Программирование на Groovy
для любопытных

Содержание

…..Краткое введение

Groovy объектно-ориентированный язык программирования, реализованный на виртуальной машине Java (JVM). Язык имеет динамическую типизацию.

Имя файла с программным кодом должно иметь расширение (***.groovy***). Этот файл можно скомпилировать в программный код Java и затем выполнить на JVM. Имеется собственный простой редактор с подсветкой синтаксиса, позволяющий выполнять, редактировать и тестировать программы. Есть также интерактивный режим (Repl), позволяющий вводить строки программы и немедленно выполнять их с получением результата.

Для установки Groovy требуется только скачать дистрибутив и добавить в переменную PATH адрес директории ***bin***. После этого будут доступны в командной строке следующие команды:
- ***groovy name.groovy*** – выполнение программы из файла ***name.groovy***
- ***groovysh*** – вызов режима Repl
- ***groovyconsole*** – вызов редактора groovy
- ***groovyc name.groovy*** – компиляция программы из файла ***name.groovy***, в результате чего будут созданы несколько файлов, как

это обычно бывает в сумасшедшем Java, в частности, файл ***name.class***, который можно выполнить средствами Java. В последнем случае в исходном файле должны быть выполнены некоторые требования Java, в частности, должен присутствовать пакет ***main***.

Прежде чем приступить к изложению материала, сделаю пару замечаний:

Считаю, что нет смысла начинать с рассмотрения базовых элементов языка, поскольку они традиционны и я предполагаю, что читатель не начинающий программист - при первом знакомстве с программированием для начала надо почитать совсем другие книги. Например, я не собираюсь объяснять цикл ***for***, логические операции, классы и объекты и тому подобное, если только в контексте их применения не встречаются некоторые особенности, присущие не всем современным языкам. Отмечу только, что арифметическая операция деления имеет следующие разновидности:
println 5.intdiv(3) → ***1*** (целая часть результата деления)
println 5 % 3 → ***2*** (остаток от деления целых чисел)
println 5 / 3 → ***1.6666***… (обычное деление)

Вопреки повсеместным рекомендациям насчёт того, что идентификаторы переменных и функций всегда должны отражать их смысловое содержание, убеждён, что надо стремиться к максимальной краткости кода. Если объект существует короткое время, иногда лишь на нескольких строчках кода, его название должно быть как можно короче. Всюду, где это возможно, мои идентификаторы будут представлены одной — двумя буквами, так функции я чаще всего буду обозначать буквой ***f*** или ***g,*** а переменные — ***x, y, z*** и так далее.

Всё изложение материала будет базироваться на примерах без многословных рассуждений о несущественных деталях и о всякой замысловатой терминологии — совсем не в терминах дело. Исходя из тех же соображений краткости, буду стараться применять самые короткие термины, часто в английском варианте. Вместе с тем, каждый пример будем обсуждать максимально подробно, с тем, чтобы всё было понятно и не слишком опытному программисту.

Итак, следуя вышесказанному, начнём изложение сразу с вложенных функций, играющих в Groovy важную роль.

…..1. Функции, передаваемые в качестве аргументов другим функциям (closures)

…..1.1 Передача closure функциям

Слово *closure* переводится, как замыкание. В Groovy оно означает не именованную (анонимную) функцию или блок кода в фигурных скобках, предназначенных для передачи другой функции на правах аргумента. Впрочем, при желании блоку можно присваивать и название, для возможности его повторного использования и для улучшения читабельности. Рассмотрим этот приём на простейшем примере. Пусть требуется вывести на печать числа из заданного диапазона чисел через некоторый интервал. Программа с применением closure может выглядеть так:

```
def f(n, d, g)  {
   for (i=0; i<=n; i+=d) { g(i) }
}
```

Значит, функции в Groovy объявляются со служебным словом ***def***, а список аргументов должен быть в круглых скобках. Вместо ***def*** может быть имя типа возвращаемого функцией результата. Фактически слово ***def*** является заместителем имени типа и указывает на то, что действительный тип результата будет выведен динамически. Если тип будет указан, как ***void***, функция должна возвращать ***null***. Наконец, слово ***def*** имеет синоним ***Object***.

Объявленная нами функция ***f*** принимает три аргумента: число ***n***, определяющее размер рассматриваемого диапазона ***0 .. n***, интервал чисел ***d***, и аргумент, обозначенный буквой ***g***. Рассматривая код тела функции ***f*** - оно ограничено фигурными скобками, обнаружим, что ***g*** – это функция, поскольку ей передаётся аргумент ***i***. Функция ***g*** использована в теле цикла ***for*** (оно также ограничивается фигурными скобками) и, значит, выполняется многократно, каждый раз принимая в качестве аргумента текущее значение переменной цикла ***i***. Таким образом, наша функция ***f*** принимает другую функцию ***g***, как аргумент, а то, что передаётся на место ***g*** и называется closure. Ясно, что конкретный вид closure должен быть определён при вызове функции ***f***. Пусть нам требуется только вывести искомые числа на печать. Тогда вызов функции ***f*** может иметь такой вид:

```
f(10, 2, {println it}) → 0 2 4 6 8 10
```

Здесь и всюду далее будем использовать стрелку (→) вместо слов получим, будет равно и тому подобное.

На самом деле числа будут расположены столбиком, поскольку оператор вывода ***println*** в конце выводимого текста добавляет перевод строки.

Можем видеть, что при передаче closure в качестве параметра, весь его текст должен быть заключён в фигурные скобки (на самом деле это может быть блок произвольного размера). Использованная тут переменная с идентификатором ***it***, принята по умолчанию и в таком варианте closure идентификатор нельзя изменить. Когда функция ***g*** принимает значение closure, переменной ***it*** передаётся значение переменной цикла ***i.***

Кстати, объявленная таким образом функция ***g*** может рассматриваться, как переменная, имеющая специальный тип ***Closure***. Этот тип выводится компилятором исходя из контекста, но его можно задать и принудительно, объявив функцию ***f*** так:

def f(n, d, Closure g) {…}

Польза от этого видимо только в улучшении читабельности кода.

В примере использованы целые числа (тип ***Integer***), но ничто нам не мешает рассматривать и числа с плавающей точкой (тип ***Double***):

f(3, 0.5, {printf " %5.3s", it }) → 0 0.5 1.0 1.5 2.0 2.5 3.0

Только шаг ***d*** мы ввели, как число с плавающей точкой, а остальные оставили в форме целых чисел, но этого компилятору оказалось достаточно, чтобы для вычислений принять тип ***Double***. В этом заключается достоинство динамической типизации, при которой тип переменных определяется (выводится, derive) на этапе runtime исходя из контекста. Впрочем, Groovy позволяет и принудительное задание типа, что бывает необходимо во избежание двусмысленности. Позже мы ещё используем этот приём. Чтобы расположить числа в строку и разделить их промежутками, применили форматированный вывод — оператор ***printf*** с форматом ***" %5.3s"***.

Операторы ***print*** и ***printf*** не содержат ничего нового. А вот ввод с клавиатуры в Groovy использует средства Java и потому выглядит довольно неуклюже. Есть несколько способов ввода с клавиатуры, наиболее простой из них можно реализовать приблизительно так:

```
s = System.console().readLine "введите строку ="
print s.toDouble()**2
```

Не будем пока обсуждать встроенные методы, использованные здесь. Отметим только, что так можно ввести с клавиатуры и инициировать какую-нибудь переменную значением типа ***String.*** Текст ***введите строку =*** будет выведен в командной строке, где и требуется напечатать вводимый текст. Для получения, например, числа типа ***Double***, (в примере мы его возводим в квадрат) требуется применить метод ***toDouble()*** для перевода из ***String*** в ***Double***.

Однако, вернёмся к closure. Имеется возможность не использовать встроенную переменную ***it***, а ввести собственный идентификатор, что вообще необходимо, если параметров больше одного. Для этой цели применяется такой синтаксис:

f(3, 0.5, {t -> println t})

Теперь введённая нами переменная ***t*** будет выполнять точно ту же самую роль, что и стандартная переменная ***it*** в предыдущем случае. Если вызываемая функция располагается на последнем месте в списке аргументов, как у нашей функции ***f***, допустимо вызов записывать и в такой форме:

f(3, 0.5) {t -> printf " %5.3s", t }

то-есть, выносить closure из круглых скобок.

Обычно функции, принимающие другие функции в качестве аргументов, называют функциями высшего порядка (higher-order function). Будем также для краткости называть их головными функциями.

Достоинство этой техники заключается в том, что функции высшего порядка можно передавать различные closure, изменяя, таким образом, её функциональность. Например, мы можем использовать функцию ***f*** для вычисления суммы чисел из заданного диапазона, для чего достаточно лишь изменить closure. Хотя сама функция ***f*** остаётся неизменной, приведём для наглядности весь текст:

```
def f(n, d, g)  {
   for (i=0; i<=n; i+=d) {g(i)}
}
s = 0
f(10, 2, {t -> s+= t})
println "summa = ${s}"  → summa = 30
```

В этом примере мы вычислили сумму всех чётных чисел из диапазона ***0..10,*** накопив эту сумму в переменной ***s***.

В операторе вывода ***println*** использована так называемая интерполяция, позволяющая вставлять в строку результат какого-нибудь вычисления. Текст ***${s}*** означает, что требуется вычислить выражение внутри фигурных скобок (в примере только извлечь значение переменной ***s***), преобразовать результат к типу ***String*** и вставить в строку, ограниченную кавычками.

Рассмотрим теперь пример, в котором closure принимает больше одного параметра и содержит условное выражение:

```
def f(x, y, g) {
   if ( g(x,y) ) println "x меньше y"
      else println "x больше y"
}
f(2, 5) {r, t -> (r < t)}  →  x меньше y
```

Здесь closure ***{r, t -> (r < t)}*** возвращает значение ***true*** или ***false***, которое подставляется на место функции ***g(x,y)***. Кстати, можно не вводить новые идентификаторы ***r*** и ***t***, а использовать те же ***x*** и ***y***, записав вызов функции ***f*** в таком виде:

```
f(2, 5) {x, y -> (x < y)}
```

При этом следует помнить, что это разные переменные. Конфликта имён нет, потому-что в пространстве closure переменные ***x*** и ***y***, объявленные в головной функции, не видны.

Отметим попутно, что цикл ***for*** можно организовать с использованием диапазона в такой форме:

```
def f(n, g)  {
   for (i in 0..n ) {g(i)}
}
s = 0
f(10, {t -> s+= t})
println "summa = ${s}"  → summa = 55
```

Вычислили сумму чисел из диапазона ***0..10***.

Здесь уместно будет сделать пару замечаний по поводу объявления переменных в Groovy, поскольку переменные участвуют практически во всех примерах далее.

При объявлении новой переменной можно, как и для функций, применять ключевое слово ***def,*** или не применять. В чём тут разница, посмотрим на примере:

```
a = 32
def c = 'there', d = 'yonder'
```

```
def f(){
   println a → 32
   def c = 'here'
   //println d → здесь будет ошибка, если строку раскомментировать
   c
}
println f() → here – функция f возвращает значение локальной c.
print c →  there  - выводится значение глобальной c.
```

Переменная ***a*** объявлена без слова ***def*** и в таком случае она видна в теле дальше расположенных функций; в примере значение ***a*** выводится на печать в теле функции ***f***. Переменные ***c*** и ***d*** объявлены со словом ***def*** (можно при одном ***def*** объявлять сразу несколько переменных через запятую). И в этом варианте переменные ***c*** и ***d*** не видны в теле функции — попытка распечатать ***d*** приводит к ошибке. В теле функции есть возможность объявить со словом ***def*** новую переменную с тем же идентификатором ***c***. Эта новая переменная ***c*** локальна, то-есть видна только в теле функции ***f***. За пределами функции доступно старое значение ***c***, равное ***there***.

При объявлении функции ключевое слово ***def*** необходимо, или его можно заменить указателем типа возвращаемого функцией результата:

```
Double f(x)  {
   Math.tan(x)
}
print f(0.3) → 0.30933624960962325
```

Вариант с ***def*** более универсальный, так как при этом не ограничивается тип возвращаемого функцией результата.

Переменную можно инициировать функцией и тогда эта переменная сама становится функцией. При такой операции применяется специальный синтаксис ***this.&***. Посмотрим на примере:

```
def f(){ 77 }; println f() → 77
g = this.&f; println g() → 77 (можно применить def)
h = g; println h() → 77  (можно применить def)
f = 'something else';  print f() → 77
print f → something else
```

При повторном присваивании ***this.&*** не требуется (***h = g).*** Можно использовать для переменной тот же идентификатор, что и для функции (у нас ***f***), конфликта имён в таком случае не возникает.

…..1.2 Применение closures в классах

Во-первых отметим, что методам класса closure можно передать точно также, как и любой функции:

```
class A  {
   def f(x, g) {g(x)}
}
e = new A()
res = e.f(3, {t -> t **3})
print res  → 27
```

В общем случае, при создании экземпляра класса ему могут передаваться параметры. В нашем примере параметров нет, но пустые скобки необходимы: ***e = new A()***

Метод ***f*** принимает переменную ***x*** и closure ***g***. Фактически они имеют равные права и за пределами метода ***f***, конечно, не видны. Но часто требуется использовать переменную, доступную всем методам класса, её обычно называют переменной экземпляра класса. Точно также, бывает желательно передать классу closure, так, чтобы он был доступен всем методам класса, можно сказать по аналогии, что тогда это будет closure экземпляра класса. В этом случае применяется особый синтаксис, рассмотрим его на примере:

```
class Rab {
   def x, g
   Rab (a, r) {x = a; g = r}
   def f() { g(x) }
   def fi(y) { g(y) }
   }
def e1 = new Rab(0, {println "Hello e1"})
e1.f()  →  Hello e1
def r = { t -> println Math.sin(t) }
def e2 = new Rab(1.5, r)
e2.f()  → 0.9974   - это sin(1.5)
e2.fi(2.0)  → 0.9092   - это sin(2.0)
```

В классе ***Rab*** создана переменная экземпляра ***x*** и closure ***g***, доступные для всех методов класса. В этом случае их требуется предварительно объявить (без инициации) с помощью директивы ***def***:

```
def x, g
```

Строка ***Rab (a,r) {x = a; g = r}*** является конструктором класса, который на этапе создания экземпляра принимает параметры класса (в данном случае ***a*** и ***r***) и инициирует ими переменную экземпляра ***x*** и closure ***g***. В экземпляре класса ***e1*** переменная ***x*** не использована, но поскольку передать параметр необходимо, мы приравняли его к нулю. В классе имеется два метода: метод ***f*** не принимает параметров, а использует переменную экземпляра ***x,*** и метод ***fi*** переменную ***x*** не использует, но имеет аргумент ***y***, который инициируется при вызове метода.

При создании экземпляра класса ***e2*** мы предварительно присвоили closure имя: ***def r = { t -> println Math.sin(t) }*** (использовали тот же идентификатор ***r***, но это не имеет никакого значения, идентификатор может быть любым)**.** Такой приём полезен, во-первых, потому, что улучшает читабельность, если closure имеет длинный текст, а во-вторых - closure с именем можно использовать многократно.

…..1.3 Closures с параметрами

Closures сами могут иметь и принимать параметры при вызове. Задать и инициировать параметры closure можно в теле головной функции, а действия над этими параметрами можно определить при вызове функции. Посмотрим на такой пример:

```
def f(g)  {
   g(1.5, 2.0)
   }
res = f() {x, y ->  x**2 + Math.cos(y)}
println "1.5**2 + cos(2.0) = ${res}"  → 1.5**2 + cos(2.0) = 1.8338
```

Функция ***f*** имеет в списке аргументов только closure ***g***. В теле функции заданы и инициированы числами типа ***Double*** два параметра closure. В таком варианте при вызове ***f*** пустые скобки ***()*** указывают, что аргументов нет, кроме closure ***g,*** код которого далее добавляется в фигурных скобках. При этом переменным ***x*** и ***y*** передаются значения параметров, установленные в теле вызываемой функции. Можно применить и такой, более очевидный вызов:

```
res = f ( {x, y ->  x**2 + Math.cos(y)} )
```

Вот ещё один пример:

```
def f(g) {
   g (new Date("09/20/2018"), "Ваш день рождения")
```

}
f({ date, d -> println "Напоминаем, что ${date} будет '${d}'" }) → Напоминаем, что Thu Sep 20 00:00:00 MSK 2018 будет 'Ваш день рождения'

Здесь первому параметру closure ***g*** присвоено значение экземпляра стандартного класса ***Date***, задающего форму для печати заданной даты. Как уже упоминалось, обычно можно не указывать явно тип параметров, компилятор выводит их из контекста, но можно и указать. Так, в этом примере переменная ***date*** имеет тип ***Date*** (экземпляры классов всегда имеют тип по названию класса), а переменная ***d*** тип ***String***:

f({Date date, String d -> println "Напоминаем, что ${date} будет '${d}'" })

Позже мы буде рассматривать метапрограммирование, в котором closures будут использоваться для перегрузки или замены методов и тогда принудительное указание типов часто будет необходимым.

…..1.4 Использование closures для автоматического закрытия файлов

Groovy имеет встроенную систему очистки мусора — на основе соответствующих средств Java. Но иногда бывает необходимо предусматривать собственные действия, особенно при использовании ресурсоёмких (resourse-intensive) классов. Примером таких действий могут быть методы, наподобие ***close()*** или ***destroy()***. Например, пусть требуется запрограммировать вывод в файл ***output.tx***t текста ***'Hello'***. В Groovy для этой цели используется стандартный класс ***FileWriter***:

w = new FileWriter('output.txt')
w.write('Hello World')

Создавая экземпляр ***w*** класса ***FileWriter***, мы передаём ему название файла (здесь может быть относительный, как у нас, или полный путь). При создании экземпляра файл открывается для записи, или создаётся заново, если до этого ещё не существовал. Затем метод класса ***write*** выводит свой аргумент в файл. Однако, вывода не произойдёт, а файл будет считаться занятым, до тех пор пока мы его не закроем:

w.close()

Можно обеспечить автоматическое закрытие файла, чтобы потом не заботится об этом. Для этой цели применяется метод ***withWriter***

класса ***FileWriter***. Этот метод принимает в качестве параметра closure, осуществляющий вывод, и автоматически закрывает файл:

```
w = new FileWriter('output.txt')
w.withWriter {t -> t.write('Hello')}
```

При вызове метода ***withWriter*** параметр closure ***t*** принимает информацию о расположении и имени файла.

Кстати, можно не вводить имя экземпляра класса ***w***, а применить такую форму, если нравится:

```
new FileWriter('output.txt').withWriter {t -> t.write('Hello')}
```

…..1.5 Использование closures в качестве сопрограмм (coroutine)

Использование сопрограмм позволяет организовать некоторое подобие многопоточности. Головная программа приостанавливает работу в некоторой точке, передаёт управление сопрограмме и после выполнения сопрограммой нужных действий продолжает работу с места останова. При этом между головной программой и сопрограммой происходит обмен данными. В Groovy роль сопрограммы может выполнять closure, например:

```
def f(n, g) {
   1.upto(n) {
      print "it = ${it}"
      g(it)
   }
}
s = 0
f(4) { s += it; println ", сумма s = ${s}" } →
it = 1, сумма s = 1
it = 2, сумма s = 3
it = 3, сумма s = 6
it = 4, сумма s = 10
```

Здесь ***1.upto(n)*** задаёт цикл, в котором в качестве переменной цикла использована та же самая переменная ***it***, которая задействована в closures, а ***1*** и ***n*** определяют диапазон её изменения. Closure, передаваемый функции ***f*** в качестве аргумента, выполняет роль сопрограммы. Головная программа изменяет переменную ***it***, выводит её на печать и обращается к closure, где вычисляется и выводится на печать накапливающаяся сумма, после чего головная программа

продолжает работу. Как обычно, в closure можно вместо ***it*** использовать и переменную с собственным идентификатором:
f(4) {t -> s += t; println ", сумма s = ${s}" }

…..1.6 Сокращение числа передаваемых параметров (curried closure)

Closure могут принимать несколько параметров, при этом некоторые из них при повторных вызовах не изменяются. Вызывая closure повторно мы должны каждый раз повторять значения этих параметров, что делает код громоздким, особенно если передаются большие строки текста. В Groovy имеется встроенный метод ***curry***, позволяющий исключить передачу одних и тех же данных, а сам этот приём называется каррированием (термин заимствован из языка Haskell, но там он имеет, в общем-то, другой смысл). При каррировании используется особый синтаксис; продемонстрируем на примере:

```
def f(x, g)  {
   r = g.curry(x)
   r Math.sin(x)
   r Math.cos(x)
}
f(1.5) {s, t -> println "x = ${s}, res = ${t}"} →
x = 1.5, res = 0.9974     - sin(1.5)
x = 1.5, res = 0.0707     - cos(1.5)
```

Эти три строки:

```
   r = g.curry(x)
   r Math.sin(x)
   r Math.cos(x)
```

можно трактовать так: первая строка указывает, что closure ***g*** будет вызываться повторно, при этом при каждом вызове переменная ***x*** будет иметь одно и то же значение, указанное при вызове функции ***f (1.5)***. Дальше перечисляются варианты кода, которые будет иметь closure ***g*** при повторных вызовах. При этом записывается введённый в первой строке идентификатор (у нас ***r***) и рядом указывается нужный код. В самом closure вводится два параметра (***s*** и ***t***), первому передается значение каррированной переменной (***x***), а второму — последовательно, указанный в перечислении вариантов код (у нас ***Math.sin(x)*** и **Math.cos(x))**.

Можно, конечно, поступать и так:

```
def fi(x) { a = x**2; b = a + x }
def xi(x) { if (x>0) x else -x }
def f(x, g)  {
   r = g.curry(x)
   r xi(x)
   r fi(x)
}
f(-3.5) {s, t -> println "x = ${s}, res = ${t}"} →
x = -3.5, res = 3.5
x = -3.5, res = 8.75
```

Конечно, в наших примерах мы получили не слишком много пользы от этого приёма, поскольку ***x*** принимает значение числа, было бы иначе, если бы аргументу **x** передавался большой текст, или неизменных параметров было бы много.

Метод ***curry*** позволяет каррировать сразу несколько параметров. Кроме него имеются ещё методы ***rcurry*** и ***ncurry***, обеспечивающие дополнительные возможности.

…..1.7 Применение closure в динамике

Есть возможность проверять, передано головной функции closure, или нет, например:

```
def f(x, g)  {
   if (g) {g(x)} else {x**2}
}
println "res = ${f(0.5,0)}"  → res = 0.25
println "res = ${f(0.5, {Math.cos(it)})}"  → res = 0.8775
```

Значит, если рассматривать closure, как условное выражение, то получим ***true*** (в Groovy всё будет давать ***true***, кроме значений: ***false, 0*** и ***null***). Запрограммированная с такой проверкой функция ***f*** может вызываться, как с closure, так и без него. Поскольку в списке аргументов на месте closure должно быть что-то указано, можно тут вставлять ***false, 0*** или ***null***.

Имеется возможность динамически устанавливать количество и тип параметров, принимаемых closure. Рассмотрим такой пример: продаётся некий товар в количестве ***n*** штук при цене ***c*** за единицу. Дополнительно может быть (или не быть) наценка с коэффициентом

k. Требуется написать программу, которая могла бы вычислять общую стоимость товара для обоих вариантов.

```
def f(n, c, k, g)  {
   if (g.maximumNumberOfParameters == 3)
   {r = g(n, c, k)} else {r = g(n, c)}
   println "Стоимость покупки: ${r}"
}
```

f(10, 32.7, 0.08) {n, c, k -> n*c + n*c*k} → Стоимость покупки: 353.160

f(10, 32.7, 0) {n, c -> n*c} →Стоимость покупки: 327.0

Встроенный метод ***maximumNumberOfParameters*** (только сумасшедший мог придумать такой идентификатор) позволяет определить количество аргументов у любой функции:

```
def fi(x, y) {}
fi. maximumNumberOfParameters → 2
```

В нашем примере функции ***f*** можно передавать closure с двумя, или с тремя параметрами.

Есть также возможность определять типы параметров closure, например:

```
def f(x,y, g)  {
   print "Всего ${g.maximumNumberOfParameters} параметр(ов) типа:"
   for(p in g.parameterTypes) {println p.name}
}
```

f(1.3,9) {Double t, Integer s -> t + s} → Всего 2 параметр(ов) типа:java.lang.Double, java.lang.Integer

Встроенный метод ***parameterTypes*** определяет тип параметра. Для определённости типы параметров заданы в closure. Цикл ***for*** просматривает все параметры, а встроенный метод ***name*** формирует имя типа.

…..1.8 Closure, как одиночный (single) метод

Можно просто инициировать переменную каким-нибудь closure и тогда эта переменная может использоваться, как метод:

```
def q = { t -> println("hello ${t}") }
q("world!") → hello world!
```

При этом параметр closure становится аргументом метода. Ясно, что как всегда можно воспользоваться и встроенной ***it***:

def q = { println("hello ${it}") }
q("world!") → hello world!

Параметров может быть больше одного:

q = { s, t -> println(s + t) }
q("hello ", "world!") → hello world!

Значит, слово ***def*** можно и опускать.

Имеется также тип ***Closure***, так что можно поступать и так:

Closure q = { t -> println("hello ${t}") }
q("world!") → hello world!

Closure можно определять предварительно, присваивая ему идентификатор:

g = { i -> print i}
[1, 2, 3].each g → 123

Closure могут быть вложенными:

def f = { a, c -> c(c(a)) }
r = f(5, {it * 3})
print r → 45

Здесь closure ***{it * 3}*** подставляется на место ***c(a)***, а затем на место ***c(c(a))***. В результате число ***5*** умножается на ***3*** дважды.

При передаче closure функции, в качестве параметра, вложенность программируется более простым и очевидным способом:

```
def f (x, g, q)  {
   g(q(x))
}
r = f(5,{it *3}, {it +2})
print r → 21
```

То же самое можно запрограммировать и так:

def f = { a, g, q -> g(q(a)) }
print f(5){it*3}{it*4} → 60

Приведём ещё один пример вложенного closure. Пусть требуется создать функции, выделяющие из списка чётные или нечётные числа с помощью вложенных closure и каррирования:

```
def g = { r, s -> s.findAll(r) }
def even = { it % 2 == 0 }
def odd = { !even(it) }
def e = g.curry(even)
def o = g.curry(odd)
println e(0..8) → [0, 2, 4, 6, 8]
```

println o(0..8) → [1, 3, 5, 7]

Closure ***g*** принимает два параметра: условное выражение ***r*** и список ***s***. Условные выражения, позволяющие отфильтровать чётные и нечётные числа, представлены двумя closure ***even*** и ***odd*** соответственно. Функции ***e*** (выделяет чётные числа) и ***o*** (выделяет нечётные числа) получены с помощью каррирования closure ***even*** и ***odd.*** Следует иметь ввиду, что при каррировании функции значение по умолчанию всегда задаётся первому параметру (в примере параметру ***r***, а не ***s***). Встроенный метод ***findAll*** является итератором, позволяющим отфильтровывать из списка элементы по заданному условию. Подробнее итераторы рассмотрим далее.

…..1.9 Рекурсия и closure

Clojure, как и все современные языки, позволяет рекурсивный вызов функций. Классический пример на эту тему — вычисление факториала для заданного числа ***n***:

```
def fact(BigInteger n) {
   if (n == 1) 1 else n * fact(n - 1)
}
try {
   println "factorial of 5 is ${fact(5)}"
   fact(5000)
} catch(Throwable ex) {
   println "Ошибка: ${ex.class.name}"
}
factorial of 5 is 120
Ошибка: java.lang.StackOverflowError
```

В примере мы задали тип числа ***n – BigInteger*** (большие целые числа), поскольку факториал очень быстро растёт с увеличением ***n***. Пара операторов ***try – catch*** позволяет зафиксировать ошибку; её характеристики формируются функцией ***Throwable***, а методы ***class*** и ***name*** определяют тип ошибки и название исключения. Для числа ***5000*** имеет место переполнение стека, что и зафиксировала наша программа. Переполнение стека происходит потому, что рекурсия в приведённом виде создаёт цепь вычислений:

n * (n – 1) * (n – 2) * … * 2 * 1

с запоминанием каждого сомножителя в памяти. Можно сомножители не сохранять в памяти, если использовать «аккумулятор» для накопления результата:

```
def fact(n, ac) {
        if (n == 1) ac else fact(n-1, n * ac)
}
println fact(5, 1) → 120
```

Здесь роль аккумулятора выполняет переменная ***ac***; переполнение стека не будет ни при каком ***n***. Приходится, однако, задавать начальное значение аккумулятора, в данном случае оно равно ***1***. Программу легко переделать, например, для вычисления суммы чисел ***0..n***, тогда начальное значение акуумулятора надо принять равным ***0***:

```
def sum(n, ac) {
        if (n == 0) ac else sum(n-1, n + ac)
}
println sum(5, 0) → 15
```

В Groovy есть встроенный метод ***trampoline***, который можно считать встроенным closure. Этот метод фактически сводит рекурсию к циклу ***for*** (разобраться в деталях можно в документации Groovy). Просто приведём пример применения метода ***trampoline***:

```
def fact(n) {
   g = {i, t -> i == 1 ? t : g.trampoline(i - 1, i * t)}
   println "factorial of ${n} is ${g(n, 1)}"
}
fact(5) → factorial of 5 is 120
```

В данном варианте также использован аккумулятор (***t***), начальное значение для которого принято по умолчанию равным ***1.***

Здесь использована ещё одна форма условного выражения, имеющего вид:

условие ? выражение 1 : выражение 2

Если условие даёт ***true***, выполняется выражение 1, иначе — выражение 2:

```
def f(x) {x>5 ? "${x} больше 5" : "${x} меньше 5" }
println f(8) → 8 больше 5
print f(2) → 2 меньше 5
```

Можно также для рекурсии использовать встроенную функцию ***call***, позволяющую вызывать closure для повторного выполнения:

```
def fact = { n -> n == 0 ? 1 : n * call(n - 1) }
```

print fact(5) → 120
Этот вариант позволяет использовать переменную ***fact***, как параметр. Например:
def f(g) {print g(4)}
f(fact) → 24

…..1.10 Expando

Имеется встроенный класс ***Expando***, который позволяет в свои экземпляры вставлять именованные closure и затем использовать их, как методы:

```
def p = new Expando()
p.g = { s.size() }
p.s = "Groovy"
println p.g() → 6
p.s = "Ruby"
println p.g() → 4
p.s = "PHP"
print p.g() → 3
```

Здесь в объекте ***p*** вставлен closure ***g***, имеющий, кроме того, введённый нами параметр ***s***. И этот параметр доступен далее, как переменная экземпляра класса ***Expando***. Надо, правда, иметь ввиду, что переменная ***s*** в сущности является глобальной и может иметь место конфликт имён, если во внешней области тоже будет объявлена переменная с этим же именем.

…..2. Работа с текстом

…..2.1 Литералы и выражения

В Groovy литералы записываются в одинарных кавычках:
s = 'Hello'
Посмотрим, какой тип у переменной s:
s.class → class java.lang.String
Значит ***s*** имеет тип ***String***.

Дальше я всегда буду опускать бессмысленные слова ***java.lang***, хотя они присутствуют везде и всюду, и писать просто
s.class → class String
или даже

s.class → String

В большинстве языков программирования знаки (***character***) записываются в одинарных кавычках: ***'a'.*** В Groovy знак можно создать с помощью директивы ***as:***

c = 'a' as char
c.class → Character

Или задать тип принудительно:

char s = '9'
s.class →Character

Тип ***Character*** может быть выведен компилятором из контекста, как и все другие типы.

Ну, а что же мы получим, применив двойные кавычки?

s = "Hello"
s.class → String

Значит, это тот же тип ***String***. Это можно проверить ещё и так:

'Hello' == "Hello" → true

Тем не менее, между строками в одинарных и двойных кавычках есть существенная разница. Во первых, литералы внутри себя могут содержать текст в двойных кавычках:

println 'He said, "That is Groovy"' → He said, "That is Groovy"

Во-вторых, интерполяция возможна только при двойных кавычках, а в одинарных — всё выводится, как есть:

x = 23
print "x = ${x}" → x = 23
print 'x = ${x}' → x = ${x}

Для многострочного текста применяются тройные кавычки. При этом три двойных кавычки позволяют интерполяцию, а три одинарных кавычки выводят текст, как есть:

```
def f(x, y)  {
   s = """Результат вычисления по формуле
(x + sin(y))/x**2 равен ${(x + Math.sin(y))/x**2}"""
   print s
}
f(3.2, 0.75) →
Результат вычисления по формуле
(x + sin(y))/x**2 равен 0.37906628515852875
```

…..2.2 Методы для преобразования строк

Тип ***String*** в Groovy не изменяем (immutable): однажды создав переменную типа ***String***, мы не можем её изменить, можно только создать новую переменную с этим же именем. Переменные типа ***String*** представляют разновидность коллекций - знаки из текста можно извлекать по индексу:

s = 'Привет, Мир!'
print s[8] → М

Но нет возможности изменить знак в тексте:

s[8] = 'p'
ERROR groovy.lang.MissingMethodException:
No signature of method: java.lang.String.putAt() is applicable for argument types: (java.lang.Integer, java.lang.String) values: [8, p]
Possible solutions: putAt(java.lang.String, java.lang.Object), getAt(int), getAt(int), getAt(java.lang.String), getAt(java.util.Collection), getAt(java.util.Collection)

Так выглядит текст исключения, генерируемого на подобную попытку (это следствие использования средств языка Java, отличающегося невероятной многословностью). Рекомендованный в этом ответе метод ***getAt*** позволяет извлекать знаки текста, в частности, знаки с индексами, заданными в виде элементов вектора:

s.getAt([1, 3, 9]) → рви

Для того, чтобы вставить в текст знак, являющийся по существу директивой и вызывающий некоторые действия (например ***$*** вызывает интерполяцию), используется обратный слеш:

c = 95
print "Цена товара \$${c}" → Цена товара $95

Если извлекается просто значение переменной (а не вычисляется некоторое выражение), фигурные скобки можно опускать:

print "Цена товара \$$c" → Цена товара $95

Вместо двойных кавычек можно применять прямой слеш (если это зачем-то нужно); потребуются ещё и круглые скобки:

value = 12
println (/He paid $$value for that/) → He paid $25 for that

Строки вместе с интерполяцией можно запоминать в переменных, чтобы потом использовать их, например, для вывода:

n = 19
s = "Ему уже ${n} лет"
print s → Ему уже 19 лет

Имеется много методов для работы со строковыми переменными. Метод ***toString()*** трансформирует переменную любого типа к типу ***String***:
x = 0.784
s = x.toString() → 0.784
s.class → String
Для принудительного изменения типа можно также применять оператор ***as***:
s = x as String
Метод ***replace*** позволяет заменять заданные знаки в тексте (напоминаю, что сама строковая переменная при этом не изменяется):
s = 'Hello World!'
s1 = s.replace('Hello', "Привет") → Привет World!
print s → Hello World!
print s1 → Привет World!

Встроенный класс ***StringBuilder*** позволяет создавать объекты (экземпляры), представляющие текст:
s = new StringBuilder('Hello World!')
s.class → StringBuilder
print s → Hello World!
Эти объекты изменяемы (mutable):
s.replace(0, 5, 'Привет') → Привет World!
print s → Привет World! **-** переменная ***s*** изменилась

Поскольку строки — коллекции, к ним можно применять итераторы. Итератор ***each*** принимает closure, которому передаёт последовательно все элементы коллекции:
s ***= 'Hello World!'***
s1 = ' '
s.each{t -> s1 = s1 + t + ','}
print s1 → H,e,l,l,o, ,W,o,r,l,d,!,
Следовательно, знак ***(+)*** в Groovy для строк – знак конкатенации. Можно, конечно, и так:
s.each{s1 += it + '/'}
print s1 → H,e,l,l,o, ,W,o,r,l,d,!,H/e/l/l/o/ /W/o/r/l/d/!/

Оператор умножения ***(*)*** используется для повторения текста:
x = 'Bob'
y = x * 3
print y → BobBobBob

Рассмотрим ещё один показательный пример:

```
h = [Apple : 663.01, Microsoft : 30.95, Google : 684.71]
c = {-> company }
p = {-> price }
q = "Today $c stock closed at $p"
h.each { key, value ->
   company = key
   price = value
   println q
}
```

Здесь ***h*** — хеш (ассоциированный список, отображение, словарь, мап в других языках), в котором ключи представляют названия компаний, а значения — акционерный капитал. Обратите внимание — при использовании литералов в качестве ключей кавычки можно опускать (хотя можно и применять). Подробнее хеш рассмотрим далее.

c и ***p*** — это closures без аргументов, а ***q*** — шаблон для вывода информации на печать. Итератор ***each*** передаёт ключи и значения хеша переменным ***key*** и ***value***, представляющим параметры closure итератора.

В результате выполнения данной программы будет выведено:

```
Today Apple stock closed at 663.01
Today Microsoft stock closed at 30.95
Today Google stock closed at 684.71
```

Знак ***(-)***, соответственно, позволяет удалять части текста из строк:

```
s = "Кто, волны, вас остановил"
s1 = s - ", волны,"
print s1 → Кто вас остановил
```

Приведём пример создания фрагмента XML документа, в котором перечисляются языки программирования и, соответственно, их авторы. Исходные данные представим в виде хеша, в котором ключи — названия языков, а значения — имена авторов:

```
s = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy']
content = ''
s.each {language, author ->
   fragment = """
<language name="${language}">
  <author>${author}</author>
```

```
</language>
"""
   content += fragment
}
xml = "<languages>${content}</languages>"
println xml
```

Получим такой прелестный результат:

```
<languages>
<language name="C++">
  <author>Stroustrup</author>
</language>

<language name="Java">
  <author>Gosling</author>
</language>

<language name="Lisp">
  <author>McCarthy</author>
</language>
</languages>
```

Есть возможность создавать ранги с элементами типа String:

```
s = 'воз'..'вор'
print s → [воз, вои, вой, вок, вол, вом, вон, воо, воп, вор]
```

И даже использовать их в цикле for:

```
for (i in 'воз'..'вор')  {
   print " ${i}"
}
→ воз вои вой вок вол вом вон воо воп вор
```

Метод ***split*** позволяет текст разбивать на элементы, помещая их в создаваемый список. Разбивку можно проводить по любому знаку, передаваемому методу ***split*** в качестве параметра:

```
s = "жили были дед да баба"
v = s.split(" ")
print v → [жили, были, дед, да, баба]
```

Здесь мы разбили текст по пробелам и теперь можем обрабатывать его, как список:

```
v[2] → дед
```

Можем выполнить разбивку по любому знаку, например по букве ***д***:

v = s.split("д")
print v → [жили были , е, , а баба]
При этом, как видим, сам знак (***д***), по которому выполнена разбивка, теряется из текста.

Имеется ещё целый ряд методов, позволяющих преобразовывать строки, например, такие, как ***plus() (+), multiply() (*), next() (++), replaceAll(), tokenize().***

…..2.3 Регулярные выражения (regex)

Для создания регулярного выражения в качестве образца при сравнении (pattern-matching) в Groovy применяется специальный оператор, обозначается (~). Например:
r = ~"hello"
Проверим тип переменной ***r***:
r.class → regex.Pattern
Таким образом, применение оператора (~) к строке создаёт экземпляр класса ***Pattern***. Теперь переменную ***r*** можно использовать в операциях сопоставления с образцом. Кроме двойных кавычек применимы одинарные кавычки и прямые слеши, например:
r = ~ "\\d*\\w*" - использованы двойные кавычки. При этом обратный слеш приходится дублировать, поскольку при двойных кавычках (впрочем и при одинарных тоже) одинарный обратный слеш воспринимается, как оператор управляющего символа.
r = ~/\d*\w*/ - полный аналог предыдущего варианта. Теперь обратные слеши не воспринимаются, как операторы управляющих символов и их дублирование не требуется. По-видимому, второй вариант можно считать более предпочтительным.

Для сопоставления с образцом в Groovy применяется знак ***(=~)***. Приведём примеры:
r = ~/(G|g)roovy/
Данное регулярное выражение позволяет проверить, содержится ли в заданном тексте слово Groovy или groovy. Проанализируем следующую фразу:
s = 'Groovy is Hip'
Для анализа применим условное выражение ***if***:
if (s =~ r) println "слово Groovy имеется" else println "слово Groovy отсутствует" → слово Groovy имеется

Проверим на варианте groovy:
s1 = 'groovy is Hip'
if (s1 =~ r) println "слово groovy имеется" else println "слово groovy отсутствует" → ***слово groovy имеется***
Кстати, нельзя поменять местами анализируемый текст и регулярное выражение - ***(r =~ s1)*** даст неверный результат.

Имеется два встроенных метода: ***replaceFirst*** и ***replaceAll***, которые позволяют заменять найденное при сравнении с образцом слово на заданное. Если в тексте обнаружено несколько искомых слов, то метод ***replaceFirst*** заменяет только первое из них, а метод ***replaceAll*** заменяет все:
s = 'Groovy is groovy, really groovy'
println s
res = (s =~ /groovy/).replaceAll('hip')
println res → Заменяются оба слова ***groovy*** на ***hip***:
Groovy is groovy, really groovy
Groovy is hip, really hip
res = (s =~ /groovy/).replaceFirst('hip')
println res → Теперь заменено только первое слово:
Groovy is hip, really groovy

Оператор сравнения (==~) применяется, если требуется проверить точное совпадение двух текстов, когда образец текста представлен, как регулярное выражение:
```
def f(s, reg) {
   if (s ==~ reg)
    { println("Фамилия записана верно.") }
  else { println("Ошибка, повторите.") }
}
reg1 = /Wisniewski/
f("Wisniewski", reg1) → Фамилия записана верно.
f("Wisnewski", reg1) → Ошибка, повторите.
```

В регулярное выражение в подобном варианте можно вставлять знак вопроса (?), который означает, что расположенный перед ним знак не обязателен, то-есть, его можно опустить, а сравнение всё-равно даст true.
```
reg1 = /Wisniew?ski/
f("Wisnieski", reg1)→ Фамилия записана верно.
```

Можно задавать разные допустимые варианты:

reg1 = /Wisn(ie|ei)w?ski/
f("Wisneiwski", reg1) → Фамилия записана верно.
Здесь мы допускаем перестановку букв ***i*** и ***e***.

Более подробным изучением регулярных выражений здесь мы заниматься не будем, поскольку это надолго отвлекло бы нас в сторону.

…..3. Работа с коллекциями

Мы уже применяли коллекции в примерах выше. Рассмотрим теперь их более подробно.

…..3.1 Списки

Списки в Groovy похожи на векторы или массивы во многих других языках. Для объявления списка достаточно его инициировать:
a = [1, 2, 3, 4, 5]
Посмотрим на тип переменной ***a***:
a.class → ArrayList
В названии типа присутствуют оба слова — Array и List, видимо список в Groovy обладает свойствами и списков и массивов.

Для извлечения элемента из списка принят обычный синтаксис:
a[3] → 4
Размер списка (количество элементов) позволяет найти метод ***size:***
a.size → 5
Groovy допускает применение отрицательных индексов, при которых отсчёт номера элемента будет выполняться с конца списка:
a[-1] → 5
a[-4] → 2
Можно извлечь сразу группу рядом расположенных элементов, используя ранг:
a[2..4] → [3, 4, 5]
Результат получаем в виде нового списка. Ранг можно применять и при отрицательных индексах:
a[-3..-1] → [3, 4, 5]

Списки в Groovy не обладают свойством не изменяемости, то-есть, они mutable. В частности можно изменить любой элемент списка, для этого достаточно инициировать его новым значением:
a[2]=77

print a -> [1, 2, 77, 4, 5]

Изменяемость списков имеет одно важное следствие, без учёта которого можно допустить грубую ошибку. Посмотрим на такой пример:

x = [1, 2, 3]
y = x
x[0] = 77
print y → [77, 2, 3]
y[1] = 22
print x → [77, 22, 3]

Значит, ***x*** и ***y*** всё время ссылаются на один и тот же список и изменение одной переменной приводит к изменению и другой тоже.

Для объединения двух списков в один применяется оператор конкатенации ***(+)***:

a = [1, 2, 3]
b = [4, 5]
c = a + b
print c → [1, 2, 3, 4, 5]

Добавить один элемент в список можно также с помощью оператора ***(+)***:

c = a + 8 → [1, 2, 3, 8]

При этом сам список ***a*** остаётся неизменным:

print a → [1, 2, 3]

Тот же результат получим и в такой нотации:

a + [8] → [1, 2, 3, 8]

В такой форме можно добавлять элемент и в начало списка:

c = [78] + a
print c → [78, 1, 2, 3]

Для добавления элемента в список есть ещё оператор (<<, синоним ***leftShift***):

c = a << 12 → [1, 2, 3, 12]
print c → [1, 2, 3, 12]

Интересно, что в данном случае список a изменяется:

print a → [1, 2, 3, 12]

При записи в такой форме:

c = a << [12]

Добавленный элемент будет представлять собой список:

print c → [1, 2, 3, [12]]

Можно добавить список с несколькими элементами:
a << [7, 8, 9]
print a → [1, 2, 3, [12], [7, 8, 9]]

Оператор (-) позволяет удалять заданные элементы из списка:
a = [1, 2, 3, 4, 5, 6, 7]
b = [3, 4, 5]
c = a — b
print c → [1, 2, 6, 7]

При этом не требуется, чтобы удаляемые элементы располагались компактно:
b = [2, 4, 6, 7]
a - b → [1, 3, 5]

Если среди списка удаляемых элементов окажутся не существующие в исходном списке, такие элементы просто игнорируются без сообщения об ошибке.

Элементы списков в Groovy могут иметь разные типы, без каких-либо ограничений:
a = [1, 5, "Hello", false, 0.785]
print a[2] →Hello

Так объявляется пустой список:
a = []

Инициирование переменной рангом создаёт список:
s = (1..5)
print s → [1, 2, 3, 4, 5]

Можно создавать ранг без последнего значения таким образом:
s = (1..<5)
print s → [1, 2, 3, 4]

В Groovy есть возможность создавать и настоящие массивы (не сильно отличаются от списков). Например:
int[] m = [1, 2, 3]
println m instanceof List → false

Здесь ***m*** – массив, метод ***instanceof*** определил, что это не список. Для объявления массива можно использовать и вариант с оператором ***as***:
def n = ["a", "b", "c"] as String[]

Можно создавать матрицы:
def mat = new Integer[3][3]

Матрица ***mat*** имеет размер ***3x3***, все её элементы имеют значение ***null***.

Пустая матрица ***a*** неопределённого размера создаётся так:

Integer[][][] a

Для массивов применимы все методы работы с векторами.

…..3.2 Применение итераторов для списков

Основной итератор ***each*** мы уже применяли для строк. Для списков он работает также — итератор принимает closure, обрабатывающий все элементы списка по очереди:

a = [1, 3, 4, 1, 8, 9]
a.each{print "${it**2} "} → 1 9 16 1 64 81

Итератор не создаёт нового списка, возвращается просто исходный список:

b = a.each{it**2}
print b → [1, 3, 4, 1, 8, 9]

Чтобы получить новый список, требуются дополнительные действия:

b = []
a.each{b += it**2}
print b → [1, 9, 16, 1, 64, 81]

Можно, конечно, и так:

b = []
a.each{b << it**2}
print b → [1, 9, 16, 1, 64, 81]

Есть также итератор ***collect***, который в данной ситуации предпочтительнее. ***Collect*** работает практически также, как и ***each***, но он создаёт новый список с элементами, полученными при обработке элементов исходного срписка:

s = ['groovy', 'ruby', 'scala', 'clojure']
t = s.collect {it.toUpperCase()}
print t → [GROOVY, RUBY, SCALA, CLOJURE]

В этом примере список ***s*** содержит элементы типа ***String***. Метод ***toUpperCase*** переводит все знаки текстовых переменных в верхний регистр. Итератор ***collect*** вернул результирующий список, которому мы присвоили идентификатор ***t***.

Иногда бывает полезен итератор ***times***, который просто повторяет приданный ему блок заданное число раз:

def i = 0
4.times{ print i++ } → 0123

Кстати, в Groovy имеется и вариант ***++i***. Посмотрите, в чём разница:

def i=0
4.times{ print (++i) } → 1234
Здесь скобки - ***(++i)*** обязятельны.
Конечно, можно и ***(i--)***:
def i=4
4.times{ print i-- } → 4321
Соответственно:
def i=4
4.times{ print (--i) } → 3210

Условный оператор ***while*** тоже работает, как итератор:
i = 4
while(i > 0){ print i-- } → 4321

…..3.3 Методы поиска в списках

Метод ***find*** позволяет определить, встречается ли в списке элемент с заданным значением:
a = [3, 7, 1, 12, 9, 7]
a.find {it == 7} → 7
a.find {it == 25} → null
Таким образом, если заданный элемент имеется в списке, метод ***find*** просто возвращает его значение; если же элемент не найден, возвращается ***null***. Поскольку в условных выражениях значение ***null*** (а также и ***0***) трактуется, как ***false***, а другие значения — как ***true***, можно программировать такую проверку:
if (a.find {it == 7}) print 'элемент 7 есть в списке'
else print 'элемента 7 нет в списке' → элемент 7 есть в списке
Так можно найти любой элемент, кроме ***0***.

Метод ***find*** на самом деле тоже является итератором, только в своём closure этот итератор должен содержать условное выражение. Элементы списка последовательно передаются в closure до тех пор, пока условие возвращает значение ***false***. Как только встретится элемент, для которого условие даст ***true***, цикл заканчивается, а значение данного элемента возвращается в качестве результата. Если элемент не будет найден до конца списка, итератор возвращает ***null***.
Условия поиска могут быть разными, например:
a.find {it > 7} → 12
Возвращается первый элемент больше ***7***.

Есть также итератор поиска ***findAll***, который возвращает все элементы списка, удовлетворяющие заданному условию:
a.findAll {it > 3} → [7, 12, 9, 7]
Результат возвращается в виде нового списка.

Метод ***findIndexOf*** вместо элемента, удовлетворяющего заданному условию, возвращает его индекс:
a.findIndexOf {it > 9} → 3
То-есть, первый элемент больше ***9*** имеет индекс, равный ***3***.

Приведём ещё один, более содержательный пример на применение метода ***findAll***. Создадим с помощью этого метода функции ***feven*** и ***fodd***, выбирающие из заданного списка чётные и нечётные члены соответственно:

```
def g = { f, s -> s.findAll(f) }
def even = { it % 2 == 0 }
def odd = { !even(it) }
def feven = g.curry(even)
def fodd = g.curry(odd)
println feven(0..8) → [0, 2, 4, 6, 8]
print fodd(0..8) → [1, 3, 5, 7]
```

Здесь использованы closure и их каррирование, рассмотренные нами ранее. (Пожалуй, тут придётся самостоятельно поразмышлять, как это работает)

…..3.4 Ещё разные методы для работы со списками

Метод ***sum*** позволяет вычислять сумму всех элементов списка (если, конечно, эти элементы — числа):

```
a = [2, 7, 5, 9]
a.sum() → 23
```

Замечание: некоторые встроенные методы Groovy , не имеющие параметров, требуют обязательного добавления пустых скобок, как, например, только что использованный нами метод sum. Другие методы позволяют эти скобки опускать. И, наконец, есть методы, для которых эти скобки вообще не допустимы. Трудно сказать, каковы причины для таких правил.

Посмотрим на ещё один пример использования метода ***sum***. Допустим, что нам надо в списке найти общее количество букв во всех элементах. Очевидно, что элементы должны иметь тип ***String***. Это можно запрограммировать примерно так:

```
a = ['Programming', 'In', 'Groovy']
```

s = 0
a.each { s += it.size() }
print s → 19
Тут использован метод ***size***, вычисляющий количество букв в отдельной строке. Кстати, это тот же метод ***size***, что и для списков. Но вот какие чудеса:
a = [1, 2, 3]
a.size → 3
a.size() → 3
То-есть, для списков пустые скобки можно ставить, или нет — как вам заблагорассудится (хотя зачем ставить лишние знаки, которые никому не нужны). Теперь для строк:
b = "Hello World"
b.size() → 11 - метод работает
b.size - ничего не выйдет, генерируется ошибка!

Подсчёт букв в списке можно выполнить эффектнее, применив итератор ***collect*** в сочетании с методом ***sum***:
a = ['Programming', 'In', 'Groovy']
print a.collect { it.size() }.sum() → 19
Здесь итератор ***collect*** сначала создаёт промежуточный список с элементами, равными числу букв в каждом слове исходного списка ***a***. Затем метод ***sum*** находит общую сумму.

Познакомимся всё на том же примере с ещё одним полезным итератором с названием ***injec***. Этот итератор кроме обработки каждого элемента коллекции по очереди (точно так же, как итераторы ***each*** и ***collec***t), позволяет ещё выполнить заданные действия над всеми элементами списка в совокупности.
println a.inject(0) { s, e -> s + e.size() } → 19
В данном случае closure принимает два параметра: параметр ***s*** предназначен для накапливания суммы (аккумулятор), а параметр ***e*** принимает значения элементов списка. Сам итератор ***inject*** также принимает один параметр (в данном случае ***0***), который задаёт начальное значение для аккумулятора ***s***. Приведём другой пример применения итератора ***inject***:
b = [5, 2, 3, 7]
p = b.inject(1) {r, x -> r * x}
print p → 210

Теперь мы вычислили произведение элементов списка и потому начальное значение аккумулятора принято равным ***1***.

Итератор ***join*** выполняет конкатенацию элементов списка. Кроме того, он принимает параметр типа ***String***, который используется, как разделитель между словами.

print a.join(' |**| ') → Programming |**| in |**| Groovy

Если элементы исходного списка имеют тип другой, чем ***String***, итератор сам предварительно преобразует их к типу ***String***:

```
a = [1, 2, 3, 4]
b = a.join(' ~ ')
print b → 1 ~ 2 ~ 3 ~ 4
b.class → String
```

Метод ***flatten*** позволяет «раскрывать» вложенные списки:

```
a = [1, 2, ["a", "b", "c"], [0.78, 1.3]]
b = a.flatten()
print b → [1, 2, a, b, c, 0.78, 1.3]
```

При этом глубина вложенности может быть любой:

```
a = [1, 2, [3, [4, 5], 6], 7]
b = a.flatten()
print b → [1, 2, 3, 4, 5, 6, 7]
```

Метод ***reverse*** изменяет порядок расположения элементов на обратный:

```
a = [1, 2, 3, 4, 5, 6, 7]
a.reverse() → [7, 6, 5, 4, 3, 2, 1]
```

Этот метод (как, кстати, и большинство других методов для работы со списками) применим и для строк:

```
b = "Hello World"
b.reverse() → dlroW olleH
```

Обычно метод ***size*** вычисляет размер списка:

```
a = ['Programming', 'In', 'Groovy']
print a.size() → 3
```

Есть также вариант этого метода, вычисляющего размеры всех элементов списка, если их тип ***String***. Для этого достаточно поставить знак ***(*)*** после идентификатора списка:

print a*.size() → [11, 2, 6]

Знак ***(*)*** в этой ситуации называют раскрывающим оператором; мы ещё встретимся с ним впоследствии. Обычно программисты

называют трюки подобного рода «синтаксическим сахаром». С помощью итератора ***collect*** этот же результат можно получить так:
print a.collect { it.size() } → [11, 2, 6]

Приведём пример ещё одного синтаксического сахара, позволяющего элементы для любой функции передавать в виде списка:
def f(a, b, c, d) {a + b + c + d}
a = [3, 5, 1, 6]
print f(*a) → 15
Тут также использована звёздочка (не обязательная). Ясно, что число элементов в списке должно равняться числу аргументов функции. Такой приём удобен, когда число аргументов функции достаточно велико, или когда передаваемые параметры используются неоднократно.

Знак (*****) перед идентификатором называется раскрывающим оператором, так как он применяется для раскрытия списка или хеша. Пусть требуется вставить в список ***b*** элементы списка ***a***:
def a = [4, 5]
def b = [1, 2, 3, a, 6]
print b → [1, 2, 3, [4, 5], 6]
В таком варианте список ***a*** становится одним из элементов списка ***b***. Применим раскрывающий оператор:
def a = [4, 5]
def b = [1, 2, 3, *a, 6]
print b → [1, 2, 3, 4, 5, 6]
Теперь элементы списка ***a*** стали элементами списка ***b***, то-есть, список ***a*** раскрылся.

Посмотрим теперь на пример с хешем (они рассматриваются далее):
def m = [c:3, d:4]
def n = [a:1, b:2, *:m]
print n → [a:1, b:2, c:3, d:4]
При вставке элементов хеша кроме звёздочки надо поставить ещё и двоеточие.

Имеется несколько методов, проверяющих наличие заданного элемента в списке. Методы ***in***, ***contains*** и ***isCase*** практически идентичны:
def s = ['лебедь', 'рак', 'щука']

println 'рак' in s → true
println s.contains('рак') → true
println s.isCase('ворона') → false

Списки удобно применять для группового присваивания в такой нотации:

(x, y) = [3, 9]
x → 3
y → 9

Так можно выполнять обмен значениями без ввода промежуточной переменной:

(x, y) = [y, x]
x → 9
y → 3

…..3.5 Ассоциированные списки

В документации Groovy этот вид коллекций называется ***map***. Будем, однако использовать более привычный и тоже краткий термин хеш— hash. Создание hash также просто, как и списка — достаточно инициировать какими-либо значениями:

h = ['Сергей' : 25, 'Константин' : 18, 'Наталья' : 17]

Посмотрим на тип объекта ***h***:

h.getClass() → LinkedHashMap (почему-то привычный метод ***class*** в данном случае возвращает ***null***)

Следовательно, объявление какого-нибудь ***hash*** создаёт экземпляр класса ***LinkedHashMap***.

Хеш представляет собой коллекцию из пар ключ — значение, разделённых двоеточием. Как и список, хеш ограничивается квадратными скобками.

Типы ключей и значений могут быть любыми без каких-либо ограничений. В нашем примере ключи (имена людей) представлены типом ***String***, а значения (возраст) — типом ***Integer***. В одном хеше могут быть представлены разные типы:

h1 = ["aa":23, 12:'NNN', true:12]

Для извлечения значения из хеша применяется тот же оператор ***[]***, что и для списков, только роль индексов играют ключи:

print h['Наталья'] → 17

Допустимо, в качестве синтаксического сахара, квадратные скобки опускать, а записывать ключ в нотации метода:

print h.'Сергей' → 25

Ещё один синтаксический сахар состоит в том, что в случае, когда ключи имеют тип ***String***, кавычки тоже можно опускать и при объявлении хеша и при извлечении значений (правда, при использовании кириллицы в этом случае некоторые буквы дают ошибку):

h = [Ruby : 'Maz', Java : 'Gosling', Lisp : 'McCarthy']
h.Ruby → Maz

Здесь ключи — названия языков программирования, а значения — авторы языков.

Очевидно, что применение в одном и том же хеше двух одинаковых ключей не имеет смысла, значения же могут быть какими угодно.

Так можно задать пустой хеш:

hs = [:]

Как и списки, хеши изменяемы и для них действует то же правило:

x = [a : 1, b : 2, c: 3]
y = x
x['a'] = 77
print y → [a:77, b:2, c:3]

Переменные ***x*** и ***y*** ссылаются на один и тот же хеш.

…..3.6 Итераторы для hash

Для хешей применимы те же итераторы, что и для списков, разумеется, с некоторыми модификациями. Посмотрим на пример использования итератора ***each***:

h = [Ruby : 'Maz', Java : 'Gosling', Lisp : 'McCarthy']
h.each {println "Язык ${it.key} разработал ${it.value}"}

В результате получим:

Язык Ruby разработал Maz
Язык Java разработал Gosling
Язык Lisp разработал McCarthy

Итератор ***each*** и в этом случае имеет свой closure. Фиксированной переменной ***it*** последовательно передаются пары ключ — значение (в документации их называют ***MapEntry***), к которым применимы два встроенных метода. Из каждой пары метод ***key*** извлекает ключ, а метод ***value*** — значение.

Можно обойтись и без этих встроенных методов, поскольку итератор ***each*** способен передавать в closure ключи и значения непосредственно:
h.each {x, y -> println "Язык $x разработал $y"} → тот же результат.
В этом варианте в closure передаются два параметра: переменная ***x*** принимает ключи, а переменная ***y*** – значения. Напоминаю, что правильно было бы писать ***${x}*** и ***${y}***, но для случая, когда выражения представлены только переменными, синтаксический сахар позволяет скобки опускать.

Аналогично и другие итераторы позволяют передавать в closure один параметр, и тогда потребуются методы ***key*** и ***value***, или два, представляющие ключ и значение непосредственно.

Итератор **collect** работает аналогично итератору ***each***, но дополнительно возвращает в качестве результата список. Например, чтобы получить список ключей хеша, можно действовать двумя способами:
a = h.collect {it.key}
a = h.collect {x, y -> x}
В обоих случаях список a будет содержать ключи:
print a → [Ruby, Java, Lisp]
b = h.collect {it.value }
b = h.collect {x, y -> y}
Теперь получим список значений:
print b → [Maz, Gosling, McCarthy]
Одновременно в closure можно выполнять какие-то действия над ключами или значениями, например:
a = h.collect {x, y -> x.toUpperCase()}
print a → [RUBY, JAVA, LISP]
Все буквы в списке ключей перевели в верхний регистр.

Итератор ***find*** отыскивает в хеше пару по заданному условию точно также, как это делается для списков. Условие может быть задано как для значений, так и для ключей, или для обоих вместе.
a = h.find {x, y -> y.size() > 3}
print a → Java=Gosling
print a.key → Java
print a.value → Gosling
В этом примере отыскивается первая пара, для которой значение имеет больше трёх букв. Результатом является эта самая ***MapEntry***, о

которой упоминалось ранее. Из неё можно извлечь ключ и значение методами ***key*** и ***value*** соответственно.

Имеется также метод ***keySet***, передающий итератору ***each*** ключи хеша. Посмотрите на пример:

```
h = ["China": 1 , "India" : 2, "USA" :3]
r = 0
h.keySet().each { r+= h[it] }
println r → 6
```

Метод ***any*** позволяет проверить, встречается ли в хеше пара, для которой выполняется заданное условие:

```
r = h.any {x, y -> y.size() > 7}
print r → true
```

А метод ***every*** проверяет, удовлетворяют ли заданному условию все пары хеша:

```
r = h.every {x, y -> y.size() > 7}
print r → false
```

Способ проверки в этих методах точно такой же, как и у метода ***find***.

Метод ***groupBy*** позволяет элементы хеша сгруппировать по какому-нибудь признаку. Проще всего понять работу этого метода на примере. Допустим, что у нас есть список людей с указанием имён и фамилий. Требуется разбить этот список на группы так, чтобы в каждой группе были люди с одинаковыми именами (но не с фамилиями). Это можно сделать приблизительно так:

```
h = [ k1 : 'Brian Goetz', k2 : 'Brian Sletten',
k3 : 'David Bock', k4 : 'David Geary',
k5 : 'Scott Davis', k6 : 'Scott Leberknight',
k7 : 'Stuart Halloway']
a = h.groupBy { it.value.split(' ')[0] }
println a
a.each { x, y -> println "$x : ${y.collect { k, n -> n }.join(', ')}"}
```

Мы представили список людей в виде хеша, в котором ключи обозначили ***k1, k2, k3*** и так далее, а значения — имя и фамилия через пробел. Методу ***groupBy*** придали closure, выполняющий следующие действия: ***it.value*** выделяет значения хеша, то-есть, имена и фамилии через пробел. К этим значениям применяем метод ***split(' ')***, который разбивает фразу из двух слов на два элемента, а аргумент ***(' ')*** указывает, что разбивать надо по пробелу между словами. В результате создаётся список из двух элементов, в котором первый

элемент представляет имя, а второй — фамилию. Далее оператор ***[0]*** извлекает первый элемент из этого списка, то-есть имя человека. Сам метод ***groupBy*** создаёт новый хеш в котором ключами становятся выделенные в closure имена, а элементами — вложенные хеши, представляющие группы элементов исходного хеша, содержащие эти имена. Вот как выглядит созданный методом ***groupBy*** хеш, обозначенный нами буквой ***a***:

[Brian:[k1:Brian Goetz, k2:Brian Sletten], David:[k3:David Bock, k4:David Geary], Scott:[k5:Scott Davis, k6:Scott Leberknight], Stuart: [k7:Stuart Halloway]]

Далее мы с помощью итератора ***each*** организуем вывод промежуточного хеша ***a*** на печать. Все запрограммированные здесь действия нам уже знакомы — вывод на печать каждой группы организован с помощью итератора ***collec***t. Метод ***join(', ')*** вставляет запятые с пробелом между элементами вложенных хешей. И вот как выглядит результат:

Brian : Brian Goetz, Brian Sletten
David : David Bock, David Geary
Scott : Scott Davis, Scott Leberknight
Stuart : Stuart Halloway

Создать хеш можно с помощью итератора ***eachWithIndex***, который передаёт в свой closure значения ключей:

```
k = 1..3
v = 'a'..'c'
d = [:]
k.eachWithIndex { it, i -> d[it] = v[i] }
println d → [1:a, 2:b, 3:c]
```

Здесь мы создали сначала пустой хеш ***d***, а затем в цикле инициировали его члены значениями. Ключи и значения предварительно представлены списками, созданными с помощью оператора ранга.

…..4. GDK

Совокупность разнообразных средств (инструментов) для облегчения разработки в Java называют JDK. В Groovy имеется возможность использования этого JDK, а кроме того, добавлены ещё и другие средства, обеспеченные новыми возможностями Groovy.

Набор этих дополнительных средств разработки по аналогии называют GDK (хотя, конечно, они не сопоставимы с JDK по объёму и возможностям). Далее рассмотрим кое-что из этого GDK.

…..4.1 Методы with, sleep

Метод ***with*** полезен, когда для одного и того же объекта приходится вызывать много разных методов, например:

```
a = [1, 2]
a.add(3)
a.add(4)
println a.size() → 4
println a.contains(2) → true
```

Здесь мы к списку ***a*** применили несколько методов и при вызове каждого из них указывали идентификатор списка и ставили точку. Это может показаться обременительным, особенно если идентификатор длинный (что очень широко практикуется в Java), и метод ***with*** позволяет избавиться от таких повторений:

```
a = [1, 2]
a.with() {
        add(3)
        add(4)
        println size() → 4
        println contains(2) → true
}
```

Конечно, это не более, чем всё тот же «синтаксический сахар», но далее мы встретим ситуацию, когда метод ***with*** существенно полезен.

Метод ***sleep***, позволяющий приостановку выполнения программы (потока), в Groovy имеет дополнительные возможности. Сначала посмотрим, как можно применять этот метод:

```
p = Thread.start {
   println "Старт потока"
   t1 = System.nanoTime()
   sleep(2000)
   t2 = System.nanoTime()
   println "Выполнено за ${(t2 - t1)/10**9} секунд"
}
sleep(100)
println "Выполняем прерывание потока"
```

p.interrupt()

Метод ***Thread.start*** запускает поток, которому можно дать имя (у нас ***p***) для ссылок на него в дальнейшем. ***System.nanoTime()*** возвращает текущее время в наносекундах. Аргумент у ***sleep*** задаётся в миллисекундах. При выводе мы затраченное время переводим в секунды. Метод ***interrupt*** прерывает поток. Повторный вызов ***sleep(100)*** требуется для того, чтобы успел сработать старт потока до оператора вывода. И так выглядит результат:

Старт потока
Выполняем прерывание потока
Выполнено за 1.999882533 секунд

В Groovy методу ***sleep*** можно передавать ***closure***, в котором параметр ***it*** принимает информацию об исключении. Кроме того closure имеет параметр ***flag***, позволяющий управлять прерыванием. Если задать переменной ***flag*** значение ***true***, то прерывание потока остановит и задержку времени методом ***sleep***. Если же ***flag*** задать равным ***false***, задержка будет выполнена в полном объёме, независимо от прерывания потока.

```
def f(flag) {
   p = Thread.start {
      println "Старт потока"
      t1 = System.nanoTime()
      sleep(2000) {
         println "Прерывание... " + it
         flag
      }
      t2 = System.nanoTime()
      println "Выполнено за ${(t2 - t1)/10**9} секунд"
   }
   p.interrupt()
   p.join()
}
f(true)
f(false)
```

Метод ***join*** требуется при повторных запусках потока. Дальше приведён результат этого примера:

Старт потока
Прерывание... java.lang.InterruptedException: sleep interrupted

Выполнено за 0.002430918 секунд
Старт потока
Прерывание... java.lang.InterruptedException: sleep interrupted
Выполнено за 2.000363689 секунд

…..4.2 Косвенный доступ к полям (свойствам) и методам класса

В Groovy принято называть поля класса свойствами; будем придерживаться этой терминологии.

В общем случае имеется свободный доступ к свойствам класса:

class A { String s }

Создали класс ***A***, имеющий одно свойство — переменную типа ***String***. При создании экземпляра класса можно инициировать это свойство с применением такого синтаксиса:

p = new A (s : "Hello")

Теперь можно получить значение ***s*** обычным способом:

print p.s → Hello

Также свободно можно изменять свойство:

p.s = "Привет"
print p.s → Привет

Имеется также хороший метод ***getProperties***, позволяющий получить для экземпляра название класса и все его свойства в виде списка:

p.getProperties() → [class:class A, s:Привет]

Любопытства ради можете вызвать метод ***getProperties*** для самого класса:

A.getProperties()

Получите огромный текст, дающий представление о том, насколько громоздким и неуклюжим является внутреннее представление класса в Java, а соответственно, и в Groovy.

Иногда полезно не задавать жёстко имена свойствам, к которым требуется иметь доступ (по чтению, или по записи), а иметь возможность задавать эти имена на этапе выполнения программы (runtime). Очень полезен такой приём в веб-проектировании, когда доступ к свойствам класса запрашивается из некоторых сторонних источников. Groovy позволяет реализовать подобный подход. Рассмотрим такой пример:

p = ['miles', 'fuel']
class Car {

```
    int miles, fuel
}
c = new Car(fuel: 80, miles: 25)
p.each {println "$it = ${c[it]}" }
c[p[1]] = 100
println "fuel now is ${c.fuel}"
```

Здесь мы задали имена свойств, к которым хотели бы получить доступ, предварительно в списке ***p***. Этот список может меняться динамически на этапе выполнения программы. В частности, может изменяться количество свойств в списке или их порядок. Доступ к свойству осуществляется с помощью оператора ***[]***. Текст ***c[p[1]] = 100*** эквивалентен тексту ***c['fuel'] = 100*** или ***c.fuel = 100***.

При выполнении примера получим:

```
miles = 25
fuel = 80
fuel now is 100
```

Для косвенного доступа к методам класса применяется метод ***invokeMethod***. Его возможности продемонстрируем на примере:

```
class Person  {
    def f() { println "Человек" }
    def f(name) { println "Его зовут $name" }
    def f(name, age) {println "Его зовут $name ему $age лет" }
}
p = new Person()
p.invokeMethod("f", null)
p.invokeMethod("f", 'Сергей')
p.invokeMethod("f", ['Сергей,', 25] as Object[])
```

Тут надо отметить следующие обстоятельства. Как видим, в классе могут быть несколько методов с одним и тем же именем. При вызове выбирается нужный метод по числу переданных параметров: если параметров нет, вызывается первый метод ***f***, если параметр один — второй, а если передаётся два параметра — вызывается третий метод ***f***. Имя самого метода передаётся методу ***invokeMethod*** в форме строковой переменной (у нас ***"f"***). Если передаётся больше одного параметра, их надо оформить в виде списка, к которому ещё требуется добавить директиву ***as Object[]***.

Приведём результат этого примера:

```
Человек
```

Его зовут Сергей
Его зовут Сергей, ему 25 лет

Как и при косвенном доступе к свойствам, этот трюк позволяет вызывать нужные методы в динамике, на этапе runtime. Например, можно вводить имя нужного метода и параметров к нему с клавиатуры во время выполнения программы:

```
class A  {
   def f1(x) { print "Это f1 с параметром x = $x" }
   def f2(x) { print "Это f2 с параметром x = $x" }
   def f3(x) { print "Это f3 с параметром x = $x" }
}
p = new A()
s = System.console().
readLine "введите имя метода: f1, f2, f3: "
x = System.console().
readLine "введите параметр x: "
p.invokeMethod(s, x)
```

Ввод с клавиатуры будет выглядеть так:
введите имя метода: f1, f2, f3: - вводим ***f1***
введите параметр x: - вводим ***77***
Получим такой результат:
Это f1 с параметром x = 77

Имеется также метод ***getMetaClass()***, который можно вызвать для экземпляра класса (или для самого класса). Метод выдаёт некоторую информацию, не слишком, правда, полезную.

…..4.3 Программный доступ к файлам

Операции чтения из файла и записи в файл в Groovy упрощены до предела. Это, пожалуй, первое, действительно стоящее средство из этого самого GDK.

Для того, чтобы открыть файл для чтения, достаточно создать экземпляр библиотечного класса ***File***, передав ему, как параметр, название файла в кавычках. В названии может быть полный путь, или относительный.

```
r = new File('prob.groovy')
s = r.text
```

print s - выводится текст файла ***prob.groovy***.

Созданный нами объект ***r*** при распечатке выдаёт просто название файла, но для него можно вызвать встроенный метод ***text***, который возвращает весь текст файла (мы этим текстом инициировали переменную ***s***). Всё это можно записать и одной строкой:

print new File('prob.groovy').text

Приведём также пример, в котором указан полный путь для читаемого файла:

```
r = new File('C:/Users/Boris/Documents/ocaml/stac.ml')
s = r.text
print s →получили такой текст:
let push c x = x :: c;;
let pop s = match s with
  | h::t -> (h,t)
  | [] -> raise Empty;;   - это текст программы на языке Ocaml.
```

Если требуется читать текст по строкам, можно применять итератор ***eachLine***. Пусть в файле ***output.txt*** содержатся первые четыре строчки начала поэмы Пушкина «Руслан и Людмила». Тогда можно прочитать этот текст по строкам примерно так:

```
r = new File('output.txt')
a = []
s = r.eachLine { a+= it }
println s → [У лукоморья дуб зелёный, Златая цепь на дубе том., И днём и ночью кот учёный, Всё ходит по цепи кругом.]
print s[2] → И днём и ночью кот учёный
```

Здесь мы создали список ***s***, элементы которого представлены строчками поэмы и мы можем извлечь любую строку по индексу. Для поиска строк по какому-либо признаку применяется метод ***filterLine:***

```
r = new File('output.txt')
s = r.filterLine { it =~ /цеп/ }
println s →
Златая цепь на дубе том.
Всё ходит по цепи кругом.
```

С помощью регулярного выражения отобрали строки, в которых встречается сочетание букв "цеп".

Для записи в файл можно применять метод ***withWriter***:

```
s = '''Идёт направо - песнь заводит,
Налево - сказки говорит'''
```

r = new File("output.txt")
r.withWriter{ it << s }

В текстовой переменной ***s*** содержится текст, предназначенный для вывода в файл ***output.txt***. Текст записан в тройных кавычках, поскольку он содержит больше одной строки. Для того, чтобы открыть файл для записи также достаточно только создать экземпляр класса ***File***, передав ему название файла (полное или относительное). Метод ***withWriter*** принимает closure, в котором надо выводимый текст передать параметру ***it*** с помощью оператора ***(<<)***. Если в файле ***output.txt*** уже содержится текст, он будет уничтожен — можно прочесть содержимое файла и убедиться, что в нём только две последние строчки. Если записываемый файл до этого не существовал, он будет создан.

Имеется ещё целый ряд методов для организации потоков ввода-вывода.

…..4.4 Использование классов и скриптов Groovy, расположенных в отдельных файлах

Классы, предназначенные для многократного использования, в особенности, если они велики по объёму кода, целесообразно размещать в отдельных файлах. Использовать такие классы в Groovy совсем просто. Пусть у нас имеется такой класс:

```
class Person {
   def f(name, age) {
      println "$name имеет возраст $age лет"
   }
}
```

Необходимо поместить этот класс в файл с таким же именем: ***Person.groovy***. Программу, в которой мы хотим использовать этот класс поместим в ту же директорию, например под именем ***prob.groovy:***

p = new Person ()
p.f('Анна', 19) → Анна имеет возраст 19 лет

Теперь достаточно запустить нашу программу, например, из командной строки, или из текстового редактора Groovy. Подключение файла с классом произойдёт автоматически, так, как будто всё располагается в одном файле:

Если файлы располагаются в разных директориях, необходимо указывать полный путь.

Файл с классом можно предварительно откомпилировать командой:

groovyc Person.groovy

В результате компиляции будет создан файл ***Person.class.*** Программа ***prob.groovy*** запускается как обычно. Поскольку класс уже откомпилирован, выполнение программы будет происходить быстрее.

Так же просто можно из текущей программы (скрипта) выполнить программу (скрипт), расположенную в отдельном файле. Пусть у нас в файле ***rab.groovy*** имеется такая программа:

```
def f(x) {
   x**2
}
print "Res = ${f(3)}"
```

Для того, чтобы выполнить этот код из другой программы, надо создать экземпляр класса ***GroovyShell***. Далее используется встроенный метод ***evaluate***, принимающий в качестве параметра экземпляр класса ***File***:

```
r = new GroovyShell()
r.evaluate(new File('rab.groovy')) → Res = 9
```

Программа ***rab.groovy*** в этом варианте не может рассматриваться, как процедура в общепринятом понимании этого слова, поскольку нет возможности как передать в неё параметры, так и получить результат. Для того, чтобы получить результат из вызываемой программы можно применить метод ***binding*** (на самом деле это экземпляр класса, но для нас это не важно) . Пусть теперь программа в файле ***rab.groovy*** имеет такой вид:

```
def f(x) {
   x**2
}
y = f(5)
f(3)
```

То-есть, теперь эта программа только возвращает результат, но ничего не выводит на печать. Применяем метод ***binding***:

```
r = new GroovyShell(binding)
s = r.evaluate(new File('rab.groovy'))
println ("Res = $s") → Res = 9
```

print y → 25

Значит, в этом варианте переменная ***s*** получает значение, возвращаемое вызываемой программой ***rab.groovy***. Кроме того, в вызывающей программе доступны переменные, объявленные в вызываемой программе (у нас это переменная ***y***).

Для передачи параметров в вызываемую программу применяется метод setProperty. Пусть теперь программа ***rab.groovy*** будет такой:

```
def f(x) {
   x**2
}
f(y)
```

Установить значение переменной ***y*** из вызывающей программы можно так:

```
b1 = new Binding()
b1.setProperty('y', 8)
r = new GroovyShell(b1)
s = r.evaluate(new File('rab.groovy'))
println ("Res = $s") → Res = 64
```

Метод ***setProperty*** принимает два параметра: первый — идентификатор переменной (в кавычках) , второй — значение этой переменной. В таком варианте вызываемая программа становится полноценной процедурой.

…..4.5 Метапрограммирование

Метапрограммирование позволяет изменять программный код на этапе runtime (на лету). В частности есть возможность вызывать методы и свойства в динамике. Метод ***getMetaMethod*** принимает название динамически вызываемого метода. Посмотрим на примере, как это работает:

```
s = "hello"
mN = System.console().
        readLine "Введите имя встроенного метода: "
mI = s.metaClass.getMetaMethod(mN)
println mI.invoke(s)
```

В этом фрагменте мы что-то делаем с переменной ***s*** типа ***String***. С помощью метода ***getMetaMethod***, принадлежащего объекту ***metaClass***, для метода, имя которого мы ввели с клавиатуры и присвоили его переменной ***mN***, создаётся некий новый метод ***mI***.

Обратите внимание, что при создании метода ***mI*** используется также и переменная ***s***. После всего этого метод ***mI*** можно выполнить с помощью знакомого уже нам оператора ***invoke***, которому в качестве параметра надо передать переменную ***s***. Работа с этим фрагментом может выглядеть так. При запуске программы в командной строке появится приглашение на ввод имени нужного нам метода:
Введите имя встроенного метода: toUpperCase
Будет выведено — ***HELLO***
Другой вариант:
Введите имя встроенного метода: reverse
Будет выведено — ***olleh***

Следовательно, по своему желанию на этапе runtime можно фактически менять код программы, выполняя над текстовой переменной нужные действия.

Очевидно, что можно работать и с другими объектами. Например, инициируем переменную ***s*** списком:
s = [1.2, 5.07, 3, 4]
и введём название метода ***sum***:
Введите имя встроенного метода: sum
В результате получим ***13.27***

Есть также другая, очень простая нотация, позволяющая делать то же самое:

```
def f(x)  {
   mN = System.console().
      readLine "Введите имя встроенного метода: "
   print x."$mN"()
}
```

f('hello') → Введите имя встроенного метода: toUpperCase
Получим: **HELLO**
Этот вариант - тоже синтаксический сахар; теперь нужные методы метапрограммирования вызываются неявно.

Эта техника имеет много различных возможностей, освоить которые не трудно.

Groovy обладает инструментами тесного взаимодействия с Java. Фактически можно использовать все средства и библиотеки Java. Для того, чтобы это взаимодействие применять на практике, надо хорошо овладеть и языком Groovy и языком Java. Рассматривать этот материал, на мой взгляд, и скучно и утомительно. Лучше напоследок

проанализируем ещё ряд практических примеров программирования на Groovy, взятых из разных источников.

…..5. Некоторые характерные примеры

…..5.1 Передача аргументов в программу при запуске её из командной строки

Обратимся опять к программе приветствия. Поместим в файл ***rab.groovy*** такой код:

```
println "hello, world"
for (x in args ) {
   println "Argument:" + x
}
args.each{ println "hello, $it" }
print args
```

Запустим программу из командной строеи командой:

groovy rab.groovy "У лукоморья дуб зелёный"

Получим такой результат:

```
hello, world
Аргумент:У лукоморья дуб зелёный
hello, У лукоморья дуб зелёный
[У лукоморья дуб зелёный]
```

Текст после названия файла передаётся в программу в виде списка под названием ***args***. В нашем случае весь текст в кавычках и потому воспринимается, как один элемент списка ***args***. Если же текст не заключать в кавычки, каждое слово воспринимается теперь, как отдельный элемент списка. При этом разбивка текста происходит по пробелам. В примере мы вывели на печать элементы списка двумя способами. В итоге будет другой результат:

```
hello, world
Аргумент:У
Аргумент:лукоморья
Аргумент:дуб
Аргумент:зелёный
hello, У
hello, лукоморья
hello, дуб
```

hello, зелёный
[У, лукоморья, дуб, зелёный]

…..5.2 Перечисление (enum) и классический переключатель (switch- case)

Пусть нам требуется по имени дня недели определить, рабочий это день, или выходной. Это можно запрограммировать, например, так:

```
enum Day { Воскресенье, Понедельник, Вторник, Среда,
   Четверг, Пятница, Суббота }
def f(d) {
   switch (d) {
      case [Day.Воскресенье, Day.Суббота]: println "Выходной"
         break
      case Day.Понедельник..Day.Пятница: println "Рабочий день"
         break
     default: println "Такого дня нет"
   }
}
print f(Day.Суббота) → Выходной
print f(Day.Четверг) →  Рабочий день
```

С помощью ключевого слова ***enum*** объявляется так называемое перечисление — разновидность коллекции. После слова ***enum*** записывается название перечисления (дальше будем называть его enum) с заглавной буквы, а дальше в фигурных скобках перечисляются элементы. Элементами enum может быть текст без кавычек. Для получения элемента надо указать название данного enum и через точку сам элемент, то-есть, применив оператор печати получим:

```
print  Day.MONDAY →  MONDAY
```

Очевидно, что такая операция не имеет никакого смысла. Но enum иногда может быть полезным, например, его можно использовать в переключателе ***switch-case***, как это и сделано в нашем примере. Сам переключатель традиционный, последняя ветвь должна содержать оператор ***default***, который будет выполнен, если не одно ***case*** не дало ***true***.

Для сравнения покажем, как это же самое можно запрограммировать, применив вместо enum список:

```
day = [ Воскресенье, Понедельник, Вторник, Среда,
```

```
        Четверг, Пятница, Суббота]
def f(d) {
    switch (d) {
        case [day[0], day[6]]: println "Выходной"
            break
        case day[1..5]: println "Рабочий день"
            break
        default: println "Такого дня нет"
    }
}
```

f("Saturday") → Такого дня нет (не на том языке)

Покажем пример, в котором enum имеет более содержательную информацию. В частности, enum может иметь свойства, конструкторы и методы:

```
enum Planet {
    Меркурий (3.303e+23, 2.4397e6),
    Венера (4.869e+24, 6.0518e6),
    Земля (5.976e+24, 6.37814e6),
    Марс (6.421e+23, 3.3972e6),
    double mass
    double radius
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    def f() {
        println "${name()} имеет массу ${mass} " +
            "и радиус ${radius}"
    }
}
```

Planet.Земля.f() → Земля имеет массу 5.976E24 и радиус 6378140.0

В этом коде отдельные фрагменты представляют:

свойства -

```
double mass
double radius
```

конструктор -

```
Planet(double mass, double radius) {
        this.mass = mass;
```

```
        this.radius = radius;
    }
```

метод -

```
def f() {
    println "${name()} имеет массу ${mass} " +
          "и радиус ${radius}"
}
```

Данный enum содержит элементы, представленные их именами с дополнительной информацией. В конструкторе эта информация расшифровывается. В теле метода ***f*** использован встроенный метод ***name***, извлекающий имена элементов.

При вызовах элементов можно избавиться от необходимости каждый раз добавлять впереди название enum. Для этого в нашем примере надо ввести строку:

import static Planet.*

Тогда вместо строки

Planet.Земля.f()

можно будет писать просто

Земля.f()

Этот приём работает и при вызовах библиотечных функций, например:

import static java.lang.Math.*

Теперь вместо

Math.sin(x)

можно писать

sin(x)

Знак звёздочки указывает, что импортируются все функции библиотеки ***Math***. Но можно импортировать только нужную функцию:

import static java.lang.Math.sin

При импорте можно вводить синонимы для функций, если это даст какие-то дополнительные удобства:

import static java.lang.Math.sin as sine

Теперь можно писать

sine(x)

…..5.3 Не постоянное количество аргументов функции

Посмотрим на такой пример:

```
def f(Double... sp) {
   def s = 0
   for (int i = 0; i < sp.size(); i++) { s += sp[i] }
   return s
}
print f(1, 2, 3) → 6.0
print f(2.3, 0.9, 3, 5, 1.2)  → 12.4
```

Здесь задан список аргументов функции ***f*** в оригинальной форме ***(Double... sp)***, то-есть, указан тип аргумента и через многоточие — идентификатор аргумента. Такая форма, как видим, позволяет задавать разное количество аргументов.

Служебное слово ***def*** можно всюду заменять на указатель типа, например:

```
int f(Double... sp) {
   Double s = 0
   for (int i = 0; i < sp.size(); i++) { s += sp[i] }
   return s
}
println f(1, 2, 3) → 6
print f(2.3, 0.9, 3, 5, 1.2)  → 12
```

Записывая ***int f***, мы задаём тип возвращаемому функцией ***f*** результату, а при слове ***def*** этот тип выводится компилятором. Теперь заменим строку ***Double s = 0*** на ***int s = 0***:

```
int f(Double... sp) {
   int s = 0
   for (int i = 0; i < sp.size(); i++) { s += sp[i] }
   return s
}
println f(1, 2, 3) → 6
print f(2.3, 0.9, 3, 5, 1.2) → 11
```

Подумайте, почему изменился второй результат.

Есть и другая форма для задания списка аргументов переменной длины:

```
Double f(Double[] sp) {
   Double s = 0
   for (int i = 0; i < sp.size(); i++) { s += sp[i] }
   return s
}
```

println f(1, 2, 3) → 6.0
print f(2.3, 0.9, 3, 5, 1.2) → 12.4

Собственно, говоря, оператор ***[]*** просто объявляет список.
Но напоминаю, что есть встроенная функция ***sum***:
print([2.3, 0.9, 3, 5, 1.2].sum()) → 12.4

Вместо заданного типа элементов можно применять боле общий вариант ***Object***, а впереди списка с произвольным числом аргументов могут находится одиночные аргументы:

```
def f( x, Object[] e )  {
        def a= [x]
        e.each{ a<< it }
        a
}
println f( 1 ) → [1]
print f( 1, 2, 3, 4 ) → [1, 2, 3, 4]
```

…..5.4 Наибольший общий делитель (gcd)

С помощью одного лишь условного оператора ***if*** и рекурсии можно реализовать довольно сложный по логике известный алгоритм нахождения наибольшего общего делителя для двух чисел:

```
def gcd( m, n ) { if( m%n == 0 ) return n; gcd(n,m%n) }
print gcd( 28, 35 ) → 7
```

Оператор (***%***) возвращает остаток от деления двух целых чисел.

…..5.5 Некоторые дополнительные трюки при использовании closure

Closure может принимать именованные параметры. Эти параметры будут представлены в виде хеша, которым будет инициирован первый параметр, независимо тот того, где именованные параметры располагаются в списке. Фраза получилась довольно туманной, но посмотрим на пример:

```
def f= {m, i, j-> i + j + m.x + m.y }
print f(6, x:4, y:3, 7) → 20
```

Здесь с помощью оператора ***(:)*** в выражениях ***x:4*** и ***y:3*** инициируются значения переменных ***x*** и ***y***, которые, тем самым, становятся именованными параметрами. При такой передаче параметры closure примут такие значения:

i = 6

j = 7
m = [x : 4, y : 3]
Хешем становится параметр ***m*** потому, что именно он располагается на первом месте в списке. Вместо ***m.x*** и ***m.y*** можно, разумеется, писать ***m['x']*** и ***m['y'].***

Подобным образом можно использовать именованные параметры для любой функции. Посмотрим на примерах:
def f(x,y,z) {[x,y,z]}
Функция ***f*** просто возвращает свои аргументы в виде списка. Попробуем передавать функции ***f*** именованные параметры:
f(a:1,b:2,3,4) → [[a:1, b:2], 3, 4]
Значит, переменная ***x*** получила значение в виде хеша, составленного из именованных параметров, а переменные ***y*** и ***z*** инициированы не именованными параметрами. Это правило соблюдается независимо от того, где конкретно располагаются именованные параметры:
f(a:1,3,b:2,4) → [[a:1, b:2], 3, 4]
f(4,3,b:2,a:1) → [[b:2, a:1], 4, 3]
Все именованные параметры объединяются в хеш и передаются одному аргументу функции. Поэтому, например, такой вызов функции ***f*** приведёт к ошибке:
f(a:1, b:2, c:3)
Требуется передать три параметра, а в этом вызове функции ***f*** – только один.

Как и для функций, значения параметров для closure можно задавать по умолчанию:
def g = { a, b, c=3, d='Hello' -> "${a+b+c} $d" }
println g(2, 1) → 6 Hello
print g(9, 8, 7, 'Привет!') → 24 Привет!

Параметры ***c*** и ***d*** заданы по умолчанию. Если при вызове closure эти параметры не задавать, будут использованы значения по умолчанию. Но можно задать новые значения, и тогда те, что по умолчанию, будут проигнорированы. Как обычно, параметры по умолчанию должны располагаться в конце списка.

Closure могут принимать разное количество аргументов. Обычно для этого используется список, для которого дальше можно применять итератор ***each***. Впереди списка могут стоять одиночные параметры.
def g = { a, Double[] p -> def s = []

```
    p.each{ s<< it**2 }
    println "$a plus $s"
}
g('ttt', 5) → ttt plus [25]
g('ttt', 2, 8.3, 0.7) → ttt plus [4, 68.89, 0.49]
```

Выражение ***Double[] p*** объявляет список ***p*** с элементами типа ***Double***. Если не нужно объявлять тип, можно использовать выражение ***Object[] p***. Могут, конечно, применяться и другие итераторы, например:

```
def g = { a, Double[] p -> s = p.collect{it**2 }
    println "$a plus $s"
}
```

Этот вариант в точности соответствует предыдущему.

Closure может иметь переменное количество вложенных closure. Для этого используется директива ***Closure[]***, позволяющая объявить список с элементами, представленными разными closure. Дополнительно надо ещё применить выражение ***as List***. После этого к списку можно применять итераторы:

```
def g = { a, Closure[] s ->
    (s as List).inject(a){ q, f-> f(q) }
}
print g(7){it*3}{it+1}{it*2}.toString() → 44
```

В этом примере мы применили три вложенных closure.

Для удобства, группу параметров closure можно объединять в список (если эти параметры — не вложенные closure):

```
def g= {a, b, c-> a + b + c}
def s = [1, 2, 3]
print g(s) → 6
```

Для параметров closure (как и для параметров любой другой функции) можно объявлять тип:

```
def f = { int[] x -> x.inject(1){ s, t-> s * t } }
print f(4, 2, 3) → 24
```

Параметр ***x*** объявлен, как список с элементами типа ***int***. В данном closure используется итератор ***inject*** со своим собственным closure.

Если тип параметра ***x*** не объявлять, придётся поступать так:

```
def f = { x -> x.inject(1){ s, t-> s * t } }
print f([4, 2, 3]) → 24
```

То-есть, аргументом функции ***f*** теперь должен быть список.

Есть возможность применять анонимные (не именованные) closure. Вот такой своеобразный трюк позволяет организовать итерации анонимного closure с помощью директивы ***call***.

```
def fib = [];
{ a, b -> fib << a; a<10 && call(b, a+b)}(1,1)
print fib → [1, 1, 2, 3, 5, 8, 13]
```

Параметры для такого closure задаются в конце. В этом примере вычисляются числа Фибоначчи. Обращаю внимание на необходимость поставить точку с запятой при объявлении списка ***fib***.

…..5.6 Некоторые важные особенности функций в Groovy

Функции в Groovy могут иметь параметры, позволяющие как передавать информацию функции, так и получать информацию из функции.

```
def f( s, v ){
   s << v
}
x = []
f(x, 1)
f(x, 2)
println x → [1, 2]
print x == f(x, 3) → true
```

Функция ***f*** получает два параметра: список ***s*** (то, что это список следует из оператора (<<), добавляющего элемент в список) и переменную произвольного типа ***v***. При каждом вызове функции ***f*** к заданному списку добавляется очередной элемент. Таким образом, параметр ***x*** в нашем примере и передаёт информацию функции и получает её обратно. Сама функция ***f*** возвращает тот же самый список, что и передаваемый параметру ***x***; мы убедились в этом с помощью условного оператора. А что получится, если функцию объявить со словом ***void*** вместо ***def***?

```
void f( s, v ){
   s << v
}
x = []
f(x, 1)
f(x, 2)
println x → [1, 2]
```

print x == f(x, 3) → false

Теперь функция возвращает ***null***, но по-прежнему добавляет элемент в список.

При объявлении функции со словом ***void***, она всегда возвращает ***null***, если только возвращаемое значение не определяется оператором ***return*** - в этом случае компилятор фиксирует ошибку.

В Groovy имеется специальная функция ***call***, которая позволяет обращаться к ней неявно. Посмотрим на пример:

```
class A  {
   def call(x) { x**2 }
}
def p = new A()
println p.call(7) → 49
println p(9) → 81
```

Здесь ***p.call(7)*** — явный вызов метода, а вариант p(9) показывает, что явно название специального метода ***call*** можно не указывать, то-есть, метод вызывается по умолчанию.

…..5.7 Примеры работы с классами

В конструкторе классов можно не вводить новых идентификаторов для переменных, а использовать оператор ***this***:

```
class A  {
   def x
   A(x){this.x = x}
}
def p = new A(77)
print p.x → 77
```

Здесь в конструкторе параметр класса и переменная экземпляра класса имеют один и тот же идентификатор ***x***, что целесообразно, поскольку без необходимости не следует вводить новые идентификаторы. Но, вообще говоря, это разные переменные.

Посмотрим на типичный пример абстрактного класса:

```
class Car  {
   def name
   def model
}
def s = [
    new Car(name:'Renault', model:'Clio'),
```

```
    new Car(model:'508', name:'Peujeot')
]
def a = s.name
print a → [Renault, Peujeot]
```

Класс ***Car*** содержит только объявление двух полей (свойств) без их инициализации. Список ***s*** имеет элементы, представленные экземплярами класса ***Car***, в которых задаются именованные аргументы (тогда порядок их расположения не имеет значения). Вызов имени марки автомобиля (***s.name***) формирует список.

Рассмотрим теперь более сложный случай использования полей класса.

```
class Person  {
   def name
   def age
}
def f(s, g)  {
     s.collect{ g(it) }
}
def des(Person p)  {
   "$p.name is $p.age"
}
def q = this.&des
def a = [
   new Person(name:'Bob', age:42),
   new Person(name:'Julia', age:35) ]
def h = f(a, q)
println h → [Bob is 42, Julia is 35]
```

Класс ***Person*** такой же абстрактный, как и класс ***Car***. Но теперь над списком экземпляров класса ***Person a*** выполняются некоторые манипуляции с помощью функции ***f***. Кроме списка ***a*** функции ***f*** передаётся closure ***q***, использующий в свою очередь дополнительную функцию ***des***, принимающую параметр ***p*** типа ***Person***, ибо создавая класс, мы создаём новый тип. Для того, чтобы передать функцию ***des*** в closure ***q***, использована ссылка вида ***this.&des***. Здесь знак ***&*** - так называемый ссылочный оператор, а оператор ***this*** играет обычную роль; он указывает, что функция ***des*** будет применяться к тому объекту, для которого она вызвана. Конечно, ссылку можно было бы заменить ещё одним вложенным closure, такой приём мы уже

рассматривали ранее, но ссылочный оператор позволяет применять уже готовые функции, без их модификации под closure.

Имеется также интерфейс ***implements Iterable***, позволяющий работать с экземплярами класса, обрабатывая их в цикле. В примере используется класс ***Comp***, идентичный классам ***Car*** и ***Person***.

```
class Comp  {
   def id
   def name
}
s = [new Comp(id:1, name:'Peter'), new Comp(id:2, name:'Bob')]
class C implements Iterable <Comp> {
   def x
   C(x) {this.x = x}
   Iterator <Comp> iterator() { x.iterator() }
}
p = new C(s)
println p*.id → [1, 2]
print p*.name → [Peter, Bob]
```

Указатель <***Comp***> выбирает нужный класс из нескольких. В нашем примере класс только один и этот указатель не обязателен. Комбинация знаков (*******.) также называется раскрывающим оператором, а вариант без звёздочки здесь не работает. В классе ***C*** мы ввели явный конструктор для передачи экземпляру списка ***s***. В конструкторе принят один и тот же идентификатор ***x*** для получаемого параметра и для переменной экземпляра. В таком варианте необходимо использовать директиву ***this.*** Техника работы в примере не сложная, хотя трудно понять детали всего процесса. К тому же пример слишком тривиальный, чтобы можно было оценить возможности данного интерфейса.

Рассмотрим пример с использованием встроенных методов ***getAt*** и ***putAt***. Применение директивы ***[]*** на самом деле приводит к вызову одного из этих методов. Посмотрим на примере:

```
s = [1, 2, 3]
print s[1] → 2
```

В действительности здесь применяется метод ***getAt***:

```
print s.getAt(1) → 2
```

Если оператор ***[]*** использован слева от знака равенства, вызывается метод ***putAt***:

s[1] = 77
print s → [1, 77, 3]
А на самом деле:
s.putAt(1, 88)
print s → [1, 88, 3]

Groovy позволяет переопределять (override) методы классов, в том числе и встроенные методы. Для этого достаточно объявить их заново:

```
class User  {
   def id
   def name
   def getAt(i)  {
      switch(i)  {
         case 0: return id
         case 1: return name
      }
      throw new IllegalArgumentException ("Не найден элемент $i")
   }
   void putAt(i, v)  {
      switch(i)  {
         case 0: id = v; return
         case 1: name = v; return
      }
      throw new IllegalArgumentException ("Не найден элемент $i")
   }
}
def p = new User()
p[1] = 'Anna'
println p[1] → Anna
p[3] = 'Peter' → IllegalArgumentException: Не найден элемент 3
```

В этом примере переопределены методы ***getAt*** и ***putAt***, в результате чего получена возможность обращаться к свойствам (полям) класса по индексу и возвращать исключение при недопустимом индексе (в примере приведен сокращённый текст исключения, на самом деле выводится, как всегда в Java, много ненужных слов). Отметим, что в конструкции ***switch — case*** вместо необходимого варианта ***default*** использован другой приём. Метод ***putAt*** объявлен со словом ***void***, поскольку он ничего не возвращает по смыслу.

Естественно, что к полям класса можно при этом обращаться и по их идентификаторам:

p.id = 5

println p.id → 5

Рассмотрим ещё один характерный пример переопределения (перегрузки) метода.

Все арифметические и логические операторы кроме привычных обозначений имеют словесные синонимы. Например, оператор сложения (+) имеет синоним ***plus***:

4 + 9 → 13

4.plus(9) → 13

Синоним оператора (-) - ***minus***, оператора (*) - ***multipl***у и так далее. Переопределим оператор ***plus*** так, чтобы он мог складывать экземпляры классов:

```
class Prob  {
   def x
   Prob(x) {this.x = x}
   Prob plus (Prob m) { new Prob(x + m.x) }
}
def b1 = new Prob(3.7)
def b2 = new Prob(1.29)
def b = b1 + b2
print b.x → 4.99
```

Всё определяется строкой

Prob plus (Prob m) { new Prob(x + m.x) }

которую надо понимать так. Функция ***plus*** возвращает результат типа ***Prob***, и принимает аргумент ***m*** того же типа ***Prob.*** Результатом является экземпляр класса ***Prob*** с аргументом ***(x + m.x)***. По-прежнему работают оба синонима операции сложения, то-есть можно складывать и так:

```
def c = b1.plus(b2)
print c.x → 4.99
```

Если в классе Prob создать метод:

Prob plus(y) { new Prob(x + y) }

тогда можно будет выполнять и такое сложение:

```
def z = b1 + 1.29
print z.x → 4.99
```

Конкретный смысл метода ***plus*** для экземпляров классов может быть каким угодно, например, можно заставить метод выполнять умножение вместо сложения:

```
class Prob  {
   def x
   Prob(x) {this.x = x}
   Prob plus (Prob m) { new Prob(x * m.x) }
}
def b1 = new Prob(3.7)
def b2 = new Prob(2.0)
def b = b1 + b2
print b.x → 7.40
```

Отметим попутно, что как и при объявлении любой функции (а также и переменной), при создании экземпляра класса допустимы следующие варианты:

1)Как в примере, с помощью ключевого слова def:

```
def b1 = new Prob(3.7)
```

2)Слово ***def*** можно опустить:

```
b1 = new Prob(3.7)
```

3)Вместо ***def*** можно указать возвращаемый тип. Тип экземпляра класса всегда называется по имени класса:

```
Prob b1 = new Prob(3.7)
```

В нашем примере все эти варианты дают одинаковый результат, но вообще различие между ними то же, что и для функций и мы это уже обсуждали.

К полям и методам класса можно обращаться не используя идентификатор для экземпляра класса:

```
class Prob  {
   def f() { print 'Hello' }
}
new Prob().f() → Hello
```

Для ясности можно поставить скобки:

```
(new Prob()).f()
```

Техника метапрограммирования позволяет встраивать новые методы в существующие классы. В следующем примере с помощью метода ***metaClass*** и встроенной переменной ***delegate*** в класс ***String*** вставлен новый метод ***g***:

```
class Prob {
```

```
   def f() {
      String.metaClass.g = { "Привет, $delegate" }
      println 'какой-то текст'.g()
   }
}
new Prob().f() → Привет, какой-то текст
```

Применение метода ***g*** к любому тексту создаёт новый текст, в котором в начале стоит слово ***Привет***, а на место ***delegate*** вставляется тот текст, для которого вызван метод ***g***.

Метапрограммирование позволяет также вставлять методы из одного класса в другой класс. Например:

```
class A {
   def f(x) { println "$x из класса A" }
}
class B {
   def g() { println "Это класс B" }
}
def p = new A()
B.metaClass.fi = p.&f
def q = new B()
q.fi(555) → 555 из класса A
q.g() → Это класс B
```

Строка

```
B.metaClass.fi = p.&f
```

вставляет метод ***f*** из класса ***A*** в класс ***B***, где этому методу присвоено имя ***fi*** (в принципе, имя можно и не изменять). Знак ***&*** - уже знакомый нам ссылочный оператор.

В модуле ***groovy.transform*** имеется набор неких объектов, называемых аннотациями (annotations), таких, как ***Canonical***, ***Delegate, Immutable, Lazy, Newify*** и другие. С помощью этих аннотаций можно добавлять классам дополнительную функциональность. Посмотрим на пример использования аннотации ***Immutable***, её надо указать перед объявлением класса со служебным знаком **@** впереди:

```
import groovy.transform.*
@Immutable
class Person {
   String name
```

```
    int age
}
def p = new Person('Люда', 27)
print p.name → Люда
```

Мы уже знаем, что в классе нужен или явный конструктор, или при создании экземпляра использовать именованные параметры:

```
def p = new Person(name: 'Люда', age: 27)
```

Применение ***Immutable*** позволяет передавать параметры без конструктора, просто перечисляя их в указанном порядке. На самом деле конструктор создаётся аннотацией ***Immutable***. Итак, использование аннотаций создаёт дополнительные удобства в различных случаях.

Отметим ещё, что при импорте отдельной аннотации, её имя надо указывать без предваряющего знака **@**:

```
import groovy.transform.Immutable
```

Можно также не писать отдельную строку импорта, а вызывать аннотацию прямо:

```
@groovy.transform.Immutable
```

Аннотация ***Category*** принимает в качестве аргумента тип объекта, передаваемого классу, и позволяет использовать методы класса напрямую, без создания экземпляра класса. Переданный классу объект доступен, как ***this.*** Например:

```
@Category(List)
class A  {
   def f() {
      def x = this
      x.each {print "${it**3} "}
   }
}
use(A) {[1, 2, 3, 4, 5].f()} → 1 8 27 64 125
```

Метод класса вызывается с использования директивы ***use.*** При этом аннотация не мешает использовать класс, как обычно:

```
def p = new A()
p.f([1, 2, 3, 4, 5]) → 1 8 27 64 125
```

Допустим и такой вариант этой аннотации:

```
@Category(Object)
```

В этом случае классу можно передавать объекты любого типа.

Инициировать поля экземпляра класса можно и после создания объекта с помощью известного уже нам метода ***with***:

```
class Person  {
   String name
   int age
   String address
}
```

def p = new Person() - (создали объект без инициации полей)

```
p.with  {
   name = 'Иван Иванов'
   age = 45
   address = 'Москва'
}
```

println p.name → Иван Иванов

Метод ***properties*** позволяет прочитать все свойства (поля) объекта:

```
p.properties.each {println it} →
        class=class Person
        address=Москва
        age=45
        name=Иван Иванов
```

К методам класса можно обращаться динамически, задавая нужное имя метода на этапе runtime. Рассмотрим такой пример:

```
class A  {
   def f(x) { println "f(x) = ${x**2}" }
   def f(x, y) { println "f(x, y) = ${x + y}" }
}
def p = new A()
def ff(ob, r, t) { ob."$r"(t) }
def ff(ob, r, t, s) { ob."$r"(t, s) }
ff(p, "f", 5 ) → 25
ff(p, "f", 3, 9) → 12
```

Вспомогательная функция ***ff*** принимает экземпляр класса (***p***), имя нужного метода и параметры этого метода. При передаче имени метода применяется строка, которая раскрывается с помощью интерполяции (как в операторе print) в теле вспомогательной функции. Мы использовали одно и то же имя для двух методов класса (***f***) и для двух вспомогательных функций (***ff***). При этом нужный

вариант выбирается по количеству передаваемых параметров. Конечно, это не обязательно и все имена могли быть и разными.

…..5.8 Трейты (trait)

Как и многие другие современные языки, Groovy позволяет использовать трейты, с помощью которых можно изменять функциональность классов, без изменения их кода. Внешне трейты похожи на классы:

```
trait Prob  {
   def f(x) { println "Работает trait: $x**2 = ${x**2}" }
   def s = 'Hello'
}
```

Трейт ***Prob*** содержит метод ***f*** и свойство ***s***. Трейты можно встраивать (можно, наверное, также говорить и наследовать) в классы с помощью ключевого слова ***implements*** например:

```
class A implements Prob  {
   def g() { println "Здесь можно что-то делать" }
}
def p = new A()
p.g()  → Здесь можно что-то делать
p.f(5)  → Работает trait: 5**2 = 25
print p.s  → Hello
```

В классе А можно использовать методы и свойства трейта ***Prob***. Практически можно считать, что код трейта просто вставлен в тело класса.

Объявленное в трейте поле можно непосредственно использовать в конструкторе класса:

```
trait Tr  {
   String name
}
class Person implements Tr {
   Person(name) {this.name = name}
   def f() { println name }
}
def p = new Person('Anna')
p.f()  → Anna
```

Или в конструкторе неявной формы:

```
trait Tr  {
```

```
    String name
}
class Person implements Tr {
    def f() { println name }
}
def r = new Person(name: 'Peter')
r.f() → Peter
```

Один трейт может расширять функциональность другого трейта. Для этого используется слово ***extends***:

```
trait Tr1 { String name }
trait Tr2 extends Tr1 {
    String f() {
        "Hello, $name"
    }
}
class A implements Tr2 {
    def g() { print f() }
}
def p = new A(name: 'Alice')
p.g() → Hello, Alice
```

Абстрактный трейт ***Tr1*** содержит только объявление поля ***name***. Трейт ***Tr2*** добавляет метод, формирующий строку приветствия. Этот метод использован в классе ***A*** для вывода приветствия. Обратите внимание на то, что в классе ***A*** подмешивается трейт ***Tr2***, а не ***Tr1***.

Трейты можно подмешивать не только в классы, их можно включать динамически на этапе runtime прямо в экземпляры классов (в объекты). Одновременно можно подмешивать несколько трейтов с помощью ключевого слова ***withTraits*** (не случайно тут множественная форма).

```
trait Tr1 {
    String f() { "Привет!" }
}
trait Tr2 {
    Double g(x) { x }
}
class A {def fi() {println 'Hello'} }
def p = new A()
def q = p.withTraits Tr1, Tr2
```

println q.f() → Привет!
println q.g(2.7) → 2.7
p.fi() → Hello
q.fi() → Hello

Два трейта включены в экземпляр ***p*** класса ***A*** (у нас этот класс имеет только один метод ***fi***). При таком включении создаётся новый объект ***q***. При этом метод класса ***fi*** доступен как в объекте ***p***, так и в объекте ***q***.

Несколько трейтов сразу можно включать и в классы:

```
trait Tr1 {
   String f() { "Привет!" }
}
trait Tr2 {
   Double g(x) { x }
}
class A implements Tr1, Tr2 { }
def p = new A()
println p.f() → Привет!
print p.g(2.7) → 2.7
```

Но в этом варианте динамика утрачивается.

Кроме трейтов Groovy позволяет использовать интерфейсы, подобные тем, что в Java. Всё, в общем, как обычно.

Есть смысл остановиться на этом этапе. Тот, кто освоил язык в изложенном здесь объёме, легко овладеет и другими возможностями Groovy. Отметим в заключение, что мир Groovy огромен и предоставляет любителю много дорог для прогулок в любом направлении.