будет иметь вид ***!(x)***.

Некоторая особенность есть для оператора ***(::)***, добавляющего элемент к коллекции. Вместо ***val a = 2 :: Nil*** надо писать:

val a = Nil.::(2) → a:List[Int](2)

val b = a.::(3) → b:List[Int](3, 2) и так далее

Оператор ***(::)*** добавляет элементы в начало списка.

Все операторы имеют и «унарную» форму:

var x = 3

x += 5 → на самом деле означает ***x = x + 5***

Ясно, что в последнем случае переменная ***x*** должна быть объявлена со словом ***var*** – изменяемая.

Можно это делать и для ***(::)***

var a = List(1,2,3)

a ::= 4 → a = List[Int](4,1,2,3)

Для операторов сравнения унарная форма не имеет смысла.

В классе ***BigInt*** есть оператор ***(/%)***, возвращающий частное от деления и остаток:

val x:BigInt = 7

val y:BigInt = 3

x /% y → (2,1) здесь результат — кортеж, смотрите далее.

2.10 Блоки (Code Block)

Метод или переменная могут быть определены на одной строке:

```
def f() = "Hello, World!"
```

Или они могут быть определены с помощью блока, заключённого в фигурные скобки и не обязательно на одной строке:

```
def f():String = { "Hello, World!" }
def fi():String = {
        val d = new java.util.Date()
        d.toString()
}
```

Блок, как и любая функция, всегда возвращает последнее вычисленное значение.

Переменным можно присваивать значение блока, как методам:

```
val a:Double = {
        val x = 1.5
        math.sin(x)  → 0.997
}
```

Для инициализации переменной ***a*** значением ***0.997*** использован блок.

2.11 Объявление переменной (Variable Declaration)

Мы уже видели, что переменные объявляются с ключевыми словами ***val, var*** и ***lazy val***. **v*ar*** – переменные могут менять своё значение после первой инициализации, а ***val*** – поля можно инициализировать только один раз. ***lazy val*** переменные тоже получают значение только один раз и только тогда, когда к этим значениям будет осуществлён доступ. Использование ленивых переменных имеет смысл, когда их вычисление требует

значительного времени, а результат вычисления может и не потребоваться. В такой ситуации ленивые вычисления могут дать экономию времени.

Предпочтительнее использовать ***val*** – переменные всегда, если только нет прямой необходимости применить вариант ***var***. Это даёт дополнительную гарантию от случайных ошибок, связанных с непредусмотренным изменением полей.

Scala поддерживает множественную инициализацию:
val (x:Int, y:String) = (33, "Москва") → x = 33, y = "Москва"
или, используя автоматический вывод типов:
val (x, y, z) = (33, "Москва", 24.78)
При множественной инициализации справа от знака равенства используется так называемый кортеж ***(Tuple)***, а также вместо него может быть любое выражение, возвращающее кортеж.

2.12 Val – def

val служит для объявления переменных (неизменяемых -*immutable*), а ***def*** – для объявления функций (методов). В обоих случаях синтаксис аналогичный: ключевое слово (***val*** или ***def***), идентификатор, знак равенства ***(=)***, выражение. И это не случайно, так как переменные тоже можно трактовать, как выражения и объявлять со словом ***def***.
def x = 99 → x: Int
val y = 99 → y: Int = 99
В принципе ***x*** и ***y*** можно дальше использовать на одинаковых правах:
x+y → res: Int = 198
Одинаково указывается и тип:
def x :String = "Hello" → x: String
Но между этими вариантами есть и разница — она отражена в результатах, выведенных интерпретатором (в случае ***def*** значение переменной ***x*** не выведено на экран). Если объявление имеет вид: ***def x = e***, где ***e*** – выражение, то это выражение не вычисляется сразу в момент объявления, а вычисляется только там, где переменная будет использована (потому в нашем примере значение ***x*** не было выведено на экран; оно ещё и не вычислялось). При объявлении:
val y = e выражение ***e*** вычисляется сразу, а потом уже

подставляется на место ***y*** его готовое значение.
Фактически объявление со словом ***def*** почти то же самое, что и ***lazy val***.

С другой стороны, с помощью ***val*** можно объявлять и методы, используя анонимные функции:
val f = (x:Int, y:Int) => x * y → f: (Int, Int) => Int = <function2>
(Что такое ***function2***, мы рассмотрим дальше). Как и во всех других случаях, указывать типы аргументов необходимо.
f(5,9) → res: Int = 45
Вообще говоря, эта конструкция с полным объявлением всех типов выглядит так:
val f:((Int, Int) => Int) = (x:Int, y:Int) => x * y
Но теперь, если указан тип функции - ***(Int, Int) => Int*** (функция принимает два аргумента типа ***Int*** и возвращает результат типа ***Int***), то типы аргументов справа можно опустить:
val f:((Int, Int) => Int) = (x, y) => x * y

Обычно выражения типа ***(Int, Int) => Int)*** называют сигнатурой функций.

Объявление методов с ***val*** широко применяется при функциональном стиле на Scala.

Глава 3 Коллекции

В Scala имеется большой набор коллекций и огромное количество различных способов работы с ними. Коллекции играют такую большую роль, что, пожалуй, целесообразно рассмотреть их в отдельной главе. Можно получить представление о разнообразии коллекций из одного только перечня их названий: список (***List***), вектор (***Vector***), ранг (***Range***), массив (***Array***), ассоциированный массив (***Hash***), множество (***Set***), пара (***Pair***), кортеж (***Tuple***). Эти виды коллекций встречаются и в других языках, но в Scala есть ещё: ***Seq, Set, Option***. Тип ***String*** тоже обрабатывается, как коллекция. Почти все эти виды коллекций неизменяемы (*immutable*), но, кроме того, многие из них имеют ещё и изменяемые (*mutable*) разновидности, расположенные в модуле *mutable*. Так, например, *mutable* -списки с названием ***ListBuffer*** вызываются так:

val a = collection.mutable.ListBuffer(1,2,3)
Это их полный адрес, но можно воспользоваться и командой import:
import collection.mutable._
val a = ListBuffer(1,2,3)

Кроме того, есть, разумеется, стандартные структуры данных, которые тоже относятся к коллекциям: поток (***Stream***), стек (***Stack***), очередь (***Queue***), связанный список (***LinkedList***) из того же модуля ***mutable***. С ума можно сойти! Обсудим пока только основные аспекты.

3.1 Списки (*List*)

Для объявления списка применяется следующий синтаксис:
val a:List[Int] = List(2, 35, 9, 18)
Список ***a*** будет иметь элементы типа ***Int***. То же самое можно объявить и так:
val a = List[Int](2, 35, 9, 18)
Если тип не объявлять вовсе, то он будет выведен компилятором:
val a = List(2, 35, 9, 18) → опять будет ***List[Int](2, 35, 9, 18)***
Попробуем ввести в состав списка элементы разных типов:
val a = List(2, "Dog", 34.21, false) → получим ***List[Any](2, Dog, 34.21, false)***
Как и можно было ожидать, элементы будут иметь тип ***Any***.
Можно, конечно, и сразу объявить этот тип:
val a:List[Any] = List(2, "Dog", 34.21, false)
Для извлечения элементов из списка можно использовать индекс, как это обычно бывает у массивов:
a(1) → ***Dog*** → значит, нумерация начинается с нуля, а скобки круглые.
Ранее мы уже упоминали, что к переменным типа ***Any*** можно применять метод toString: ***val x = a(1).toString → x: String = Dog***
А чтобы получить элементы других типов придётся делать так:
val x = a(0).toString.toInt → x: Int = 2
val y = a(3).toString.toBoolean → y: Boolean = false и так далее.
Уже говорилось, что есть специальный метод ***asInstanceOf***:
val x = a(0).asInstanceOf[Int] → x:Int = 2
Точно также надо поступать и для трансформации в другие типы.

Есть ещё тип ***Number***, который применяется, если в списке

элементы являются числами, но разных типов: ***Int, Double*** и т. д.:
val a = List[Number](1, 44.5, 8d) → a: List[Number] = List(1, 44.5, 8.0)
Чтобы трансформировать тип ***Number*** в число, надо поступать также, как и с типом ***Any***:
val x = a(0).toString.toInt → x: Int = 1
val x = a(1).toString.toDouble → x: Double = 44.5

Списки, объявленные с ***val*** не изменяемые, операции со списками создают новые экземпляры списков. Так оператор **(::)** (называется ***cons***) создаёт новый список, добавляя заданный элемент в начало списка:
val a = List(2,4)
val b = 3 :: a → List(3,2,4)
При этом добавляемый элемент называют головой ***(head)*** списка ***b***, а список ***a*** – хвостом ***(tail)*** списка ***b***. Есть одноименные операторы для извлечения головы и хвоста из списка:
b.head → 3
b.tail → List(2,4)
Кстати, метод ***last*** позволяет извлечь последний элемент списка:
b.last → 4
А метод ***init*** возвращает список без последнего элемента
val a = List(1,2,3,4,5)
val b = a.init → b:List[Int] = List(1, 2, 3, 4)
Ещё несколько методов на примерах:
a.take(2) → List(1, 2)
a.drop(2) → List(3, 4, 5)
a.splitAt(2) → (List(1, 2),List(3, 4, 5))
В последнем случае результат — кортеж, элементами которого являются списки (о кортежах дальше).
a.slice(1,4) →List(2, 3, 4)
a.reverse → List(5, 4, 3, 2, 1)
Ну, и так далее...

Оператор ***(::)*** позволяет создавать списки:
val a = 1 :: 2 :: 3 :: Nil
Nil представляет пустой список, к которому последовательно добавляются элементы в начало. Вместо ***Nil*** можно писать ***List()*** - список без элементов. Этот способ представления списков очень полезен, как мы увидим далее.

Оператор ***(:+)*** позволяет добавлять элемент в конец списка, а оператор ***(+:)*** - в начало:
val a = List(1, 2, 3)
val b = a :+ 4 → b:List[Int] = List(1, 2, 3, 4)
val c = 0 +: a → c:List[Int] = List(0, 1, 2, 3)
То-есть, то же самое, что и
val c = 0 :: a → c:List[Int] = List(0, 1, 2, 3)

Если список объявить ***var***, в него можно добавлять элементы «на месте», об этом уже говорилось:
var a = List(1,2,3)
a = a :+ 4 → a:List[Int] = List(1, 2, 3, 4)
Оператор ***(++)*** позволяет объединять два списка в один:
val a = List(1,2,3)
val b = List(11,22,33)
val c = a ++ b → List[Int](1,2,3,11,22,33)
Если список ***a*** объявить с ***var***, к нему также можно добавлять другие списки, то-есть писать ***a = a ++*** *b* и даже ***a ++= b***.

Для слияния двух списков в один имеется ещё один метод, обозначаемый тройным двоеточием ***(:::)***:
val x = List(1,2,3)
val y = List(99, 98, 97)
val z = x ::: y → z: List[Int] = List(1, 2, 3, 99, 98, 97)
Соединяемые списки могут содержать элементы разных типов, тогда результат получает тип ***Any***.

Нет никакой возможности изменять элементы списка, то-есть писать, например, ***a(3) = 999*** – списки *immutable*. Есть, правда, возможность использовать изменяемые списки под названием ***ListBuffer***, или связанные списки ***LinkedList,*** но о них пока отложим разговор. А вообще, если надо изменять элементы коллекции, лучше использовать массивы, но о них дальше.

Элементами списков (как, собственно, и всех других коллекций) может быть всё что угодно, например, экземпляры классов или другие коллекции. В частности, элементами могут быть другие списки:
val a = List(List(1,2,3), List(4,5,6), List(7,8,9))
Таким образом создаются двумерные списки, элементы из них извлекаются по двум индексам:
a(1)(2) → 6

3.2 Ранги *(Range)*

Иначе ранги называют диапазонами. Объявим ранг ***r:***

val r = 0 to 5 -> r: = Range(0, 1, 2, 3, 4, 5)

На самом деле ***to*** метод:

val r = 0.to(5) → будет тоже самое

Вместо ***to*** можно использовать ***until*** и тогда диапазон не будет включать последнее значение:

val r = 0 until 5 → r: = Range(0, 1, 2, 3, 4)

Можно объявлять ранг и так:

val a = Range(0, 5) → Range(0, 1, 2, 3, 4)

Последнее значение ***5*** не вошло в диапазон.

Ранги обладают почти теми же качествами, что и списки с элементами типа ***Int***. Так, элемент ранга можно извлечь по индексу:

val x – r(3) → x:Int = 3 *(*тип всегда ***Int****)*

Можно использовать ранг с шагом больше единицы:

val r = 0 to 10 by 3 → Range(0, 3, 6, 9)

или так:

val a = Range(0, 9, 2) → Range(0, 2, 4, 6, 8)

Коллекции могут быть точными (строгими ***strict***) или ленивыми ***(lazy)***. ***Lazy*** коллекции имеют элементы, которые не записываются в память, пока к ним не будет осуществлён доступ. Таким качеством обладают, например, ранги:

val r = (1 to Integer.MAX_VALUE - 1).take(5)

Для этой коллекции в полном объёме потребовалось бы несколько гигабайт, но поскольку осуществлена выборка только пяти (***take*** – ключевое слово), то только они и будут занимать память. Списки, например, не обладают этим свойством. Позже мы ещё немного поговорим о ленивых вычислениях.

3.3 Векторы (*Vector*)

Вектор объявляется подобно списку:

val b:Vector[Int] = Vector(2, 35, 9, 18)

Вектор это индексируемая последовательность с быстрым произвольным доступом к элементам. Векторы реализованы, как деревья, что позволяет многократно ускорить поиск элемента по индексу. В основном свойства векторов совпадают со свойствами

списков, но не все. Например, для векторов не работает оператор ***(::)***
33 :: b → ошибка,
но
b.head → 2 **-** всё нормально

3.4 Множества (*Set*)

Множества это не индексируемая коллекция, в которой каждый элемент может присутствовать только в одном экземпляре. Элементы множества могут иметь любые типы, допустимы смешанные множества:
val a = Set("aa", 34.7, 67) → a:Set[Any] = Set(aa, 34.7, 67)
Ясно, что тип множества будет ***Any***.
Для добавления новых элементов в множество применяется оператор ***(+)***:
val a = Set(1, 6, 2, 8, 4)
val b = a + 99 → Set(1, 6, 2, 99, 8, 4)
Если добавить существующий элемент, множество не изменится. В множествах не сохраняется порядок добавленных элементов. (Вообще говоря, расположение элементов упорядочено по результатам, возвращаемым методом ***hashCode***. В Scala все объекты имеют этот метод. Можете попробовать, но пока не будем вдаваться в детали). Благодаря этому поиск в множествах выполняется быстрее, чем в других коллекциях. Метод ***contains*** проверяет, входит ли в множество указанное значение:
a.contains(6) → true
a.contains(22) → false
Слово ***contains*** вообще-то можно опускать:
a(4) → true (действует только для множества)
Метод ***subsetOf*** проверяет, присутствуют ли все элементы множества в другом множестве
val a = Set(1,6,3,11,8)
val b = Set(6,11,8)
b subsetOf(a) → true
val c = Set(13, 11,8)
c subsetOf(a) → false
Методы ***union (|), intersect (&)*** и ***diff (&~)*** (можно применять слова или знаки) покажу на примерах:

a | c → Set(1, 6, 13, 3, 11, 8)
a & c → Set(11, 8)
a &~ c → Set(1, 6, 3)
Операцию объединения можно также записывать ***(++)***, а операцию разности множеств - ***(--)***.

3.5 Массивы (***Array***)

Объявляется массив также, как и список:
val a:Array[Int] = Array(1, 2, 3), или так:
val a = Array[Int](1, 2, 3)
Явно тип тоже можно не объявлять:
val a = Array(1, 2, 3) → будет то же самое
Точно также, как и списки, создаются массивы с элементами разных типов (на самом деле элементы будут иметь тип *Any)*.
val a = Array(2, "Dog", 34.21) → a: Array[Any] = Array(2, Dog, 34.21)
И извлекаются элементы также, как у списков:
val x = a(2) → x:Any = 34.21
val y = x.toStrig.toDouble → y:Double = 34.21
Теперь укажем, в чём же массивы отличаются от списков. Прежде всего элементы массива можно инициировать новыми значениями, даже и для ***val*** массивов:
val a = Array(1, 2, 3) → a: Array[Int] = Array(1, 2, 3)
a(2) = 767 → a: Array[Int] = Array(1, 2, 767)
С элементами списков так делать нельзя.

Массивы являются представителями класса ***Array***, поэтому их можно объявлять и таким способом :
val a = new Array[Int](3) → a: Array[Int] = Array(0, 0, 0)
(***List*** тоже класс, но абстрактный, а потому к нему не применим метод ***new***). Аргумент ***(3)*** указывает размер создаваемого массива, а все элементы инициализированы нулями.
val a = new Array[String](4) → a: Array[String] = Array(null, null, null, null)
В этом случае все элементы имеют специальное значение ***null***.
a(2) = "Dog" → a:Array[String] = Array(null, null, Dog, null)
Значит, элементы так созданного массива тоже можно изменять.
И наконец:
val a = new Array[Any](4) → a: Array[Any] = Array(null, null, null,

null)

Теперь можно задавать элементам значения разных типов.

Есть возможность работать и с многомерными массивами, но их объявление весьма громоздко:

val a:Array[Array[Int]] = Array(Array(1,2,3), Array(4,5,6), Array(7,8,9)) → a: Array[Array[Int]] = Array(Array(1, 2, 3), Array(4, 5, 6), Array(7, 8, 9))

На самом деле это массив с элементами — массивами.

Извлекаются элементы с помощью двойного индекса:

val x = a(2)(1) → x:Int = 8

Можно, конечно воспользоваться автоматическим выводом типов:

val a = Array(Array(1,2,3), Array(4,5,6), Array(7,8,9)) → то же самое.

Scala имеет также специальный метод ***ofDim*** для создания многомерных массивов. Покажем на примере:

val m = Array.ofDim[Double](3, 4) → m: Array[Array[Double]] = Array(Array(0.0, 0.0, 0.0, 0.0), Array(0.0, 0.0, 0.0, 0.0), Array(0.0, 0.0, 0.0, 0.0))

Получили двумерный массив размером 3 x 4 с элементами типа ***Double***, равными нулю. Инициируем какой-нибудь элемент другим значением:

m(1)(2) = 34.78 → можно убедиться, что элемент изменил значение.

Точно также создаются многомерные массивы любой размерности:

val m = Array.ofDim[Int](3, 4, 3) → тоже все элементы равны нулю.

Метод ***toArray*** позволяет другие коллекции трансформировать в массив.

val a = Set("aa", "mm", "kk")

a.toArray → Array(aa, mm, kk)

Имеется множество методов, позволяющих добавлять элементы в массив и, наоборот, удалять их из массива. Фактически, методы, примеры которых мы рассмотрели для списков, применяются для коллекций всех видов, в том числе и для массивов. Надо руководствоваться таким правилом: метод применим к коллекции любого вида, если только он подходит по смыслу. Например, метод ***sorted*** сортирует списки и массивы, но он не может сортировать множества, в которых элементы не упорядочены.

val b = Array(3,1,7,4,9,1)

b.sorted → Array(1, 1, 3, 4, 7, 9)
val m = Set(3,1,7,4,9,1)
m.sorted → ошибка

3.6 Кортежи (*Tuple*)

Ещё одна разновидность коллекций, применяемых в Scala – так называемые ***Tuple*** (в других языках их обычно называют «кортеж»). Принят такой способ объявления кортежа:
val x:Tuple2[Int, Int] = (1,2) → x: (Int, Int) = (1,2)
Используется ключевое слово ***Tuple*** с цифрой на конце, указывающей, сколько элементов в кортеже (кроме единицы, объявлять кортеж с одним элементом не имеет смысла). Затем, как обычно, перечисляются типы элементов в квадратных скобках. Справа от знака равенства в круглых скобках, через запятую перечисляются значения всех элементов. Типы элементов могут быть разными:
val t:Tuple3[Int, String, Double] = (12, "Peter", 0.045) →
t: (Int, String, Double) = (12,Peter,0.045)
Как всегда, можно использовать автоматический вывод типов, можно опускать и само слово ***Tuple***:
val t = (12, "Peter", 0.045) → t: (Int, String, Double) = (12,Peter,0.045)
Можно объявлять кортеж и так:
val x = Tuple4(false, 0.75, "privet", 333) → x: (Boolean, Double, String, Int) = (false,0.75,privet,333)
Особую роль в Scala имеют кортежи с двумя элементами, обычно их называют ***Pair*** или «пара».
val x = Tuple2(1,2) → x: (Int, Int) = (1,2)
Кроме того, можно воспользоваться знаком ***(->)***:
val x = 1 -> 2 → будет то же самое
А если объявить кортеж так:
val x = 1 -> 2 -> 3 →x: ((Int, Int), Int) = ((1,2),3) ,
то получим пару из двух элементов, а первый из них сам будет парой.

Для извлечения элементов из кортежа принят несколько необычный синтаксис:
val t = (12, "Peter", 0.045)
t._1 → 12, t._2 → Peter, t._3 → 0.045

Следовательно, индексация элементов в кортеже начинается с единицы. Можно извлекать элементы из кортежа с помощью группового присваивания:

val t = (2, “aa”, 3.6)

val (x, y, _) = t → x = 2, y = “aa”

Поскольку третий элемент не использован, мы заменили его знаком (_).

Рассмотрим такой пример:

```
def f(in: List[Double]): (Int, Double, Double) =
in.foldLeft((0, 0d, 0d))((t, v) => (t._1 + 1, t._2 + v, t._3 + v * v))
```

Функция ***f*** принимает один аргумент — список ***in*** с элементами типа ***Double*** и возвращает кортеж из трёх элементов с типами ***Int, Double, Double***. В правой части к списку ***in*** применяется метод ***foldLeft***, у которого начальный параметр (аккумулятор) — кортеж с нулевыми значениями элементов, имеющих типы, как у возвращаемого результата (напоминаю, что ***(0d)*** означает число нуль типа ***Double***). В блоке в каждом цикле итератора к первому элементу кортежа каждый раз добавляется единица, ко второму — элементы списка ***in*** и к третьему — эти элементы, возведённые в квадрат. Пример вызова этой функции для конкретного списка:

val res = f(List(1.0, 2.0, 3.0, 4.0)) → res:(Int, Double, Double) = (4, 10.0, 30.0)

Кортежи удобны для случаев, когда функция должна возвращать несколько значений:

```
def f(x:Double) = {
        val a = Math.sin(x)
        val b = Math.cos(x)
        val c = Math.tan(x)
        (a,b,c)
}
```

f(0.7) → (0.64421768723, 0.764842187284, 0.842288380463)

Мы воспользовались тем, что блок возвращает последнее вычисленное значение, хотя Scala допускает и применение служебного слова ***return:***

return(a,b,c) → вернёт кортеж

Есть полезный метод ***partition***, возвращающий пару строк, содержащих символы из заданного текста удовлетворяющие и не удовлетворяющие заданному условию:

val x = "21 January 1924 Yar"
x.partition(_.isDigit) → (211924," January Yar")
Метод ***isDigit*** возвращает ***true***, если очередной символ — цифра, поэтому в первом элементе пары отобраны все цифры, а во втором — все буквы и пробелы.

Иногда требуется объединять значения в пары для последующей совместной их обработки. Для этой цели есть метод ***zip***, объединяющий элементы двух коллекций в пары ***(Pair)***:
val a = List("<","_",">")
val b = List(2,10,2)
val p = a.zip(b) → List((<,2), (_,10), (>,2))
Список ***p*** имеет элементы в виде пар и эти пары можно обабатывать совместно:
for ((s,n) <- p) print(s*n) → <<_____>>
Обратите внимание: произведение переменных ***s:String*** и ***n:Int*** даёт в результате строковый литерал в виде символов, повторяющихся соответствующее число раз.

3.7 Хеш, Мап (*Hash, Map*)

Map называют также: ***Hash*** (хеш), ассоциированный массив, или даже словарь. Этот вид коллекций обладает поистине огромными возможностями и позволяет эффективно решать многие задачи. Объявляется ***Map*** так:
val m = Map("one" -> 1, "two" -> 2, "three" -> 3)
Его элементы представлены парами (кортежами), в которых первый элемент называется «ключ», а второй - «значение». Ключ играет роль индекса, по которому извлекается значение:
val x = m("two") → x:Int = 2
Тип результата выводится компилятором. Можно задать типы ключей и значений явно:
val m1:Map[String, String] = Map("Lusja" -> "Elsucova", "Petja" -> "Ivanov", "Kolja" -> "Sidorov")
На эти типы не накладывается никаких ограничений.
Попробуем смешанные типы:
val m = Map("Masha" -> 18, 25 -> "Kolja", 21.3 -> "Petja")
Естественно, тип будет ***Map[Any, Any]***. Ну, и как обычно:
val x = m("Masha") → x: Any = 18
val y = x.toString.toInt → y:Int = 18

Map тоже *immutable*, но есть библиотечный метод создания изменяемых ***Map****:*
val m = collection.mutable.Map[String, Int]("a" -> 1, "b" -> 2, "c" ->3)
Теперь есть возможность изменять элементы:
m("b") = 78 → m = Map(b -> 78, a -> 1, c -> 3)
Обратите внимание, что порядок элементов изменился, в коллекциях ***Map*** не ведётся контроль за порядком расположения элементов. Можно добавлять новые элементы:
m("Bob") = 25 → Map(Bob -> 25, b -> 78, a -> 1, c -> 3)
Или так:
m += ("x" -> 55, "y" -> 66) → Map(Bob -> 25, b -> 78, y -> 66, a -> 1, x -> 55, c -> 3)
А так можно удалять элементы:
m -= ("a", "b", "c") → Map(Bob -> 25, y -> 66, x -> 55)
Ещё пара примеров:
for ((k,v) <- m)println(v) → 25 66 55 (Вывести на печать все значения*)*
for ((k,v) <- m)println(k) →Bob y x (все ключи*)*
Есть ещё вариант:
for (v <- m.values)println(v) → 25 66 55
values – служебное слово.
Следующий приём позволяет поменять местами ключи и значения:
for((k, v) <- m)yield(v, k) → Map(55 -> x, 25 -> Bob, 66 -> y)
yield – также служебное слово, мы ещё встретимся с ним дальше.
Пусть у нас есть два массива:
val k = Array("a", "b", "c")
val v = Array(2, 9, 5)
Уже говорилось, что метод ***zip*** позволяет из этих массивов сформировать массив из пар (кортежей):
val a = k.zip(v) → a: Array[(String, Int)] = Array((a,2), (b,9), (c,5)),
а метод ***toMap*** превратить этот массив в ***Map***:
val m = a.toMap → m: Map[String,Int] = Map(a -> 2, b -> 9, c -> 5)
Методы ***zip*** и ***toMap*** с таким же успехом работают и со списками.

Ключи в отдельном ***Map*** должны быть все уникальными (не должны повторяться); «значения» могут быть любыми. Создадим *Map*:
val p = Map(1 -> "David", 9 -> "Elwood")

Новые элементы добавляются с помощью знака сложения (+):
val q = p + (8 -> "Archer") → Map(1 -> David, 9 -> Elwood, 8 -> Archer)
При этом сам исходный Map ***p*** не изменяется. Если есть необходимость его изменить, надо объявлять с ***var***:
var p = Map(1 -> "David", 9 -> "Elwood")
Тогда можно изменять так:
p = p + (8 -> "Archer") → p:Map[Int,String] = Map(1 -> David, 9 -> Elwood, 8 -> Archer)
Допустимо и так:
p += (8 -> "Archer") — скобки здесь можно и убрать.

Знак вычитания (-) позволяет удалять элементы:
p -= 9 → p:Map[Int,String] = Map(1 -> David, 8 -> Archer)
Мы удалили элемент с ключом, равным ***9***. Есть возможность проверить, есть ли элемент с заданным ключом, с помощью метода *contains*:
val u = p.contains(1) → u: Boolean = true
val u = p.contains(11) → u: Boolean = false

Метод ***keys*** позволяет получить все ключи заданного в виде коллекции типа *Set* (множество):
val k = p.keys → k: Iterable[Int] = Set(1, 8)
(Переменная ***k*** получила тип ***Iterable[Int]***, но пока не будем разбираться с ним). Результат можно использовать разными способами, например:
k.reduceLeft(_ max _) → 8
Здесь использован метод ***reduceLeft***, подобных методов в Scala очень много, кое-что рассмотрим позже.

Аналогично можно получить коллекцию значений:
val v = p.values → v: Iterable[String] = MapLike(David, Archer)
Это коллекция ***MapLike***, она не то же, что ***List*** (количество видов коллекций в Scala порой кажется просто бесконечным).
v.reduceLeft((a, b) => if (a > b) a else b) → David
Вариант с ***(_ max _)*** при элементах типа ***String*** в этом случае не работает.

С помощью оператора ***(++)*** можно добавлять группу новых элементов в *Map*:
p ++= List(5 -> "Cat", 6 -> "Dog") → Map(1 -> David, 8 -> Archer, 5 -> Cat, 6 -> Dog)

Добавляемые элементы заданы в виде списка пар *(Pair)*. И можно удалять группу элементов с заданными ключами с помощью оператора *(--):*
p --= List(8, 6) → Map(1 -> David, 5 -> Cat)

3.8 Option[T] (ещё один сорт коллекций)

Option можно рассматривать, как контейнер для одного элемента заданного типа ***T***. (По моему, это несколько странная коллекция). В сущности, ***Option[T]*** может иметь только два значения: ***Some[T]*** (то-есть служебное слово ***Some*** и значение типа ***T*** в круглых скобках) или ***None***.

По-видимому, ***Option*** чаще всего используется для охраны (*guard*) при поиске каких-то данных по ключу. Например, это может быть извлечение значений из *Map* (также и из любой базы данных). Если в программе случится вызов элемента с несуществующим ключом, произойдёт исключение (сообщение об ошибке с остановом). Для предотвращения такой нежелательной ситуации при извлечении следует применять метод ***get***. Пусть у нас есть *Map*:
val h = Map(1 ->"aa", 2->"bb", 3→"cc") Тогда:
val x = h(2) → x: String = bb (всё в порядке)
val x = h(9) → ошибка!
Теперь применим ***get****:*
val x = h.get(2) → x: Option[String] = Some(bb)
val y = h.get(9) → y: Option[String] = None
Значит, ***get*** возвращает ***Option*** и не допускает исключения. Чтобы извлечь из ***Some(bb)*** сам элемент, надо ещё раз применить ***get***:
val a = x.get → a: String = bb Можно, конечно, сразу писать двойное ***get****:*
val a = h.get(2).get → a: String = bb
Однако, и это ещё не все проблемы, так как применение ***get*** к ***None*** тоже вызовет исключение. По этой причине вместо второго ***get*** лучше применять метод ***getOrElse***, который в случае ***None*** будет возвращать заданное нами значение по умолчанию вместо исключения.
val p = Map(1 -> "David", 9 -> "Elwood", 8 -> "Archer")
val x = p.getOrElse(99, "Nobody") → x: String = Nobody
Ключа ***99*** нет и метод ***getOrElse*** вернул заданное значение по

умолчанию.

val x = p.getOrElse(8, "Nobody") → x: String = Archer

Ключ ***8*** имеется и мы получили искомое значение. Таким образом, делая в программе проверку на ***None***, можно избежать ошибочных ситуаций.

Справедливости ради отметим, что использование этих ***Some*** и ***None*** выглядит довольно уродливо и что в Ruby все эти проблемы решаются легко и просто. Впрочем, Scala обладает и другими, более простыми возможностями. Так, для проверки всяких ошибочных ситуаций (исключений) есть эффективный оператор ***try-catch***. Используя тот же Map ***p***, применим для извлечения элемента этот ***try-catch***. Создадим функцию:

def f(n:Int) = try {p(n)} catch {case _:Throwable => 0}

В блоке для ***try*** мы извлекаем элемент по ключу — аргументу функции ***f***, и если всё нормально, полученное значение возвращается, как результат, а ветвь ***catch*** не выполняется. В противном случае оператор ***case*** вернёт нам нуль. (Тип ***:Throwable*** требуется по внутренним причинам, без него будет предупреждение, хотя тоже всё пройдёт нормально). Выполнив этот анализ (в общем-то тут вместо нуля можно вставить что угодно), мы обойдём аварийную ситуацию. Итак:

f(2) → :Any = 0

f(9) → :Any = Elwood

По умолчанию мы получили тип ***Any***, но мы уже знаем, как его трансформировать в другие типы. Впрочем, ничто нам не мешает получить результат другого типа, например:

def f(n:Int):String = try {p(n)} catch {case _:Throwable => 0.toString}

Теперь мы получим тип *String*.

Коллекция ***Option*** применяется также во многих модулях из библиотеки Java, которая доступна на Scala без ограничений. Не будем пока останавливаться на этом подробнее.

3.9 Коллекция *Seq*

В документации Scala можно прочитать, что ***Seq*** это упорядоченная коллекция значений. Поскольку тоже самое можно сказать про списки и массивы, то получается, что ***Seq*** их общий представитель. Действительно, если объявить

val a = Seq(1,2,3,4)
то получим такой результат:
a: Seq[Int] = List(1, 2, 3, 4)
То-есть здесь ***Seq*** и ***List*** одно и то же.
Но допустимо и так:
val c:Seq[Int] = Array(1,2,3,4)
и тогда получим:
c: Seq[Int] = WrappedArray(1, 2, 3, 4)
Значит, теперь ***Seq*** есть ***WrappedArray***, некий «упакованный массив».

Коллекция ***Seq*** позволяет использовать ряд дополнительных методов; кое-что рассмотрим дальше.

Глава 4 Управляющие структуры

В примерах выше уже использовались условия и циклы, рассмотрим теперь их более подробно.

4.1 If - else (условные выражения)

Мы уже неоднократно использовали оператор ***if*** и ***if – else***. В сущности он мало отличается от подобных операторов в других языках, в частности, в Ruby. Укажем только некоторые особенности:

Оператор ***if*** в Scala всегда возвращает значение типа ***Unit***:
val e = 5 > 3 → e:Boolean = true
val x = if (e) 55 → x:Unit = 55 (Если будет ***e = false***, то получим ***x:Unit = ()***)

После условного выражения в скобках может быть любое выражение:
if (a == b) print(“yes”) → если ***a*** равно b будет напечатано ***yes***
и даже может быть блок:

```
val a = if (e)  {
        val x = 5.0
        x/2
}
```

print(a) → 2.5 (если ***e = true***)
Переменная ***a*** имеет тип ***Unit***. Кстати, к типу ***Unit*** почему-то нельзя

применить метод ***toDouble*** но метод ***toString*** – можно. Поэтому, чтобы ***a*** трансформировать к типу ***Double***, придётся поступить так:
a.toString.toDouble → a:Double = 2.5

Вариант ***if – else*** отличается тем, что возвращает результат типа ***Any***, если ветви ***if*** и ***else*** возвращают разные типы:
val a = if (e) 2 else "Hello" → a:Any = Hello (если e = false)
a.toString → a:String = Hello
Если же обе ветви дают результат одного типа, то и всё выражение вернёт результат этого типа:
val a = if (e) "Bob" else "Hello" → a:String = Hello (если ***e = false)***

На месте обоих выражений могут быть блоки:

```
val e = true
val a:Double = if (e) {val x = 3; val y = x*x; y} else
      {val x = 2.31; math.cos(x)}
print(a) → 9.0
```

То-есть, можно в таких случаях указывать тип результата явно.

Если ветвь ***else*** отсутствует, то может оказаться, что выражение не вернёт никакого результата. В Scala предусмотрено, что в этом случае возвращается результат типа ***Unit***, равнозначный ***()***.

```
val x = 2
val y = if (x>3) "aaa" → y:Unit = ()
```

(На самом деле тут может оказаться тип ***Any*** или ***AnyVal***, но в общем-то какая разница для пустого места)

4.2 Циклы (*while* и *for*)

Оператор ***while*** применяется для циклов:
while (true) println("Hello") → будет бесконечно печатать ***Hello***
Пример более содержательного цикла:

```
var i = 0
while (i <= 5)  {
      print(i + " ") → 0 1 2 3 4 5
      i+= 1
}
```

Оператор ***while*** может стоять в конце цикла; всё это имеется в любом другом языке программирования. В Scala предпочитают для организации подобных циклов использовать рекурсию, но о ней позже.

Оператор цикла ***for*** мы уже применяли для простейших случаев.

В Scala этот оператор имеет очень существенные дополнительные возможности.

Так, оператор ***for*** поддерживает защиту (*guard*). Для этого надо после параметров цикла вставить условный оператор ***if***:
def isOdd(in: Int) = in % 2 == 1 → (true или false)
for {i <- 1 to 5 if isOdd(i)} println(i) (Обратите внимание на использование обратной стрелки ***(<-)*** и ранга ***1 to 5***).
Здесь оператор ***println(i)*** будет выполнен только для тех ***i***, для которых созданный нами метод ***isOdd*** даст значение ***true.***
Следовательно будут выведены на печать только нечётные числа (***in%2*** равно ***1*** только для нечётных ***in***, операция ***%*** возвращает остаток от деления).

Guard может быть и во вложенном цикле:
for {i <- 1 to 5
j <- 1 to 5 if ((i * j)%2 == 0)} println(i * j)
Теперь будут выведены только чётные числа.

Переменную цикла можно изменять с заданным шагом:
for (i <- 0 to (7,2))print(i + " ") → 0 2 4 6
С помощью метода ***reverse*** можно выполнять обход в обратном порядке:
for (i <- (0 to (7,2)).reverse)print(i + " ") → 6 4 2 0

Цикл ***for*** можно эффективно использовать для трансформации коллекции или её части в новую коллекцию. Определим такой список:
val a = (1 to 18 by 3) .toList
Выражение ***(1 to 18 by 3)*** определяет ранг — последовательность целых чисел от ***1*** до ***18*** через ***3***, а метод ***toList*** превращает ранг в список: ***List(1, 4, 7, 10, 13, 16)***.
Если в теле цикла ***for*** использовать ключевое слово ***yield***, то можно легко создавать новые коллекции:
for {i <- a if isOdd(i)} yield i → List(1, 7, 13)
Также и во вложенном цикле:
for {i <- a; j <- a if isOdd(i * j)} yield i * j → List(1, 7, 13, 7, 49, 91, 13, 91, 169)
Такие циклы называют ***for*** — генераторами. Позже мы рассмотрим применение ***for*** для разных трансформаций коллекций более подробно.

При обходе коллекций с помощью ***for*** нет нужды использовать

индексы:
val a = List(1,2,3,4,5)
var p = 1
for (i <- a) p *= i
p → 120
Текст можно обрабатывать, как коллекцию:
var s = 0
for (i <- "Hello") s+=i
s → 500
Подумайте, почему тут результат — целое число.

В заголовке цикла ***for*** можно указывать несколько генераторов в форме (*переменная <- выражение)*, разделяя их знаком ***(;)***
for(i <- 1 to 3;j <- 1 to 3)print((10*i+j) + " ")->11 12 13 21 22 23 31 32 33
Каждый генератор тут может иметь свой *guard*:
for(i <- 1 to 3;j <- 1 to 3 if i != j)print((10*i+j) + " ") → 12 13 21 23 31 32
Допускается любое количество определений, вводящих переменные для использования в цикле:
for(i <- 1 to 3;x = 4 -i;j <- x to 3)print((10*i+j) + " ") →13 22 23 31 32 33
Как видите, циклы ***for*** на Scala могут быть весьма изощрёнными.

4.3 Сопоставление с образцом (*Pattern Matching*)

Конструкция ***case*** – одно из средств ветвления - присутствует в разных вариантах в любом современном языке программирования, но в Scala этому оператору отведена особая роль. Scala совмещает объектно-ориентированное (ООП) и функциональное программирование. В функциональных языках сопоставление с образцом (Pattern Matching) имеет основополагающее значение. В Scala этот механизм как раз и построен на ***case***.

Начнём с простого примера вычисления чисел Фибоначчи:
```
def fib(x: Int): Int = x match {
    case 0 => 0
    case 1 => 1
    case _ => fib(x - 1) + fib(x - 2)
    }
println(fib(6)) → 8
```

print(fib(7)) → 13

Оператор построен на паре служебных слов ***match-case***. После имени исследуемой переменной (объекта), следует слово ***match***, а далее в фигурных скобках располагаются операторы сравнения ***case***. После слова ***case*** находится сопоставляемый образец, а справа от знака ***(=>)*** - результат, который будет возвращён функцией, если сопоставление даст ***true***. При этом следующие сопоставления уже не будут выполнены. Если сопоставление неудачно ***(false)*** – будет переход на следующий ***case***. В нашем примере в третьем ***case*** использована рекурсия — приём, при котором функция вызывает сама себя, организуя таким способом цикл (здесь функция ***fib*** вызывает себя дважды). Рекурсию также можно считать основным приёмом в функциональном программировании, в котором другие способы создания циклов почти совсем не применяются. Групповой знак ***(_)*** принят для случая, когда сопоставление будет удачным всегда. В нашем примере варианту ***case _*** соответствуют все варианты, кроме ***x = 0*** и ***x = 1***. Допустимо в Scala третий ***case*** записать и так:

case n => fib(n - 1) + fib(n – 2)

то-есть, можно ввести новую переменную ***n***, которая получает значение по умолчанию, равное исследуемой величине.

В Scala можно выполнять множественную проверку на одной строке:

case 0 | -1 | -2 => 0

Здесь знак ***(|)*** - логическое ***или***.

На самом деле этот код соответствует:

case 0 => 0
case -1 => 0
case -2 => 0

При сопоставлении можно вводить защиту (условие, *guard)*, вставляя дополнительное условие перед знаком ***(=>)***. Так, в функции ***fib*** можно дополнительно учесть не допустимость ввода отрицательного числа. Для этого достаточно изменить первое ***case*** на следующее:

case x if x <= 0 => 0

Теперь для отрицательного числа и нуля результатом будет нуль. Здесь мы использовали тот же идентификатор ***x***, что и аргумент функции, но можно использовать любое обозначение, например ***n***,

как мы это уже сделали выше.

Сопоставлять с образцом можно любые типы данных; это может быть, например, аргумент типа *String*:

```
def f(name: String) = name match {
    case "Elwood" | "Madeline" => Some("Cat")
    case "Archer" => Some("Dog")
    case "Pumpkin" | "Firetruck" => Some("Fish")
    case _ => None
}
print(f("Elwood").get) → Cat
```

Обращаю внимание на то, что для иллюстрации результатом сравнения в данном примере выбраны элементы коллекции *Option* и для получения результата типа *String* мы использовали метод ***get***. Правда, при варианте, когда получим ***None***, метод ***get*** даст исключение и мы уже говорили об этом раньше. Значит, лучше опять использовать метод *getOrElse*:

```
print(f("Elwoodo").getOrElse("No"))
```

Теперь при неудачном варианте мы получим ответ ***No***. В первом и третьем ***case*** использована множественная проверка.

Сопоставлять можно параллельно любые типы данных в одном операторе:

```
def f(x: Any): String = x match {
    case 1 => "One"
    case "David" | "Archer" | Some("Dog") => "Walk"
    case _ => "No Clue"
}
print(f("David")) →  Walk
```

Естественно, что тип аргумента функции надо объявить, как ***Any***, если мы желаем вводить данные разных типов. Результат будет всегда иметь тип ***String***.

Тестировать можно и типы данных:

```
def f(x: Any) = x match {
    case s: String => "String, length "+s.length
    case i: Int if i > 0 => "Natural Int"
    case i: Int => "Another Int"
    case a: AnyRef => a.getClass.getName
    case _ => "null"
}
```

println(f("Hello")) → String, length 5
println(f(-8)) → Another Int
println(f(0.87)) →java.lang.Double
print(f(null)) → null
В каждом ***case*** мы ввели новые переменные: ***s, i, a***; всем им по умолчанию будет передаваться значение аргумента ***x***. Использованный в примере тип ***AnyRef*** почти то же самое, что и тип ***An****y*. Метод ***getClass*** определяет класс объекта, а метод ***getName*** выделяет только имя класса (устраняет слово ***class***). На слова *java.lang.* советую не обращать внимания — это наследие Java. Специальное значение ***null*** не сравниться ни с одним из заданных нами типов.

4.4 Case Classes

Это особая разновидность классов, оптимизированных для использования в операциях сопоставления с образцом. В основном они имеют те же возможности, что и обычные классы, только дополнительно компилятор генерирует для них методы ***toString***, ***hashCode, equals*** (или ==) и ***copy***. О способах использования этих методов нужен отдельный разговор. Можно также объявлять и *case*-объекты.

Для объявления *case*-класса применяется служебное слово *case*:
case class Person(name: String, age: Int, valid: Boolean)
Экземпляры могут создаваться как с использованием метода ***new***, так и без него:
val m = new Person("Martin", 44, true)
или:
val p = Person("David", 45, true)
Все поля класса доступны:
m.name → Martin
p.age → 45
m.valid → true
case class неизменяемы (*immutable*) и все поля только для чтения (*read-only*):
p.name = "Fred" → это выдаст ошибку. При необходимости можно объявлять изменяемые (***var***) поля:
case class MPerson(var name: String, var age: Int)
val mp = MPerson("Jorge", 24)

mp.age = 25 → Это будет работать, но подобных приёмов рекомендуется избегать.

Сопоставление с образцом для экземпляров ***case class*** очень эффективно:

```
case class Person(name: String, age: Int, valid: Boolean)
val q = Person("David", 45, true)
def f(p: Person): Option[String] = p match {
      case Person(name, age, true) if age > 35 => Some(name)
      case _ => None
}
println(f(q)) →Some(David)
val a = f(Person("Fred", 73, false))
println(a)  → None
val b = f(Person("Jorge", 24, true))
print(b) → None
```

Функция ***f*** принимает аргумент ***p*** типа ***Person*** и возвращает результат типа ***Option[String]***. Мы уже знаем, что можно делать с таким результатом. Впрочем, можно было и не указывать тип результата в данном контексте. Далее на место аргумента мы подставили новый экземпляр класса ***Person***,
и он не прошёл сопоставление по параметру ***valid = false*** (мы его указали при сравнении явно), а результат оказался равным ***None***. В последнем случае экземпляр не прошёл *guard* по параметру ***age*** и функция опять вернула значение ***None***.

Несколько слов о методе ***copy***. С помощью этого метода можно создавать копии объектов *case*-классов с теми же значениями полей, например:

```
case class Person(name: String, age: Int, valid: Boolean)
val q = Person("David", 45, true)
```

val r = q.copy() - скобки обязательны.

Сама по себе эта операция не приносит большой пользы, но используя именованные параметры, можно изменять некоторые свойства объекта:

val r = q.copy(name = "Marta", age = 20) → Person(Marta,20,true)

4.5 Сопоставление с образцом в списках (*Pattern Matching in Lists)*

Мы уже знаем, что списки могут создаваться с помощью метода

(::), в Scala его называют ***cons cell***.
Val a = 1 :: 2 :: 3 :: Nil → List(1, 2, 3)
Первый элемент списка называют «голова» (*head)*, а всю оставшуюся часть списка - «хвост» *(tail)*. Если, например, мы имеем список:
val a = List(1, 2, 3, 4), то выделить ***head*** и ***tail*** этого списка можно так:
val h :: t = a – получим***:***
h:Int = 1 – это голова
t:List[Int] = List(2, 3, 4) – это хвост

Оператор *case* сам способен выделять *head* и *tail* из списка. Делается это так:

```
val a = List(1, 2, 3, 4)
a match {
      case h :: t => println(h); print(t);
      case _ => None}
```

Будет выведено:

```
1
List(2, 3, 4)
```

Теперь рассмотрим более содержательный пример. Пусть требуется найти сумму всех нечётных элементов списка:

```
val a = List(1,2,3,4)
def f(s: List[Int]): Int = s match {
      case Nil => 0
      case h :: t if h % 2 == 1 => h + f(t)
      case _ :: t => f(t)
}
println(f(a)) → 4
```

Здесь снова цикл организован с помощью рекурсии. Сопоставление с ***Nil*** проверяет условие исчерпания списка. Второе ***case*** выделяет ***head*** и если это нечётное число, то оно прибавляется к сумме. Последнее ***case*** соответствует чётным числам и в этом случае функция ***f*** вызывается снова с аргументом, равным ***tail****;* при этом к сумме ничего не добавляется. Так будут последовательно перебраны все элементы.

Scala способен выделять больше одного начального элемента из списка:

```
val a = List(1, 2, 3, 4, 5)
```

val h :: t :: r = a → получим:
h: Int = 1
t: Int = 2
r: List[Int] = List(3, 4, 5)
Воспользуемся этим качеством для устранения дублирующихся элементов из списка:

```
def f[T](s: List[T]): List[T] = s match {
case Nil => Nil
case h :: t :: rest if h == t => f(h :: rest)
case h :: rest => h :: f(rest)
}
val b = f(List(1,2,3,3,3,4,1,1))
println(b) → List(1, 2, 3, 4, 1)
```

У параметров функции мы задали неопределённый тип ***T*** и это позволяет нашей программе работать со списками любых типов. Отметим также, что программа способна устранять дублирование только для рядом стоящих элементов (единица входит в результирующий список дважды).

Рассмотрим ещё один аналогичный пример. Пусть имеется список с элементами *String* и надо удалить все элементы стоящие перед элементом *"ignore" :*

```
def f(s: List[String]): List[String] = s match {
        case Nil => Nil
        case _ :: "ignore" :: t => f(t)
        case x :: t => x :: f(t)
}
val a = List("aa", "bb", "ignore", "cc")
print(f(a)) → List(aa, cc)
```

Во втором ***case*** мы не обращая внимания на то, какой именно элемент стоит на первом месте, проверяем находится ли на втором месте элемент ***"ignore"***. И если да, то мы рекурсивно снова вызываем функцию ***f***, но передаём ей список элементов после ***"ignore"***, отбрасывая, таким образом первый элемент и сам элемент ***"ignore"***. Если это сравнение не сработало, то дальше мы голову оставляем в списке и тоже возвращаемся к ***f,*** передавая ей хвост.

Ещё пример. Пусть в списке с элементами смешанных типов надо оставить только элементы типа *String:*

```
def f(x: List[Any]): List[String] = x match {
case Nil => Nil
case (s: String) :: t => s :: f(t)
case _ :: t => f(t)
}
val a = List(1, "Hello", 2.7, true, "Marta")
print(f(a)) → List(Hello, Marta)
```

Объявленный тип элементов теперь ***Any***. Если первый элемент имеет тип ***String***, то мы оставляем его в списке и повторяем анализ для хвоста. Если же первый элемент не ***String***, то мы его отбрасываем и тоже повторяем анализ для хвоста.

Обычно ***case*** используется вместе с ***match***. Например, эту конструкцию можно применить в блоке для итератора, в частности, для ***filter***. Пусть требуется оставить в списке только элементы, имеющие тип ***String***, удалив все остальные:

```
val a = List(1, "aa", 3.4, "bb")
val b = a.filter(a => a match {
        case s: String => true
        case _ => false
})
print(b) →List(aa, bb)
```

Метод ***filter*** оставит в списке только те элементы, для которых блок вернёт ***true***. Кроме того, Scala позволяет иногда использовать ***case*** без ***match***; в частности, эту программу можно представить в более лаконичной форме:

```
val a = List(1, "aa", 3.4, "bb")
val b = a.filter {
        case s: String => true
        case _ => false
}
print(b) →List(aa, bb)
```

Для иллюстрации приведём ещё пример с итератором ***map***. Пусть надо в списке чётные элементы возвести в квадрат, а нечётные оставить без изменений:

```
val a = List(2, 7, 3, 4, 6, 5)
val b = a.map{
        case s if s % 2 == 1 => s
        case s if s % 2 == 0 => s * s
```

```
}
print(b) → List(4, 7, 3, 16, 36, 5)
```

*И*тератор ***map*** дальше рассмотрим отдельно.

4.6 Сопоставление с регулярными выражениями (*Matching on Regular Expressions*)

Регулярные выражения применяются во всех современных языках программирования для анализа и преобразования текста. Изучение самих регулярных выражений — особая тема и вряд ли целесообразно рассматривать её здесь. Отметим тоько, что в Scala оператор ***case*** позволяет выполнять сравнение с образцом по регулярному выражению:

```
val re = "Изучаем ([Ss]cala)".r
val a = "Изучаем Scala"
val b = "Изучаем scala"
val c = "Изучаем Ruby"
val l = List(a,b,c)
for (x <- l)  {
      x match  {
            case re(t) => println("Изучаем " + t)
            case foo => println("Выход")
            }
      }
```

Результат:

```
Изучаем Scala
Изучаем scala
Выход
```

Здесь переменная ***re*** – образец для сравнения, содержащий регулярное выражение, в которое метод ***r*** переводит строку. Регулярное выражение *(**[Ss]cale**)* соответствует словам ***Scala*** и ***scala*** (***[Ss]*** означает, что допустима любая из букв: ***S*** или ***s***)*.* В операторе ***case re(t)*** переменная ***x*** сопоставляется с образцом, а введённая тут переменная ***t*** принимает значение, равное тексту, который при сравнении с регулярным выражением дал ***true***. Этому условию удовлетворяют переменные ***a*** и ***b***. Переменная ***c*** при сравнении даст ***false*** и вывода на экран не будет. ***Case foo*** введено для исчерпания альтернатив, без этого интерпретатор зафиксирует ошибку. Кстати, переменная ***foo*** при этом будет равна тексту

"изучаем Ruby"и её можно использовать справа от знака равенства.

4.7 Более содержательный пример с *pattern-matching*

Составим программу вычисления площади круга, квадрата и прямоугольника. Без применения ***pattern-matching*** она может выглядеть примерно так:

```
trait OShape {
        def area: Double
}
class OCircle(radius: Double) extends OShape {
        def area = radius * radius * math.Pi
}
class OSquare(length: Double) extends OShape {
        def area = length * length
}
class ORectangle(h: Double, w: Double) extends OShape {
        def area = h * w
}
val p = new OCircle(3.0)
print(p.area) → 28.27
```

(Мы ограничились только площадью круга). Здесь нет ничего нового. Трейт ***Oshape*** можно было и не объявлять вовсе, а каждый класс просто использовать независимо от других. Трейт этот мог быть полезен, если бы мы решили, например, создать список с элементами — экземплярами этих классов, которые имели бы тогда тип *Oshape*:

```
val p = new OCircle(3.0)
val p1 = new OSquare(3.0)
val p2 = new ORectangle(3.0, 4.0)
val a:List[OShape] = List(p, p1, p2)
```

А теперь напишем эту программу с применением *pattern-matching*:

```
trait Shape
        case class Circle(radius: Double) extends Shape
        case class Square(length: Double) extends Shape
        case class Rectangle(h: Double, w: Double) extends Shape
object Sh {
```

```
    def area(s: Shape): Double = s match {
        case Circle(r) => r * r * math.Pi
        case Square(l) => l * l
        case Rectangle(h, w) => h * w
    }
}
print(Sh.area(Circle(3.0)))  →28.27
```

Теперь трейт нужен для того, чтобы объединить все три класса «под одним знаменем», только в этом случае мы можем объявить тип аргумента у функции ***area - area(s: Shape)***. Теперь можно добавлять и какие-то другие вычисления в этих классах, например, можно вычислить периметр этих геометрических фигур:

```
def perimeter(s: Shape) = s match {
    case Circle(r) => 2 * Math.Pi * r
    case Square(l) => 4 * l
    case Rectangle(h, w) => h * 2 + w * 2
}
```

Я думаю, трудно удержаться, чтобы не сказать здесь: «замечательная вещь этот ***pattern-matching*** в Scala!».

Глава 5 Итераторы (*Iterators*)

5.1 Варианты итераторов и их применение

Очевидно, что программируя циклы, можно выполнить любые действия над коллекциями. Но в Scala, как и в большинстве современных языков, имеется группа методов, существенно облегчающих работу с коллекциями. Обычно их называют итераторами и, в сущности, они относятся тоже к управляющим структурам. В Scala подобных методов так много и им отведена такая большая роль, что мне показалось целесообразным отвести для них отдельную главу. Кое-что повторим, но, как говорится, повторение — мать учения.

Все основные методы для работы с коллекциями реализованы в трейте ***Iterable*** (доступен автоматически). Они применимы к коллекциям всех видов, если, конечно, в конкретном случае не противоречат смыслу. Сначала рассмотрим несколько простых методов (не итераторов).

Методы ***head, last*** (ранее уже применяли)*:*

val a = List("Anna", "Marta", "Iren")

a.head → Anna – метод ***head*** возвращает первый элемент (голову).

a.last → Iren – соответственно, последний элемент.

Методы ***headOption, lastOption:***

Методы ***head, last*** для пустого списка генерируют исключение и для защиты следует применять ***headOption, lastOption***, которые отличаются только тем, что возвращают элементы коллекции *Option:*

val b = a.headOption →Some(Anna)

b.getOrElse("guard") → Anna

Для пустого списка эти методы вернут значение ***None****:*

val c = List()

val b = c.headOption → None

b.getOrElse("guard") →guard

Методы ***tail, init:***

Мы уже видели, что ***tail*** возвращает список без первого элемента (хвост), а ***init*** – без последнего:

a.tail →List(Marta, Iren)

a.init →List(Anna, Marta)

Применение этих методов к списку с одним элементом вернёт пустой список, а к пустому списку - вызовет исключение.

Метод ***length*** возвращает длину списка (это мы уже использовали), а метод ***isEmpty*** возвращает ***fals****e*, если список не пустой и ***true*** - в противном случае.

Теперь методы-итераторы, реализующие цикл. Все они имеют однотипную конструкцию:

имя коллекции.имя итератора(блок)

Блок может выглядеть по разному, но по существу это обычно анонимная функция, которую мы позже рассмотрим детально. Объединим итераторы в отдельные группы.

1) ***foreach, map, flatMap, collect***

val a = List(1,2,3,4,5)

a.foreach(x => print(x * x + ", ")) → 1, 4, 9, 16, 25,

Здесь блок — типичная анонимная функция: аргумент ***x***, знак ***(=>)***, какие-то действия над аргументом (в нашем примере выводится на печать переменная ***x*** в квадрате). Метод реализует перебор всех элементов коллекции в цикле, передавая их переменной цикла ***x*** и

выполняя над ними операции, запрограммированные в блоке. Кстати, на состав этих операций не накладывается никаких ограничений; тут может быть несколько операторов через знак ***(;)***, условные выражения и так далее. Метод ***foreach*** не возвращает никаких результатов (точнее возвращает ***Unit***, мы это уже встречали) и применяется только ради побочного эффекта. Так, в блоке можно запрограммировать изменение исходной коллекции, если она *mutable:*

var a = List(1,2,3)
a.foreach(x => a = a ++ List(x*x)) → a: List[Int] = List(1, 2, 3, 1, 4, 9)

В каждом цикле к списку ***a*** добавляем список с одним элементом ***x*x***.

А можно и так:

a.foreach(x => a = a.::(x*x)) → a: List[Int] = List(9, 4, 1, 1, 2, 3)

Итератор ***map*** делает всё то же, что и ***foreach***, но кроме того, он ещё и формирует новую коллекцию из результатов, возвращаемых блоком:

val b = a.map(x => x * x) → List(1, 4, 9, 16, 25)

Вид возвращаемой коллекции соответствует виду исходной:

val m = Array(3.1, 7.0, 4.02, 0.8)
val n = m.map(s => math.sin(s)) →Array(0.0415, 0.6569, -0.7697, 0.7173)

Теперь переменная цикла обозначена ***s***. Можно вообще заменять идентификатор на ***(_)*** и тогда даже не требуется ставить знак ***(=>)****:*

val n = m.map(math.sin(_)) →Array(0.0415, 0.6569, -0.7697, 0.7173)

Количество элементов, в возвращаемом списке всегда такое же, как в исходном. Но тип элементов в результате может быть другим, это определяется функцией блока:

val a = List[String]("A", "Cat").map(_.length) → a: List[Int] = List(1, 3)

Метод ***length*** возвращает результат типа ***Int***, а исходный список имел элементы типа ***String***.

Итератор ***flatMap*** позволяет вложенные списки (или другие коллекции) развернуть (раскрыть) в список:

val a = List(List(1, 2, 3), List(2, 4, 6), List(3, 6, 9))
a.flatMap(x => x) → List(1, 2, 3, 2, 4, 6, 3, 6, 9)

Приведём более интересный пример:

def isOdd(in: Int) = in % 2 == 1
def isEven(in: Int) = !isOdd(in)
val n = (1 to 5).toList → List(1, 2, 3, 4, 5)
val r = n.filter(isEven).map(i => n.filter(isOdd).map(j => i * j)) → List(List(2, 6, 10), List(4, 12, 20))

Сначала мы объявили две функции: ***isOdd*** возвращает ***true*** для нечётных чисел, а ***isEven*** — для чётных. Список ***n*** мы создали с помощью ранга и метода ***toList***. Метод ***filter*** (итератор) отбирает те элементы из списка, для которых блок возвращает ***true***. Дважды использованный ***map*** формирует список из двух списков с элементами, равными произведениям нечётных чисел на чётные. Теперь, если первый ***map*** заменить на ***flatMap***, то вложенные списки будут развёрнуты:

val r = n.filter(isEven).flatMap(i => n.filter(isOdd).map(j => i * j)) → List(2, 6, 10, 4, 12, 20)

Отметим, что можно запрограммировать всё это и с помощью цикла ***for*** с применением директивы ***yield*** и, пожалуй, даже более элегантно:

for {i <- n if isEven(i); j <- n if isOdd(j)} yield i * j → List(2, 6, 10, 4, 12, 20)

Как работает такой цикл, мы уже обсуждали ранее.

Метод ***collect*** работает с частично определёнными функциями (*partial functions)* – они применимы не для всех входных значений.

"-3+4".collect {case '+' => 1; case '-' => -1 } → Vector(-1, 1)

Здесь итератор ***collect*** применяется к тексту *(String)*, который в Scala рассматривается, как частный случай коллекций. В блоке использованы ***case***; ранее уже говорилось о таком применении оператора ***case***. Если рассматривать блок, как функцию, то она не определена для всех знаков, кроме ***(-)*** и ***(+)*** и итератор ***collect*** обработал только эти знаки. Результат здесь - вектор, но его можно обратить и в список:

"-3+4".collect {case '+' => 1; case '-' => -1 }.toList →List(-1, 1)

2) ***reduceLeft, reduceRight, foldLeft, foldRight***

Эти итераторы — двухместные, в том смысле, что блок для них обрабатывает два элемента коллекции.

val a = List(6,2,9,4,5)
val b = a.reduceLeft((x, y) => x — y) → b: Int = -14

Анонимная функция принимает два аргумента, которым

последовательно передаются все элементы коллекции и вычисление проводится по такой схеме:
(((6 — 2) — 9) — 4) — 5 = -14
Напишем это же с применением placeholder:
val b = a.reduceLeft(_ - _) → b: Int = -14

reduceRight работает точно также, только обработка коллекции выполняется справа-налево:
val b = a.reduceRight(_ - _) → b: Int = 14 а именно:
5 – (4 – (9 – (2 – 6))) = 14
Если заменить ***(-)*** на ***(+)***, то оба итератора вернут один и тот же результат, так как сумма не зависит от перестановки слагаемых. Один и тот же результат будет и для произведения.

Итераторы ***foldLeft*** и ***foldRight*** работают в общем-то так же, только дополнительно они принимают аргумент, определяющий значение, с которого начинается вычисление:
val b = a.foldLeft(0)(_ - _) → b: Int = -26 здесь схема такова***:***
((((0 — 6) — 2) — 9) — 4) -5 = -26
val b = a.foldLeft(10)(_ - _) → b: Int = -16

foldLeft допускается заменять знаком ***(/:)*** - это просто синтаксический сахар:
val b = (10 /: a)(_ - _) → b: Int = -16

3) ***sum, product, max, min***

Методы ***sum*** и ***product***, вычисляют сумму и произведение элементов коллекции:
a.sum → 26
a.product → 2160
А методы ***max*** и ***min*** возвращают наибольший и наименьший элемент:
a.max → 9
Для двух величин эти методы записываются так:
3 max 5 → 5 ***3 min 5 → 3***

4) ***reduce, fold***

Эти итераторы применяют двухместную операцию ко всем элементам в произвольном порядке:
a.reduce((x, y) => x – y) → -14
a.reduce((x, y) => y – x) → 14
То-есть, для изменения порядка достаточно переставить местами переменные в блоке. То же самое и для ***fold***

a.fold(0)((x, y) => x – y) → -26
a.fold(0)((x, y) => y – x) → 14

5) ***count, forall, exists***

Эти итераторы возвращают количество элементов коллекции, удовлетворяющих условию или проверяют соответствие коллекции заданному условию.

val s = "1 October 2015"
s.count(x => x.isDigit) → 5
s.count(x => x.isLetter) → 7

Методы ***isDigit, isLetter*** возвращают ***true*** для цифр и букв соответственно. Более краткая запись:

s.count(_.isLetter) → 7

Ещё пример:

val a = List(1,2,3,4,5)
a.count(_ > 3) → 2 – число элементов удовлетворяющих условию

forall позволяет проверить, удовлетворяют ли все элементы условию:

s.forall(x => x.isDigit) → false – не все знаки — цифры

exists проверяет, удовлетворяет ли хотя бы один или более элементов заданному условию:

s.exists(x => x.isDigit) → true – в тексте есть цифры

6) ***filter, filterNot, partition***

filter возвращает все элементы, удовлетворяющие условию:

s.filter(x => x.isLetter) → October

filterNot наоборот, все, не удовлетворяющие условию:

s.filterNot(x => x.isLetter) → 1 2015

Ещё пример: пусть нам нужно отобрать из списка чётные числа:

val a = List(4, 3, 27, 24, 8, 9)
val b = a.filter(x => x % 2 == 0) → b: List[Int] = List(4, 24, 8)
val b = a.filter(_ % 2 == 0) *– то же самое*

В блок можно ввести какую-нибудь функцию и она будет последовательно применяться к элементам списка:

def isOdd(x: Int) = x % 2 == 1 → isOdd: (x: Int)Boolean
val b = a.filter(isOdd) → b: List[Int] = List(3, 27, 9)

Теперь отобраны нечётные числа. Как видим, не потребовалось даже указывать аргумент у введённой в блок функции. Хотя, конечно, можно было бы и так:

val b = a.filter(x => isOdd(x))

или так:
val b = a.filter(isOdd(_))
partition возвращает пару (кортеж), где в первом элементе всё, что удовлетворяет условию, а во втором — всё, что нет:
val p = s.partition(x => x.isLetter) → p:(String, String) = (October,1 2015)
При этом тип результата определяется типом коллекции.
val a = List(4, 12, 8, 1)
val p = a.partition(x => x > 5) → p: (List[Int], List[Int]) = (List(12, 8),List(4, 1))
В первом примере получили пару с элементами типа ***String***, а во втором — с элементами в виде списков типа ***Int***.

7) ***reverse, mkString, take, drop, splitAt, takeRight, dropRight, slice***

reverse меняет порядок расположения элементов на обратный:
a.reverse → List(1, 8, 12, 4)
"Hello world!".reverse → !dlrow olleH

take возвращает первые ***n*** элементов:
val a = List(4, 12, 8, 1, 5)
a.take(2) → List(4, 12)

mkString позволяет преобразовывать коллекцию в строку с произвольными разделителями между элементами:
val x = a.mkString("\|") → x: String = 4\|12\|8\|1\|5
val x = a.mkString("<", ":", ">") → x: String = <4:12:8:1:5>
По остальным итераторам просто приведу примеры:
a.drop(2) → List(8, 1, 5)
a.splitAt(2) → (List(4, 12),List(8, 1, 5)) – это кортеж
a.takeRight(2) → List(1, 5)
a.dropRight(2) → List(4, 12, 8)
a.slice(1,3) → List(12, 8)

8) ***toList, toArray, toMap, toSeq, toSet, toStream, toIterable, toIndexedSeq***

Все эти методы трансформируют одни коллекции в другие. Например, метод ***toList*** может преобразовать текст в список, элементами которого являются знаки текста, приведённые к типу *Char:*
val c = "99 Red Balloons".toList → c: List[Char] = List(9, 9, , R, e, d, , B, a, l, l, o, o, n, s)

Метод ***isDigit*** принимающий переменную типа ***Char*** и возвращающий ***true*** если она является цифрой (уже использовался) позволяет отобрать из текста цифры:

val s = "99 Red Balloons".toList.filter(x => x.isDigit) → s: List[Char] = List(9, 9)

Таким образом в нашем примере из текста отбираются цифровые знаки (тип *Char*), собранные в список. Если ввести знак отрицания ***(!)***, то будут отобраны, наоборот, не цифровые знаки:

val s = "99 Red Balloons".toList.filter(x => !x.isDigit) → s: List[Char] = List(, R, e, d, , B, a, l, l, o, o, n, s)

Мы рассмотрели только часть методов для работы с коллекциями. Полный их список и подробное описание можно найти в документации, которая расположена в папке: */scala/api/scala-library/scala/collection*

5.2 Сортировка коллекций

Scala имеет метод ***sorted***, позволяющий сортировать коллекции:

val a = List(99, 2, 1, 45)

val b = a.sorted → b: List[Int] = List(1, 2, 45, 99)

Метод ***sorted*** сортирует по возрастанию. Чтобы отсортировать по убыванию, можно воспользоваться методом ***reverse***, меняющий порядок расположения элементов на обратный:

val b = a.sorted.reverse → b: List[Int] = List(99, 45, 2, 1)

Есть и ещё один сортировщик — ***sortWith***, принимающий блок, где можно задать нужный порядок сортировки:

val b = a.sortWith((i, j) => j < i) → b: List[Int] = List(99, 45, 2, 1)

val b = a.sortWith((i, j) => j > i) → b: List[Int] = List(1, 2, 45, 99)

Как обычно, можно и так:

val b = a.sortWith(_ < _) →b: List[Int] = List(99, 45, 2, 1)

Сортировать можно списки с элементами любого типа:

val v = List("b", "a", "elwood", "archer")

val w = v.sorted → w: List[String] = List(a, archer, b, elwood)

И текст по буквам тоже:

"Hello, World!".sorted → res: String = " !,Hwdellloor"

Можно сортировать списки по каким-нибудь свойствам элементов, например, строковые элементы можно сортировать по их длине:

val v = List("bbb", "a", "elwood", "arch")

val w = v.sortWith((i, j) => j.length > i.length) → w: List[String] =

List(a, bbb, arch, elwood)

Можно произвольным образом комбинировать операции. Создадим опять трейт *Person*:

```
trait Person {
        def age: Int
        def first: String
        def valid: Boolean
}
```

Создадим теперь объекты *Person*:

```
val d = new Person {def age = 25; def first = "David"; def valid = true
}
```

и так далее (не будем приводить громоздкий текст). Сформируем из этих объектов список и запрограммируем функцию ***f***, принимающую этот список в качестве параметра:

```
def f(in: List[Person]) = in.filter(_.valid).sortWith((i, j) => j.age >
i.age).map(_.first)
```

Эта функция выберет из списка объекты, у которых ***valid = true***, отсортирует их по возрасту ***age*** и вернёт список, элементы которого будут представлять имена ***first*** выбранных и отсортированных персон.

Имеется возможность сортировать массивы «на месте»:

```
val b = Array(2, 7, 1, 6, 3, 2, 9)
util.Sorting.quickSort(b) → b:Array[Int] = Array(1, 2, 2, 3, 6, 7, 9)
```

Массив ***b*** изменился в результате сортировки. Здесь мы применили сортировальщик ***quickSort*** из библиотеки ***util***.

Глава 6 Функции и поля

6.1 Functions, apply, update

Scala – функциональный язык, он позволяет одним функциям принимать другие функции, как параметры. Любая функция это блок кода, принимающий аргументы и возвращающий результат. В Scala функции, кроме того, всегда экземпляры (объекты). Когда функция передаётся в качестве аргумента, то она фактически рассматривается, как трейт (интерфейс); их роль часто выполняют анонимные функции. При передаче функции применяются так называемые ”Function” -ы. Так трейт, определяющий функцию —

объект и принимающий один аргумент, имеет вид:
Funcnion1[A, B]
Здесь ***A*** – тип аргумента, а ***B*** – тип результата. Если аргументов два, то Function будет иметь вид:
Function2[A, B, C] – ***A*** и ***B*** — типы аргументов, ***C*** – тип результата. Аналогично можно использовать ***Function3*** и так далее. Для вызова этих функций — трейтов применяется специальный метод ***apply***. Покажем всё это на примере:
val f = {x:Int => math.sqrt(x)}
def fi(a:Int, t: Function1[Int => Double]) = t.apply(a)
print(fi(5, f)) → 2.236
В первой строке мы определили анонимную функцию, присвоив ей идентификатор ***f***, как обычной переменной. Функция ***fi*** принимает список аргументов, включающий параметр ***a*** типа ***In****t* и ***Function***, обозначенный ***t***. В правой части для этого Function ***t*** вызывается стандартный метод ***apply***, принимающий аргумент ***a***. При вызове метода ***fi*** ему передаётся значение ***a*** и блок ***f***. Тип аргумента и результата у блока ***f*** должны соответствовать тем типам, что указаны для Function ***t***. Фактически можно опускать явный вызов метода ***apply***, передавая аргументы прямо трейту ***t***:
def fi(a:Int, t:Function1[Int, Double]) = t(a)
Метод ***apply*** при этом будет вызван компилятором неявно. В Scala имеется много подобного «синтаксического сахара», в частности ***Function1[Int, Double]*** можно заменить на ***Int => Double***:
def fi(a:Int, t: Int => Double) = t(a)

Как и всякий объект, функция имеет тип и он указывается, например, так:
fi: Double => Double
То-есть, функция ***fi*** принимает один аргумент типа *Double* и возвращает результат тоже типа ***Double***. Теперь ***fi*** можно передать другой функции:
def f(fi: Double => Double) = fi(_)
Поскольку явно аргумент не указан, то используется групповой символ ***(_)***. А теперь можно вызвать функцию ***f*** с конкретным аргументом:
f(math.sin)(2.0) → 0.909297
При этом функция ***f*** имеет тип:
f: (Double => Double) => Double

Функции, принимающие другие функции, называются функциями высшего порядка. Они могут и возвращать другие функции:

def f(n:Int) = (x:Int) => n * x → f: (n: Int)Int => Int

Скобки в выражении ***(x:Int)*** необходимы, без них это будет восприниматься не как функция, а как простая переменная ***x*** типа ***Int***. Теперь вызов ***f*** с любым аргументом вернёт функцию и ей можно присвоить идентификатор:

def fi = f(3) → fi: Int => Int

fi – обычная функция, принимающая один аргумент:

fi(9) →27

Отметим ещё, что функцию ***f*** можно вызывать, как каррированную функцию (смотрите далее):

f(3)(9) → 27

Если в класс вставить стандартный метод ***update***, то можно организовать вывод (там, где это будет нужно) какой-нибудь шаблонной фразы, вроде той, что в следующем примере:

class Up { def update(k:String, v:String) = println("Привет, " + k + " " + v) }

val u = new Up

u("дорогой") = "Александр" → Привет, дорогой Александр!

Фраза всегда будет выведена на экран, если только выполнить присваивание, подобно тому, как в нашем примере. Придумайте самостоятельно, где это могло бы быть полезно.

Применять метод ***update*** можно в разных вариантах, как это иллюстрирует следующий пример:

```
class Update {
      def update(what: String) = println("Singler: "+what)
      def update(a: Int, b: Int, what:String) = println("2d update " + what)
}
val u = new Update
u() = "Foo" → Singler: Foo
u(3, 4) = "Howdy" → 2d update Howdy
```

Значит, механизм ***update*** работает с разными параметрами и умеет выбирать нужный вариант.

Разумеется, что в правой части метода ***update*** может быть блок, в котором можно запрограммировать всё, что угодно:

```
class Up { def update(k:Double, l:Double, v:Double) = {
        val x = math.cos(v)
        x * k * l
       }
}
val u = new Up
val y = (u(2.0, 1.0) = 0.75)
print(y) → 1.4633777377476418
```

Обратите внимание на то, что если список аргументов метода ***update*** больше двух, то надо все, кроме последнего сделать аргументами объекта ***u***, а последний — поставить в правой части (почему так, я думаю, знает только один автор языка Scala). На начальной стадии обучения все эти приёмы кажутся несколько замысловатыми, но потренировавшись их можно усвоить и в действительности они могут быть полезными.

Любой фрагмент программы, заключённый в фигурные скобки, является блоком. Блок всегда возвращает последнее вычисленное значение. Надо иметь ввиду, что если последнее действие блока — присваивание, блок вернёт тип ***Unit***:

```
var r = 2; var n = 3
val x = { r = r * n;  r -= 1 } → x: Unit = ()
```

при этом ***r*** будет равно ***6***, а ***n*** равно ***2***.

Если написать

var r = n = 1 то получим ***r: Unit = ()*** по той же причине. При этом переменная ***n*** будет инициирована правильно и равна ***1***.

6.2 Поля класса и доступ к ним

По умолчанию все поля, методы, да и сами классы в Scala общедоступны (*public*) и объявлять их таковыми не требуется (хотя и допустимо). Если в классе объявить переменную (поле) с ***var***, то её можно читать и изменять в любом экземпляре класса беспрепятственно.

```
class A  { var x = 0  } → переменная должна быть инициализирована
val p = new A
p.x = 77
print(p.x) → 77
```

Таким образом, методы доступа к полю ***x*** создаются автоматически. Такой способ программирования в языках ООП не

одобряется, так как все и всегда могут изменять общедоступные поля (иногда поля, имеющие методы доступа называют свойствами класса, значит, ***x*** — свойство класса ***A***). Если объявить поле с ***val***, метод доступа по записи (по изменению поля) не будет создан:
class A { val x = 0 }
val p = new A
p.x = 77 → ошибка
print(p.x) → ***0*** - Метод по чтению имеется
Если поле объявит приватным (***private***), поля доступа вообще не будут созданы:
class A { private var x = 0 }
val p = new A
p.x = 77 → ошибка
print(p.x) → ошибка
Но теперь их можно создать самостоятельно:
class A { private var x = 0; def setx(s:Int) {x = s}; def getx = x }
Мы создали метод доступа по записи ***setx*** и по чтению — ***getx***.
val p = new A
p.setx(77)
print(p.getx) → ***77***
В принципе это мало отличается от варианта объявления поля с ***val***, однако теперь есть возможность этими методами управлять; например, можно ввести дополнительное условие в метод *setx*
class A { private var x = 0; def setx(s:Int) {if (s > 25) x = s}; def getx = x }
val p = new A
p.setx(77)
print(p.getx) → ***77***
p.setx(18)
print(p.getx) → ***77*** (поле ***x*** не может иметь значение меньше ***25***)
С помощью имеющихся средств можно комбинировать разные методы доступа к полям класса (менять его свойства).

В Scala есть возможность получить доступ к приватным полям сразу всех экземпляров класса:
class A { private var x = 0; def setx(s:Int) { x = s }; def getx(other:A) = other.x }
val p = new A
p.setx(25)

val q = new A
q.setx(77)
print(p.getx(q)) → 77

Методу доступа по чтению ***getx*** в качестве аргумента передана переменная ***other***, имеющая тип ***A*** (создавая класс, мы создаём новый тип). Представителем этой переменной могут быть экземпляры класса ***A***. В правой части метода ***getx*** возвращается поле ***x*** полученного методом какого-то другого экземпляра класса ***A*** (у нас экземпляра ***q***).

Доступ к полям других экземпляров (объектов) класса можно запретить, объявив поле с квалификатором ***private[this]***:

private[this] var x = 0

Теперь метод ***def getx(other:A) = other.x*** не допустим.

Мы уже видели, что классам можно предавать параметры. В методах класса эти параметры доступны с ключевым словом ***this***:

class A(x:Int) { def f(x:String) = { println(x + " " + this.x) } }
val p = new A(999)
p.f("aaa") → aaa 999
p.f("bbb") → bbb 999

То-есть, в методе ***f*** свой параметр ***x*** совсем не одно и тоже с параметром класса ***this.x***.

Приведём ещё один пример доступа к приватному полю другого экземпляра класса:

```
Class C {
        private var v = 5
        def inc() { v += 1 }
        def is(ot:C) = v + ot.v
}
val p = new C
p.inc()
val r = new C
print(r.is()) → 11
```

Функция может быть объявлена без знака равенства перед её телом. Такие функции не возвращают значение (они возвращают ***Unit***) и называются процедурами. Процедуры вызываются только ради получения побочного эффекта. В нашем примере процедура — функция ***inc***, её побочный эффект — увеличение на единицу переменной ***v***. В остальном пример не требует пояснений.

6.3 Функции — всегда экземпляры

В Scala функции всегда экземпляры классов, с ними можно делать то же, что и с другими экземплярами. Мы можем создавать функции и инициализировать ими переменные:

val f: Int => String = { x => "Dude: "+x }

Выражение ***Int => String*** указывает, что функция принимает аргумент типа ***Int*** и возвращает результат типа ***String***. В правой части — блок (анонимная функция). Если эту строку ввести в интерактивном режиме, получим такой ответ:

f: Int => String = <function1>

Объекты ***function*** мы уже применяли ранее; единица тут указывает на число аргументов. К ***f*** можно обращаться, как к обычной функции:

val x = f(55) → x: String = Dude: 55

К функциям можно применять операции сравнения: ***f == 6 → false***. К ним даже можно применять методы, например:

f.toString → res:String = <function1>

Получили строковую переменную, не очень ясно только, что с ней дальше можно делать.

Применим теперь эту функцию ***f*** в качестве аргумента другой функции:

val x = 42

def w(fi: Int => String) = fi(x) → w: (fi: Int => String)String

val r = w(f) → r: String = Dude: 42

Фактически тут используются неявно ***Function*** и ***apply***, мы уже рассматривали, как именно. А теперь покажем, что передать функцию, как аргумент можно и по другому:

val r = w((x:Int) => f(x)) → r: String = Dude: 42

То-есть, мы передаём функции ***w*** блок, в котором использована другая функция. Такая конструкция делает код более наглядным (readable). Можно использовать автоматический вывод типов и записать так:

val r = w(x => f(x)) → r: String = Dude: 42

Можно и дальше сокращать запись, применяя, как обычно, placeholder, но для этого надо объявить передаваемую функцию ***f*** со словом ***def***:

def f(x:Int)= "Dude: " + x

val r = w(f _) → r: String = Dude: 42

Но и знак подчёркивания в данной ситуации можно опустить, компилятор сам сможет сделать все преобразования:

val r = w(f) →r: String = Dude: 42

Однако, все эти варианты всё тот же синтаксический сахар, на самом деле компилятор каждый раз выполняет такой код:

```
val r = w(new Function1[Int, String] {
      def apply(i: Int) = f(i)
})
```

Блок, передаваемый функции может быть сложным и состоять не из одной строки кода:

```
val x = 5
def w(fi: Int => String) = fi(x)
val r = w {x => val d = 1 to x
      d.mkString(",") } → r: String = 1,2,3,4,5
```

Мы создали в блоке ранг ***d*** и применили к нему метод ***mkString***, фактически являющийся итератором, который вставляет между элементами коллекции заданный знак (у нас запятая). Затем блок передали функции ***w*** в качестве аргумента.

Функции можно сохранять в переменных:

val f = Math.sin _ → f: Double => Double = <function1>

Знак ***(_)*** за именем функции (через пробел) указывает, что в действительности переменная ***f*** может принимать аргумент и выполняться, как функция:

f(1.5) → 0.997494

Также можно поступать и с функциями, созданными нами:

```
def f(x:Double, y:Double) = x * math.sin(y)
val fi = f _
fi(2,3) → 0.282240
```

Функцию можно передавать, например, методу *map:*

val r = List(1,2,3).map(fi(2, _)) → List(1.68294, 1.818594, 0.282240)

На место ***(_)*** подставляются элементы списка; поменяем аргументы местами:

val r = List(1,2,3).map(fi(_, 2)) → List(0.909297, 1.818594, 2.72789)

6.4 Частичное применение (*Partial Application and Functions*)

Scala в своей основе близок к функциональным языкам программирования, таким, как Haskell. В этих языках любая

функция, принимающая несколько аргументов, может быть приведена к функции одного переменного. Так функция двух переменных может рассматриваться, как функция одного переменного, возвращающая результат в виде другой функции, которая, в свою очередь принимает один аргумент. Такой приём называют частичным применением (*partial application*) или каррированием (*curried*). Создать *partial application* можно из фунуций, например:

def plus(a: Int, b: Int) = "Result is: "+(a + b)

Теперь с помощью этой функции создадим это самое ***curried***, использовав блок (или анонимную функцию):

val p = {(b: Int) => plus(4, b)}

p – функция одного переменного (*partial application* для *plus),* а ***4*** — это параметр, на его место можно подставить, что угодно. Тогда имеем:

val r = p(5) → r: String = Result is: 9

Точно также можно поступить и с функцией трёх переменных:

def f(a:Int, b:Int, c:Int) = a * b * c
val p = {(b:Int, c:Int) => f(2, b, c)}
val q = {c:Int => p(3, c)}
val r = q(4) → r: Int = 24

Приём этот, конечно, очень не хитрый, но он даёт большие возможности в функциональном программировании.

Scala позволяет этот приём ещё более упростить, для чего достаточно лишь аргументы функции сгруппировать по отдельности, заключив в круглые скобки:

def plus(a: Int)(b: Int) = "Result is: "+(a + b)
val r = plus(3)(6) → r: String = Result is: 9

Теперь функцию ***plus*** можно рассматривать, как функцию одного переменного ***a***, возвращающую функцию одного переменного ***b***, то-есть, функция ***plus*** уже является каррированной. Можно объявить новую функцию одного переменного, полученную из этой каррированной функции:

def f(b:Int) = plus(3)(b)
f(6) → 9

Аргументами каррированной функции могут быть блоки с произвольным кодом:

val r = plus(1){ val r = new util.Random; r.nextInt(100); } →

r: String = Result is: 20

Теперь функция ***plus*** складывает единицу со случайным числом из диапазона ***0..100***. Случайные числа генерирует метод ***nextInt*** класса ***Random*** из библиотеки ***util*** (она загружается автоматически).

Каррированная функция ***plus*** может передаваться другой функции, как аргумент:

def w(f: Int => String) = f(4)

val r = w(plus(7)) → r: String = Result is: 11

Если всё это на первых порах не очень понятно, то удовлетворитесь простейшим примером передачи функции, как параметра:

def w(x:Int, f:Int => Int) = x*x + f(x) → w: (x: Int, f: Int => Int)Int

def fi(a:Int) = a/3 → fi: (a: Int)Int

w(4, fi) → 17

Функция ***w*** принимает в качестве аргументов переменную ***x*** типа ***Int*** и функцию ***f***, принимающую один параметр типа ***Int*** и возвращающую результат типа ***Int***. В правой части запрограммировано использование переменной ***x*** и функции ***f***. Дальше объявлена функция ***fi***, принимающая один параметр ***a:Int***. И наконец, вызывается функция ***w***, которой передаются конкретные аргументы. (Напоминаю, что деление целых чисел ***4/3*** даёт в результате ***1***)

6.5 Функции и параметры типа (*Functions and Type Parameters*)

Ранее мы уже упоминали о том, что для создаваемой функции можно задать неопределённый тип результата. Обычно его обозначают буквой ***T***, но можно как угодно. Есть также установленный неопределённый тип ***Any***. Конкретный тип определяется компилятором из контекста. В частности тип может определяться той функцией, которая передаётся, как аргумент.

def w[T](fi: Int => T): T = fi(42) → w: [T](fi: Int => T)T

Для того, чтобы функция могла принимать конкретный тип, как параметр, надо указать неопределённый тип в квадратных скобках после имени функции (перед списком аргументов): ***w[T](…)***. Функция ***w*** принимает в качестве аргумента функцию ***fi***, принимающую аргумент типа ***Int*** и возвращающую результат неопределённого типа ***T***. В правой части задано конкретное

значение аргумента функции ***fi — 42***. Используем снова всё ту же функцию ***f***:
val f: Int => String = x => "Dude: "+x
которая возвращает результат конкретного типа ***String***.
val r = w(f) → r: String = Dude: 42
В результате функция ***w*** вернула результат типа ***String***. Изменим функцию ***f***:
val f: Int => Double = { x => math.sqrt(x) }
val r = w(f) → r: Double = 6.48074
Теперь функция ***w*** возвращает результат типа ***Double***. Такой приём, конечно, полезен, поскольку придаёт функции ***w*** определённую универсальность.

Пусть параметр функции задан по умолчанию:
def f[T](x:T = 9) = print(x)
Чтобы использовать это значение по умолчанию надо вызвать функцию без параметров:
f() → 9
Можно ли задать параметр другого типа, то-есть не ***Int***?
f("Hello") → Hello - работает!
Следовательно, тип можно изменить, если даже задано значение по умолчанию. Можно задавать тип и так:
f[Double](8.9) → 8.9

Раньше уже говорилось о том, что базовые операторы тоже можно трактовать как методы:
val r = 1. +(2) →r: Int = 3
Попробуем передать этот метод функции *w*:
val r = w(3 .+) →r: Int = 45
Функция ***w*** вернула результат типа ***Int.***
А что будет, если использовать ***Double***?
val r = w(3.06.+) → r: Double = 45.06 Всё работает!
Можно работать и со списком:
def w[T](fi: Int => T): T = fi(5)
val f: Int => List[Int] = { i => (1 to i).toList }
Метод ***toLis****t* превращает ранг в список.
val r = w(f) → r: List[Int] = List(1, 2, 3, 4, 5)
В этих примерах тип результата мы не указывали явно, он выводится автоматически в соответствии с вычисленными результатами. Но ничто не мешает указать тип результата явно:

val r = w[List[Int]](f) → r: List[Int] = List(1, 2, 3, 4, 5)
Или так:
val r = w(f):List[Int] → r: List[Int] = List(1, 2, 3, 4, 5)

Вся эта морока типична для языков со строгой типизацией, в которых сначала строгую типизацию создают, а потом героически преодолевают проблемы с ней связанные.

6.6 Побочный эффект

Это понятие имеет значение в функциональном программировании, главное преимущество которого заключается в использовании только «чистых» функций (без побочных эффектов). Для иллюстрации создадим на Scala функцию с побочным эффектом. Объявим пустой ***var*** – список (*mutable*):
var st: List[String] = Nil (можно вместо ***Nil*** писать ***List()***):
и создадим такую функцию:
val f = (s: String) => {st ::= s; s+" Reg"}
Напоминаю, что метод ***(::)*** добавляет элемент в начало списка, а запись ***st ::= s*** то же самое, что и ***st = s :: st***, то-есть элемент добавляется в список и сам список изменяется. Вторая строка блока ***s+" Reg"*** означает, что блок возвращает текст, состоящий из принятого функцией аргумента ***s*** с добавленным к нему словом ***Reg*** (знак (+) в этом контексте - знак конкатенации). Следовательно, и сама функция ***f*** возвращает строку, принятую ею, как аргумент, только с добавленным словом. Если нас только это её свойство и интересует, тогда то, что она при этом ещё и изменяет список ***st*** можно считать побочным эффектом. Посмотрим на применение функции ***f***:
f("a") → a Reg
f("b") → b Reg
Это как раз то, что мы хотели. Но что же при этом происходит со списком ***st***?
print(st) → List(b, a) Порядок обратный потому, что добавление в начало списка. Попробуем использовать ***f*** в качестве блока для итератора ***map***:
List("p", "q", "r").map(f) → List(p Reg, q Reg, r Reg) – работает нормально. Ну, а что же список *st*?
print(st) → List(r, q, p, b, a)
Получается, что функция ***f*** связана с переменной ***st*** и при любом

использовании функции, она этой связи не теряет. Так в нашем примере итератор ***map*** никак не использует список ***st***, а он всё равно изменяется. Интересно, что связь эту можно разорвать, если ***st*** переобъявить со словом ***val*** (применить такое объявление до создания функции ***f*** нельзя, будет ошибка*).*
val st: List[String] = Nil
List("p", "q", "r").map(f) → List(p Reg, q Reg, r Reg)
print(st) → List() список остался пустым — побочного эффекта нет.

6.7 Помещение функций в контейнер (*Putting Functions in Containers)*

Функции это экземпляры и с ними можно делать всё то же, что и с любыми экземплярами. Создадим функцию, принимающую ***Int*** и возвращающую другую функцию:
def bf: Int => (Int => Int) = { i => {v => i + v } } → bf: Int => (Int => Int)
Я заключил блоки в скобки для наглядности, в принципе их можно опускать и тогда они предполагаются по умолчанию. Опять применим эту функцию в качестве блока для ***map***, вызванного для ранга:
val fs = (1 to 10).map(bf).toArray → fs: Array[Int => Int] = Array(<function1>, <function1>, …)
Метод ***toArray*** преобразует список в массив. Результатом является массив размером ***10***, элементы которого — функции принимающие ***Int*** и возвращающие ***Int***. Извлечём первый элемент массива и, поскольку он функция, дадим ему аргумент:
val r = fs(0)(1) → r: Int = 2
В этом варианте для функции ***bf*** параметр ***i*** принимает значение, равное ***1*** (первый элемент ранга), а параметр ***v*** задан равным ***1***.
Всё выполняется также для любого элемента массива:
val r = fs(5)(2) →r: Int = 8 (теперь ***i=6, v=2)***
Если убрать ***toArray***, будет сформирован список, работать с которым можно точно также, как и с массивом. Пример показывает, что функции можно помещать в массив, в список и так далее.

Глава 7 Ввод вывод

7.1 Ввод вывод на экран

О вводе каких-то данных с клавиатуры уже говорилось в самом начале книги. Если надо ввести, например, несколько чисел, лучше всего ввести их в виде переменной типа *String*, а затем разбить её на числа и перевести в нужный тип. Как это делается, мы уже знаем — тексты обрабатываются, как коллекции.

Не форматированный вывод на экран мы выполняли уже очень много раз с помощью операторов ***print*** и ***println***. Для форматированного вывода применяется оператор ***printf***, использующий специальный формат в котором используется знак *(%)*. Комбинация ***%s*** – для вывода текста, ***%d*** – для вывода целых чисел и ***%f*** – для чисел с плавающей запятой. После знака ***(%)*** может указываться необязательный размер поля (количество символов): ***%5s, %8d, %9.5f***. В последнем случае первая цифра — общий размер поля, а после точки — размер поля для дробной части (почему-то между целой и дробной частью принят знак ***(,)*** вместо привычной точки). Все выводимые данные прижимаются к правой границе поля. Знаки форматирования можно вставлять внутри произвольного текста, или без него, но обязательно в кавычках. Формат для печати можно помещать прямо в операторе печати, или создавать специальную переменную формата:

val f = "Hello %9s world %5d Hi %7.2f"
printf(f,"Bob", 123, 53.1234567) → Hello Bob world 123 Hi 53,12

Числа с плавающей запятой могут быть и в научной нотации: ***0.531234567e2*** будет выведено то же самое.

Имеется также возможность выполнить перегрузку метода ***toString***, что позволяет создавать разнообразные формы вывода. Пусть, например, нам надо организовать вывод комплексных чисел в принятой в математике форме. Создадим класс:

```
class Complex(re:Double, im:Double)  {
      override def toString() = " " + re + (if (im<0) " " else "+") + im
+ "i"
}
val p = new Complex(2.3, 5.6)
```

print(p.toString) → 2.3+5.6i
Для отрицательной мнимой части числа:
val p = new Complex(2.3, -5.6)
print(p.toString) → 2.3 -5.6i
В выражении для перегрузки метода использовано условие для того, чтобы организовать вывод знака ***(+)***, так как обычно перед положительными числами знак ***(+)*** не выводится в отличие от знака ***(-)***.

Ну, а вообще, для того, чтобы оформить полноценный ввод и вывод, надо пользоваться графическим интерфейсом — готовым или создать его самостоятельно. Но это уже отдельная тема.

7.2 Работа с файлами

1) Чтение из файла

Прочитать все строки из файла можно с помощью метода ***getLines*** объекта ***Source*** из модуля ***io***. Следовательно, этот модуль надо предварительно импортировать:
import io._
val x = Source.fromFile("rab1.scala", "UTF-8")
val it = x.getLines
Название файла и кодировка должны быть указаны в кавычках. Если файл находится в другой директории, здесь должен быть полный путь, но тоже в кавычках. Кодировку можно не указывать вовсе, если кодировка файла совпадает с кодировкой системы по умолчанию. В результате этих действий будет создан итератор ***it***. Здесь термин «итератор» использован в другом смысле, чем метод для обработки коллекций. Итератор ***it*** является неким заместителем коллекции, который позволяет обработку данных несколько иначе. Вообще без него можно обойтись, но объект ***Source*** возвращает именно итератор. Затем этот ***it*** можно использовать для построчной обработки текста в цикле:
for (i <- it) print(i) → будет выведен весь текст
Вместо вывода можно запрограммировать обработку текста, как коллекции. Можно также поместить строки файла в массив, в список, или в буфер с помощью методов ***toArray, toList, toBuffer:***
val m = it.toArray
(Надо также иметь ввиду, что после одного использования итератор обнуляется и если он потребовался снова, надо всё

восстановить сначала)

При объявлении переменной для чтения файла рекомендуется, кроме того, вместо (***val x = …)*** применять (***lazy val x = …)***, тогда файл будет открыт только при обращении к переменной ***x***.

Для чтения отдельного символа из файла можно использовать непосредственно ***x***:

for (c <- x) print(c)

Будут напечатаны все символы. Создадим теперь список с элементами — символами файла:

var s = List[Char]() → пустой список, тип ***Char***

for (c <- x) s = c :: s → список символов

Наконец, можно просто прочитать содержимое файла в строку с помощью метода *mkString:*

val s = x.mkString

Теперь эту строку можно обрабатывать, как нам будет нужно.

Закончив использовать объект ***x***, следует вызвать метод ***close*** для закрытия файла:

x.close()

2) Запись в файл

Scala не имеет встроенной поддержки записи в файлы, поэтому приходится использовать средства java. В частности можно воспользоваться классом ***java.io.PrintWriter***. Покажем на примере:

```
import java.io.PrintWriter
val x = new PrintWriter("rab.scala")
for (i <- 1 to 5) x.println(i)
x.close()
```

Мы записали в файл *rab.scala* целые числа от 1 до 5. Если файл *rab.scala* не существовал, он будет создан в текущей директории. Если такой файл уже был, старое содержание будет стёрто. Можно организовать форматированный вывод в файл с помощью оператора ***printf.*** Разумеется, можно использовать все средства *java* для работы с файлами.

Глава 8 Функциональное программирование

Основные понятия, на которых строится функциональное программирование это: чистые (без побочных эффектов) функции,

рекурсия, сопоставление с образцом, анонимные функции (лямбда-функции), каррированные функции, возможность использования функций в качестве параметров для других функций, коллекции и селективные операции над ними, развитая система типов. Всё это имеется в Scala так что его вполне можно считать функциональным языком программирования. Рассматривать функциональный стиль программирования подробно мы здесь не будем; ограничимся только некоторыми характерными примерами. Самый популярный пример — это, конечно, вычисление факториала:

```
def f(n:Int):Int = n match {
        case 0 => 1
        case _ => n * f(n-1)
}
print(f(5)) → 120
```

Здесь использована рекурсия и сопоставление с образцом *match-case*. Всё очень просто: если ***n=0***, то результат равен ***1***, иначе (при ***n>0)***, значение ***n*** умножается на факториал ***(n-1)***. Образуется цикл в котором выполняется цепочка умножений:

n*(n-1)*(n-2)* … *1

Поскольку результат выполнения функции повторно используется, как входной параметр (а для них всегда должен быть явно указан тип), то необходимо указывать тип возвращаемого функцией результата. Во втором ***case*** использован групповой символ ***(_)***, но тут можно поставить и знак ***n***.

Эта функция является «чистой», поскольку не имеет побочных эффектов: на входе целое число ***n***, а на выходе — факториал этого числа. Это что-то вроде чёрного ящика.

Самый известный функциональный язык — это, конечно, Haskell. На этом языке наша программа будет выглядеть так:

```
f :: Integer -> Integer
f 0 = 1
f n = n * f (n-1)
main = print(f 5) → 120
```

Это более лаконичная запись, чем на Scala; тем более, что первая строчка объявления типов необязательна. Однако, по сути программы на обоих языках практически идентичны. Но, посмотрим на другой пример: пусть требуется из двух списков

получить третий список, элементы которого будут равны сумме соответствующих элементов исходных списков. Заданные списки имеют разные размеры, лишние элементы более длинного списка надо отбросить. Программа может выглядеть примерно так:

```
def mn(ss1:List[Int], ss2:List[Int])=if (ss1.length < ss2.length)
ss1.length else ss2.length
def an(sp:List[Int], n:Int):Int=if (n==0) sp.head else an(sp.tail,n-1)
def sm(s1:List[Int],s2:List[Int],m:Int):List[Int]=
if (m==0) List(an(s1,m)+an(s2,m)) else
 sm(s1,s2,m-1)++List(an(s1,m) + an(s2,m))
def rab(r1:List[Int],r2:List[Int])=sm(r1,r2,mn(r1,r2)-1)
val a = List(1,2,3)
val b = List(1,2,3,4,5)
print(rab(a,b)) → List(2, 4, 6)
```

Три промежуточные функции ***mn***, ***an*** и ***sm*** введены только для разбивки и так довольно громоздкого текста. В финальной функции ***rab*** и в промежуточной ***an*** использована рекурсия. Конечно, не в функциональном стиле этот пример на Scala будет выглядеть намного лаконичнее и "читабельнее". Если механически вычеркнуть слова ***def***, ***List*** и все объявления типов, то эта программа может быть выполнена и на Haskell, можете проверить.

Теперь рассмотрим пример на вычисление суммы элементов списка:

```
def f(a:List[Int]):Int = a match {
      case Nil => 0
      case s :: xs => s + f(xs)
}
print(f(List(1,2,3))) → 6
```

Здесь использованы сопоставление с образцом, рекурсия и уже знакомое нам выделение головы и хвоста списка в операторе *case* с помощью записи вида ***s :: xs***. Параметр ***Nil*** соответствует пустому списку. Если список пуст результат равен нулю. Если нет, то первый элемент (голова) добавляется к сумме элементов хвоста и образуется цикл с вычислениями по цепочке:

3 + 2 + 1 + 0

Опять покажем, как это будет на Haskell:

```
f [] = 0
f (x:xs) = x + f xs
```

m = [1,2,3]
main = print(f m) → 6
Практически полное совпадение, за исключением второстепенных деталей.

Анонимные функции мы уже неоднократно применяли ранее.

Глава 9 Система типов

Язык Scala имеет весьма сложную систему типов. Сразу понять все её тонкости трудно, да и не нужно до тех пор, пока дело не дойдёт до разработки крупных и сложных проектов. В рассмотренных ранее примерах уже встречались разные варианты использования типов в простых случаях. Ограничимся здесь только некоторыми общими сведениями о системе типов Scala.

Классы, трейты, методы и функции могут иметь параметры типов, которые помещаются после имени и заключаются в квадратные скобки. Такие объекты называют обобщёнными. Приведём пример обобщённого класса:
class Pair[T,S](x:T, y:S)
Этот класс имеет два параметра типов ***T*** и ***S*** и принимает два аргумента этих типов. Можно создать экземпляр класса, передав ему аргументы фактических типов, например:
val p = new Pair(25, "Marta") → p: Pair[Int,String]
Таким образом, Scala выводит фактические типы на этапе конструирования объекта. Можно задать требуемые фактические типы явно:
val p = new Pair[Double, Any](25, "Marta") → p: Pair[Double,Any]
Создадим какой-нибудь метод класса:
class Pair[T,S](x:T, y:S) { def f = List(x, y) }
val p = new Pair(25, "Marta")
print(p.f) → List(25, Marta)
Всё работает, поскольку наш метод может создать список при любых фактических типах переменных ***x*** и ***y***. Но если мы попробуем объявить, например, такой метод:
class Pair[T,S](x:T, y:S) { def f = x * y }
то получим ошибку. Компилятор не знает, какие фактические типы получат ***x*** и ***y*** и будет ли допустима при этих типах операция

умножения. Поскольку любой тип может быть преобразован в ***String*** который в свою очередь может быть преобразован в какой-то другой тип, то можно разрешить проблему так:

```
class Pair[T,S](x:T, y:S)  {
        val x1 = x.toString.toInt
        val y1 = y.toString.toInt
        def f = x1 * y1
}
val p = new Pair(3, 7)
print(p.f) → 21
```

Это опять работает. Правда, тут возникает вопрос, имеет ли практический смысл такое программирование. К счастью есть возможность применять так называемые ограничители типов и на нашем примере это будет выглядеть так:

```
class Pair[T <: Int,S <: Int](x:T, y:S)  { def f = x * y }
```

Запись ***T <: Int*** надо понимать так, что фактический параметр будет подтипом типа ***Int***. И тогда можно создать экземпляр класса в таком виде:

```
val p = new Pair[Int, Int](2,3)
 p.f → 6
```

Надо только не забывать указывать фактические типы после имени в квадратных скобках.

Функции и методы тоже могут быть обобщёнными, то-есть принимать параметры типов:

```
def f[T <: Double](x:T, y:T) = x * y
f[Double](2.1, 3.0) → 6.3
```

Здесь мы тоже применили ограничитель типа.

Приведём ещё один пример. Пусть нам надо найти меньшее значение из двух:

```
class A[T <: Comparable[T]](x:T, y:T) { def f = if (x.compareTo(y)<0)
x else y }
```

Метод ***x.compareTo(y)*** возвращает ***-1*** если ***x<y, 0*** если ***x=y*** и ***+1*** если ***x>y***. Пришлось ввести ограничитель типа ***Comparable*** без которого компилятор не знал бы, применим ли метод ***compareTo*** при фактическом типе. Наш метод ***f*** не будет работать при типах ***Int*** и ***Double***, так как они не являются подтипами ***Comparable*** . Подобных классов и типов в Scala огромное количество и разобраться в этом хозяйстве можно только при тщательном

изучении документации.

val p = new A[String]("bbb", "aaa")

p.f → aaa

String является подтипом ***Comparable***.

Ограничитель ***(<:)*** является верхним, а есть ещё и нижний ограничитель ***(>:)***, а также и другие ограничители. Предоставляю читателям испытать «удовольствие», разбираясь самостоятельно в этом лабиринте.

Упомяну ещё о том, что есть три разновидности обобщённых типов: инвариант (обычный, тот который и применялся в примерах выше), ковариант (обозначается ***[+T]***) и контравариант (обозначается ***[-T]***). Не имею понятия, почему тут принята терминология из векторной алгебры. Приводимые в литературе примеры на эту тему столь примитивны, что невозможно оценить полезность этих понятий и у меня нет желания разбирать их здесь. Ну, а заниматься какой-то более сложной программой пока не имеет смысла.

Ну, а посему позвольте на этом закончить нашу повесть. Имейте только ввиду, что есть ещё ряд больших тем в языке Scala, которых мы здесь совсем даже и не касались.