Программирование на Fantom
для любопытных

Оглавление

Краткое введение

Язык программирования Fantom относится к типу объектно — ориентированных (ООП), со строгой статической системой типов. В Fantom всё является объектом и даже базовые типы — экземпляры своих классов. Язык интерпретируемый, реализован на виртуальной машине Java. Fantom принадлежит, таким образом, группе языков, созданных на базе Java, таких, как Scala, Groovy, Clojure и, соответственно, обладает достоинствами и недостатками, характерными для этой группы. При этом Fantom без сомнения имеет целый ряд оригинальных свойств и несомненно заслуживает внимания. Ценность его была бы ещё выше, если бы он не был построен на Java, а имел бы в своей основе Си подобные языки и был бы компилируемым.

Программа на Fantom записывается в файл с расширением ***fan***, например, ***prob.fan***. Для выполнения надо перейти в место расположения файла и выполнить команду:

fan prob.fan

Имеется также интерактивный интерпретатор (Repl), для вызова которого применяется команда ***fansh***, после которой появится приглашение и можно вводить строчки кода, которые будут немедленно выполняться с выдачей результата. Этот Repl, правда, имеет довольно ограниченные возможности.

Почти все примеры я оформил в виде законченных программ, так, чтобы их можно было скопировать и выполнить из командной строки или в IDE (Geany, F4).

1.Основы

1.1 Структура программ

В Fantom принята оригинальная система управления составными частями программы при помощи программных модулей, называемых подами (***Pod***). Фантом-программы включают три основных структурных элемента:

Pod – представляет вершину пространства имён

Type – единицы системы объектно-ориентированных типов, классы, короче говоря
Slot - поля и методы

Эти три сущности организуют дерево уровней пространства имён и их квалифицированные имена имеют такой синтаксис:
pod
pod::type
pod::type::slot
Квалифицированные имена для краткости называются в Fantom qname.

В некотором смысле Pod подобен пакету в Java и играет примерно ту же роль, что и Jar-файлы. Его имя глобально и должно быть уникальным. Имеется ряд команд для работы с подами, например:
Pod.list – список установленных подов.
Pod.find("name") – поиск нужного пода, если такового нет — генерируется исключение
Pod.find("name", false) — то же, что и в предыдущем примере, но вместо исключения возвращается значение ***null***.
MyPod.file('/img/icon.png') – поиск файла в поде ***myPod***, и так далее.

Типы (types) должны иметь уникальное имя внутри своего пода. Например, ***sys::Str*** – это qname для типа ***Str***, принадлежащего поду ***sys***. В поде ***sys*** расположены основные базовые структуры и в этом смысле он является корневым.

Соответственно, имеются и команды для выполнения операций над типами и слотами; с ними мы познакомимся по ходу дела.

1.2 Вместо helloword

Впоследствии мы ещё вернёмся к этим подам и разберёмся с их созданием и использованием. Но для первоначального ознакомления с языком можно пока обойтись без них, поскольку Fantom позволяет выполнять программы, записанные в файл как обычно. Точнее, при таком подходе некоторый под компилятор создаёт автоматически. Для иллюстрации рассмотрим простейшую программу, которая всё-таки несколько содержательнее банальной helloworld. На этом примере познакомимся с некоторыми существенными особенностями языка Fantom:

```
class Boo  {
      static Int add(Int a, Int b){return a + b}
```

```
        Int incr() {return ++c}
        Int c := 0
}
class Prob  {
        Void main()  {
                b := Boo()
                x := b.incr()
                y := Boo.add(3, 4)
                echo("x = ${x}, y = ${y}")  → x = 1, y = 7
                d := Foo()
                echo("${d.more}")  → 8
        }
}
class Foo : Boo {
  Int more() { return incr() + add(3, 4) }
}
```

Как видим, весь код состоит из одних только классов (могли быть ещё и миксины, с которыми познакомимся позже). При выполнении программы в интерпретаторе будет отыскан класс, содержащий функцию ***main***, определяющую точку входа в программу. Конкретное название этого класса в данном случае не имеет значения, хотя оно и обязано присутствовать. Часто тут применяют имя ***Main*** (то-есть, такое же, как и у головной функции ***main***, определяющей точку входа в программу) для улучшения читабельности.

Теперь разберём последовательно, что делает эта функция ***main*** в нашем примере построчно:

b := Boo()

- создаёт экземпляр (объект) ***b*** класса ***Boo***, который не имеет параметров, но скобки должны присутствовать.

x := b.incr()

- вызывает метод ***incr*** объекта ***b***, возвращающий значение локальной переменной ***c***, увеличенной на ***1***. Первоначально ***c*** была инициирована нулём в строке ***Int c := 0*** класса ***Boo***. Операции, подобные этой, выполняются в момент создания экземпляра класса. Попутно можем отметить, что тип переменной указывается перед её идентификатором. Тот факт, что для переменной ***x*** тип не указан (хотя можно было бы указать), говорит о том что Fantom умеет выводить тип из контекста и, значит, указывать тип не всегда обязательно.

Однако, для ***c*** указать тип необходимо, так как ***c*** является полем класса. Обязательно указывается и тип результата метода класса:
Int incr() {return ++c}
Если функция ничего не возвращает, как в нашем примере ***main***, то её тип будет ***Void***.
y := Boo.add(3, 4)
- вызывает метод ***add*** с передачей ему аргументов. Этот метод объявлен с модификатором ***static***, а такие методы могут быть вызваны только с указанием имени класса; нельзя написать ***y := b.add(3, 4)***, статические методы принадлежат только классу, а не его экземплярам.
echo("x = ${x}, y = ${y}") → ***x = 1, y = 7***
- выводит значения ***x*** и ***y*** на экран. Здесь и всюду далее стрелкой (→) я буду заменять слова вроде «получим», «будет выведено» и тому подобное. (Не знаю, зачем оператор вывода назван редко используемым словом ***echo***, почти во всех языках программирования применяется print). При копировании кода для выполнения надо эти стрелки и текст за ними убрать. В аргументах у оператора ***echo*** использована интерполяция — знак ***$*** и выражение в фигурных скобках. Значит, Fantom выполняет интерполяцию также, как и большинство современных языков (например, Ruby). Впрочем, интерполяцию можно было бы и не применять, переписав строку так:
echo("x = " + x + ", y = " + y) → ***x = 1, y = 7***
Значит, оператор ***echo*** сам переводит числа в строковое представление и знак ***(+)*** является знаком конкатенации строк; фактически это та же интерполяция.
d := Foo()
- создаёт экземпляр ***d*** класса ***Fo***о. Этот класс отличается от ***Boo*** тем, что он дочерний по отношению к ***Boo*** и операция наследования обозначается двоеточием.

Метод ***more*** класса ***Foo*** использует методы ***incr*** и ***add*** суперкласса ***Boo***, как свои собственные, без указания каких-нибудь дополнительных ссылок.
echo("${d.more}")
- выводит результат, возвращаемый методом ***more***. Обратите внимание, что этот результат равен ***8***, а не ***9***, как можно было бы ожидать, поскольку метод ***incr*** вызывался дважды. Значит, при создании экземпляра ***d*** переменная ***c*** обнуляется снова.

Как видим, методы возвращают результат, указанный в операторе ***return***, и значит, к сожалению, не возвращают последнего вычисленного значения в методах и блоках, как это делается в большинстве современных языков.

Отметим ещё, что для инициации переменных применяется знак присваивания ***(:=)***, при этом тип указывается явно или выводится из контекста и в дальнейшем этот тип нельзя изменить. Можно только изменить значение переменной с использованием знака **(=)**. Нельзя также объявить переменную с тем же идентификатором заново в той же области видимости.

Итак, из этого простого примера мы узнали кое-какие особенности языка Fantom и можем сделать вывод, что за исключением деталей в основном всё также, как и в других языках ООП, например, в Ruby. Но дальше мы ещё встретимся с очень важными особенностями Fantom, которые, собственно, и представляют для нас главный интерес.

1.3 Система типов

Fantom имеет традиционный набор стандартных типов. Все они представляют соответствующие классы в поде ***sys***. Нет необходимости давать им подробную характеристику, лучше с теми немногими особенностями, что характерны только для Fantom, познакомиться на примерах по ходу дела. Пока я ограничусь одним только перечислением базовых типов (не всех):

Bool, Int, Float, Decimal, Str, Num, Duration, Uri, Type, Slot, Range, List, Map

На самом деле эти типы представляют классы в поде ***sys***, точно также, как и создаваемые нами классы представляют типы в соответствующих подах. Фактически, по этой причине термины «класс» и «тип» в Fantom означают одно и то же. Есть два встроенных метода, позволяющих узнать тип переменной: функция ***Type.of*** и метод ***typeof*** (вообще-то это не одно и тоже для некоторых ситуаций). Например:

```
s := "Hello"
Type.of(s) → sys::Str
```

Значит, переменная ***s*** принадлежит классу (является его экземпляром) ***Str*** в поде ***sys***. Выражение ***sys::Str*** это qname для класса ***Str***. Кстати,

для этого выражения есть синоним ***Str#*** (применяется в некоторых особых случаях); если мы его напечатаем в Repl, то получим:
Str# → sys::Str. Это же справедливо и для других классов в поде ***sys***.
b := true
b.typeof → sys::Bool

Попробуем инициировать переменную ***t*** таким образом:
t := s.typeof
А теперь посмотрим, какой тип (класс) имеет переменная ***t***:
t.typeof → sys::Type
Значит, даже значение типа является представителем класса (типа), и имя этого класса — ***Type***.

Есть простые функции для преобразования из типа ***Str*** в другой, например:
x := "22"
y := Int(x)
y.typeof → sys::Int
z := Float(x)
z.typeof → sys::Float
Для преобразования других типов применяются методы ***toInt, toFloat, toStr*** и так далее:
3.84f.toInt → 3 **-** преобразование без округления
Если заданное преобразование невозможно (например, из ***Int*** в ***Bool***), получим сообщение об ошибке.

Итак, классы в Fantom представляют типы. Имеется ещё и вторая разновидность типов - миксины (mixins); с ними мы познакомимся позже.

1.4 Коллекции

Среди перечисленных выше базовых типов три из них: ***Range, List*** и ***Map*** представляют коллекции. Есть смысл привести здесь хотя бы краткое описание этих типов, поскольку они будут использоваться в примерах.

Ранги

Тип ***Range*** представляет непрерывную последовательность целых чисел от начального значения до конечного (ранг). Для объявления ранга применяется такой синтаксис:
x := 3..7

Переменная ***x*** представляет последовательность ***3, 4, 5, 6, 7***.
x = 3..<7
Теперь ***x*** представляет коллекцию: ***3, 4, 5, 6*** – то-есть конечное значение не входит в ранг. Граничные значения могут быть и отрицательными:
x = -2..<4 → -2, -1, 0, 1, 2, 3

Граничные значения могут быть представлены выражениями:
y := 2*3..<15/2 → 6..<7

Методы ***first*** и ***last*** позволяют извлечь соответственно первый и последний элементы ранга:
x.first → -2
x.last → 3

Приведённый выше синтаксис для рангов есть только синтаксический сахар для такой базовой конструкции:
x = Range.make(-2, 4, true) → -2..<4
x = Range.make(-2, 4, false) → -2..4
Практически все стандартные операторы (например арифметические операции) являются синтаксическим сахаром и все они имеют соответствующие синонимы.

Проверим, наконец, тип переменной ***x***:
x.typeof → sys::Range

Можно, конечно, указывать тип при объявлении ранга:
Range z := 0..5 → 0..5
Только вряд ли это имеет какой-то смысл, элементы ранга всегда имеют тип ***Int***.

Списки

Тип ***List*** содержит пронумерованную (индексированную) последовательность элементов, представленных объектами произвольных типов. Фактически списки не отличаются от коллекций, называемых обычно массивами. Объявляются списки при помощи квадратных скобок:
x := [1, 2, true, "aaa", 2.5f] → [1, 2, true, aaa, 2.5]
Обращаю внимание на то, что значения типа ***Float*** записываются с добавлением буквы ***f*** (или ***F***). Соответственно, значения типа ***Decimal*** записываются с добавлением буквы ***d (D)***.

Если элементы одного типа, этот тип может быть указан перед квадратными скобками:

y := Float[2f, 3.4f, -0.075f] → [2.0, 3.4, -0.075]

Если тип не указан, он будет выведен компилятором. Интересно, какой тип выводится для списка ***x***?

x.typeof → sys::Obj[]

Значит, для смешанных типов выводится тип элементов ***Obj***. Класс ***Obj*** в Fantom является коренным, в том смысле, что ему наследуют все классы, а сам он не имеет суперкласса.

В Fantom имеется специальное значение ***null***, используемое в некоторых специфических случаях. Если попытаться определить тип для ***null***, то компилятор запротестует. Но попробуем создать такой список:

x := ["aaa", "bbb", null, "ccc"]

И теперь определим его тип:

x.typeof → sys::Str?[]

Оказывается, что элементы этого списка имеют тип ***Str?*** Это надо понимать так, что переменные типа ***Str?*** относятся к типу ***Str***, но при этом могут ещё и принимать значение ***null***. Такой тип называют nullable. Другие типы тоже могут быть nullable: ***Int?, Float?, Bool?*** и так далее. Обозначения типов без вопросительного знака не относятся к nullable. Компилятор генерирует ошибку при попытке использовать nullable переменную там, где ожидается non-nullable тип. Дальше мы ещё познакомимся с применением таких типов. Посмотрим на пример ещё одного списка:

y := [23, 1.2f, 7, 0.78d]

Здесь присутствуют элементы типов ***Int, Float, Decimal***

y.typeof → sys::Num[]

Компилятор вывел для элементов этого списка новый для нас тип ***Num***, который, очевидно является общим (суперклассом) для классов ***Int, Float, Decimal***.

Так создаётся пустой список:

s := [,]

Его тип также будет ***sys::Obj[]***, но можно задать и другой тип, например:

Float[] s := [,]

или так:

s := Float[,]

Элементами списка могут быть любые выражения:

x := [1, 2*3, 7f/2] → [1, 6, 3.5]

Для извлечения элемента из списка применяется обычный синтаксис:
x[2] → 3.5
На самом деле это краткая форма для метода ***get***:
x.get(2) → 3.5
Можно извлечь сразу группу элементов (подсписок, срез), используя ранг:
x[0..1] →[1, 6]
Это тоже краткая форма для другого метода — ***getRange***:
x.getRange(0..1) → [1, 6]
Соответственно, можно инициировать элементы списка:
x[2] = 9 → [1, 6, 9]
Или, применив соответствующий метод ***set***:
x.set(2, 9) → [1, 6, 9]
Допустимо использовать отрицательные индексы:
x[-2] → 6
x[-2..-1] → [6, 3.5]

Элементами списков могут быть и другие коллекции, в частности списки и в этом случае получим двумерный массив (матрицу).

Метод ***toList*** позволяет преобразовывать ранги в списки:
x := 0..5
y := x.toList → [0, 1, 2, 3, 4, 5]

Перечислим основные встроенные методы для работы со списками с краткими примерами.

Размер списка метод ***size***:
s := [3, 7, 2, 9, 4, 8]
s.size → 6

Проверка, является ли список пустым ***isEmpty***:
s.isEmpty → false

Метод ***dup*** позволяет получить копию списка:
r := s.dup

Проверка присутствия в списке элементов с заданным значением ***contains***:
s.contains(9) → true
s.contains(11) → false

Проверка присутствия в списке группы элементов с заданными значениями ***containsAl***l:
s.containsAll([8, 3, 9]) → true
s.containsAll([8, 3, 9, 11]) → false

Здесь элементы задаются в форме списка, не обязательно их компактное расположение и не имеет значения порядок расположения.

Проверка присутствия в списке хотя бы одного элемента из заданной группы ***containsAny***:

s.containsAny([28, 3, 19, 11]) → true

Определение индекса для заданного элемента списка ***index***:

s.index(9) → 3

У метода ***index*** может быть второй параметр, указывающий начиная с какого элемента надо начинать поиск:

s := ["alpha", 8, "beta", "gamma", 5]

s.index("gamma", 2) → 3

s.index(8, 2) → null - когда поиск неудачен, метод возвращает ***nul***l; здесь нет элемента со значением ***8*** после элемента с индексом ***2***.

Определение индекса для заданного элемента при просмотре списка в обратном направлении ***indexr***:

s.indexr(8, 2) → 1

Метод ***first*** возвращает первый элемент списка, а метод ***last*** - последний:

s.first → alpha

s.last → 5

У метода ***last*** есть синоним — метод ***peek.***

Имеется, кроме того, метод ***pop***, который тоже возвращает последний элемент списка, но при этом он ещё и изменяет сам список, удаляя из него последний элемент:

s.pop → 5

s → [alpha, 8, beta, gamma]

Обычно метод **pop** применяют при использовании списков в роли стека (stack), но это особая тема.

Замена элемента с заданным индексом новым значением выполняется простым присваиванием:

s***[2] = 23.76f → [alpha, 8, 23.76, gamma]***

Значит, при этом можно изменить и тип элемента. Такое присваивание — просто синтаксический сахар для метода ***set***:

s.set(2, "Tanja") → [alpha, 8, Tanja, gamma]

Добавление элемента в конец списка ***add***:

s.add(15) → [alpha, 8, beta, gamma, 5, 15]

Для него также есть синтаксический сахар — оператор (**,**):

s{55,66,77} → [alpha, 8, beta, gamma, 5, 15, 55, 66, 77] - при этом используется it-блок.

Кроме того, есть также метод ***push***, применяемый для добавления элемента в конец списка, когда список используется, как стек:

s = ["alpha", 8, "beta", "gamma"]

s.push(12.3f) → [alpha, 8, beta, gamma, 12.3]

Добавление группы элементов в конец списка ***addAll***:

s.addAll([1, 2, 3]) → [alpha, 8, beta, gamma, 12.3, 1, 2, 3]

Вставка элемента по указанному индексу ***insert***:

s := [1, 2, 3, 4]

s.insert(2, 11) → [1, 2, 11, 3, 4]

Вставка группы элементов по указанному индексу ***insertAll***:

s.insertAll(3, [22, 33]) → [1, 2, 11, 22, 33, 3, 4]

Удаление заданного элемента по значению, при сравнении используется оперция (==), ***remove***:

s.remove(22) → 22 - возвращает удаляемый элемент

s → [1, 2, 11, 33, 3, 4]

Удаление заданного элемента по значению, при сравнении используется операция (===), ***removeSame:***

s***.removeSame(33) → 33***

s → [1, 2, 11, 3, 4]

Напоминаю, чем отличаются эти операции сравнения на примере:

s := [3, 7, 1]

r := [3, 7, 1]

s == r → true

s === r → false - ***s*** и ***r*** занимают не одно место в памяти

Удаление элемента по индексу ***removeAt***:

s := [1, 2, 3, 4, 5]

s.removeAt(2) → 3

s → [1, 2, 4, 5]

Удаление группы элементов по рангу индексов ***removeRange***:

s.removeRange(1..3) → [1]

Удаление группы элементов по списку элементов (по значениям) ***removeAll***:

s = [8, 3, 6, 1, 4, 9]

s.removeAll([4, 1, 3, 25]) → [8, 6, 9]

Значит, если в списке удаляемых элементов есть не существующие, они игнорируются.

Удаление всех элементов из списка ***clear***:
s.clear → [,] - возвращается пустой список

Оптимизация памяти, выделенной под список ***trim***:
s.trim() → [8, 3, 6, 1, 4, 9] - возвращается this

Добавление группы одинаковых элементов ***fill***:
s.fill(5, 3) → [8, 3, 6, 1, 4, 9, 5, 5, 5]

Перестановка двух элементов с указанными индексами ***swap***:
s := ["Java", "Fantom", "Scala", "Clojure"]
s.swap(1, 3) → [Java, Clojure, Scala, Fantom]

Метод ***moveTo*** позволяет переместить заданный (по значению) элемент на место, заданное индексом:
s.moveTo("Scala", 1) → [Java, Scala, Clojure, Fantom]
Можно использовать и отрицательные индексы:
s.moveTo("Scala", -1) → [Java, Clojure, Fantom, Scala]

Метод ***flatten*** раскрывает вложенные списки любой глубины вложенности:
s ***:= [1,[2,[3]],4]***
s.flatten → [1, 2, 3, 4]

Метод ***random*** выбирает из списка элемент случайным образом:
s := ["Java", "Fantom", "Scala", "Clojure"]
s.random() → Scala
s.random() → Fantom

Метод ***shuffle*** возвращает заданный список со случайным порядком расположения элементов:
s.shuffle() → [Scala, Fantom, Java, Clojure]
s.shuffle() → [Clojure, Scala, Fantom, Java]

Метод ***join*** превращает список с элементами любого типа в строку с заданным разделителем между «словами»:
r := s.join("<>") → Clojure<>Scala<>Fantom<>Java
r.typeof → sys::Str
Если разделитель не задан, возвращается одно слово:
[1, 2, 3, 4, 5].join → 12345

Метод ***ro*** делает список readable (только для чтения), а метод ***isRO*** проверяет, является ли список readable:
s := [1, 2, 3]
s.isRO() → false
r := s.ro()
r.isRO() → true

Соответственно, метод ***rw*** делает список read-write (для чтения и для записи), а метод ***isRW*** проверяет, является ли список read-write:
p := r.rw()
p.isRW() → true

В целом можно констатировать, что списки в Fantom - весьма универсальный вид коллекций.

Maps

Эта вид коллекций называют ещё отображением, хэшем (hash), словарём. Map содержит множество не упорядоченных пар ключ-значение. Познакомимся с синтаксисом на примерах:
m := [Int:Str][1:"one", 2:"two"] → [1:one, 2:two]
Map ***m*** содержит два элемента из пар, в которых ключи имеют тип ***Int***, а значения — ***Str***. Скобки при указании типов можно опускать:
m = Int:Str[1:"one", 2:"two"] - будет то же самое
Как всегда, можно опустить указание типов, которые тогда будут выведены из контекста:
m1 := ["Anna":19, "Bob":25, "Peter":34] → [Bob:25, Peter:34, Anna:19]
Как видим, порядок элементов в Map не фиксируется.
m1.typeof → [sys::Str:sys::Int]

Типы ключей и значений могут быть произвольными:
m := [25:"Moscov", "Bob":25, "Peter":34]
m.typeof → [sys::Obj:sys::Obj]
Значит, правила вывода типов те же, что и у списков. Создадим пустой Map:
m1 := Int:Str[:] → [:]
Или без указания типа и тогда будет тип ***Obj:Obj***:
m1 := [:]
Также могут применяться и nullable типы.

Значения Map извлекаются также, как элементы списка, только роль индекса выполняют ключи:
m["Peter"] → 34
Это тоже краткая форма для метода ***get***:
m.get("Bob") → 25
Используя ключ, можно менять значения:
m["Bob"] = 43 → [Bob:43, 25:Moscov, Peter:34]
Или, применив метод ***set***:

m.set("Bob", 28) → [Bob:28, 25:Moscov, Peter:34]

Перечислим основные встроенные методы работы с Map.

Следующие методы полностью идентичны соответствующим методам для списков, только вместо индексов используются ключи: ***size, isEmpty, get, set, dup, setAll, remove, clear.***

Метод ***containsKey*** проверяет присутствие в Map заданного ключа:

m := [1:"Fantom", 2:"Scala", 3:"Groovy", 4:"Go"]
m.containsKey(2) → true
m.containsKey(7) → false

Метод ***keys*** позволяет получить список всех ключей Map:

m.keys → [1, 2, 3, 4]

Соответственно, метод ***vals*** позволяет получить список всех значений map:

m.vals → [Fantom, Scala, Groovy, Go]

Метод ***add*** добавляет пару ключ-значение в Map:

m.add(5, "Ruby") → [1:Fantom, 2:Scala, 3:Groovy, 4:Go, 5:Ruby]

Метод ***getOrAdd*** возвращает значение по заданному ключу, а если такого ключа нет в Map, добавляет новую пару. Детали посмотрим на примере:

m := [1:"Fantom", 2:"Scala", 3:"Groovy", 4:"Go"]
m.getOrAdd(3, |Int k→Str|{"Clojure"}) → Groovy
m.getOrAdd(5, |Int k→Str|{"Clojure"}) → Clojure
m → [1:Fantom, 2:Scala, 3:Groovy, 4:Go, 5:Clojure]

Метод ***setList*** позволяет дополнять в Map члены, в которых значения заданы в списке, а ключи вычисляются в блоке. Посмотрим на такой пример:

m := [1:"Fantom", 2:"Scala", 3:"Go"]
s := ["aaa", "bbb"]
m.setList(s)|Str x->Int|{ s.index(x)+3 } → [1:Fantom, 2:Scala, 3:aaa, 4:bbb]

Здесь в качестве ключей добавляемых пар использованы индексы заданного списка, увеличенные на ***3***. Когда ключ совпадает с уже имеющимся в исходном Map, происходит замена пары (вместо ***3:Go*** стало ***3:aaa***).

Метод ***addList*** аналогичен методу ***setList***, но совпадение ключей не допустимо, компилятор выдаст ошибку:

m := [1:"Fantom", 2:"Scala", 3:"Go"]
s := ["aaa", "bbb"]

m.addList(s)|Str x->Int|{ s.index(x)+4 } → [1:Fantom, 2:Scala, 3:Go, 4:aaa, 5:bbb]

Метод ***toStr*** преобразует Map в строку:

s:= m.toStr() → [1:Fantom, 2:Scala, 3:Go]

Внешне ***s*** не отличается от Map, но это строка:

s.typeof → sys::Str

Метод ***join*** также преобразует Map в строку, с добавлением заданного разделителя:

s1 := m.join("|") → 1: Fantom|2: Scala|3: Go

s1.typeof → sys::Str

1.5 Строки

Строки (тип Str) рассматриваются в Fantom, как списки и, следовательно, к строкам применимы большинство методов для обработки списков:

s := "Language"

s.size → 8

Но есть, конечно, и методы, предназначенные только для работы со строками, или такие же методы, как для списков, но с некоторыми особенностями. Рассмотрим на примерах некоторые из них.

Метод ***all*** (итератор) позволяет проверить, все ли знаки в тексте удовлетворяют заданному условию:

"Bar".all |t| { t.isUpper } → false

"BAR".all |t| { t.isUpper } → true

Можно не вводить локальную переменную (***t***), а использовать в closure встроенную ***it***:

"BAR".all { it.isUpper } → true

Метод ***any*** проверяет, есть ли в строке хотя бы один знак, удовлетворяющий заданному условию:

"Bar".any |t| { t.isUpper } → true

"bar".any |t| { t.isUpper } → false

Метод ***capitalize*** переводит первый знак строки в верхний регистр, а метод ***decapitalize*** выполняет обратное действие:

"bar".capitalize → Bar

"BAR".decapitalize → bAR

Методы ***lower*** и ***upper*** переводят все буквы в строке в нижний и верхний регистры, соответственно:

"Apple".lower → apple

"Apple".upper → APPLE

Метод ***chars*** преобразует строку в массив байтов:

"hello".chars → [104, 101, 108, 108, 111]

Метод ***fromChars*** выполняет обратное преобразование:

Str.fromChars([108,101,111]) → leo

Метод ***compare*** позволяет сравнивать строки попарно:

"beta".compare("gamma") → -1

"gamma".compare("beta") → 1

"gamma".compare("gamma") → 0

Метод ***contains*** проверяет наличие в строке заданной подстроки:

"language".contains("age") → true

Метод ***mult*** позволяет создавать повторяющийся текст:

"Go".mult(3) → GoGoGo

Можно использовать знак умножения:

"Go" * 3 → GoGoGo

Метод ***padl*** добавляет в начало строки повторяющиеся символы, дополняя строку до заданного размера:

"hello".padl(7, 'a') → aahello

Метод ***padr*** делает то же самое, но добавляет в конец строки:

"hello".padr(7, 'a') → helloaa

Метод ***plus*** выполняет конкатенацию строк:

"aaa".plus("bbb") → aaabbb

Можно использовать оператор (+) (уже много раз применяли):

"aaa" + "bbb" → aaabbb

Метод ***replace*** позволяет заменять в строке группу знаков:

"hello".replace("el", "aaa") → haaalo

Итераторы работают со строками, как со списками и передают в блок байты (в десятичном представлении):

"abc".each { echo(it) } → 97 98 99

Если для обработки в блоке нужны буквы, надо применять метод ***toChar***:

"abc".each { echo(it.toChar) } → a b c

Метод ***endsWith*** проверяет, оканчивается ли строка заданным набором знаков:

"fantom".endsWith("tom") → true

"fantom".endsWith("abc") → false

Соответственно, есть и метод ***startsWith***.

Метод ***justr*** добавляет пробелы в начало строки, увеличивая её до заданного размера:
"xyz".justr(7) → xyz
Есть также метод ***justl***, который добавляет пробелы в конец строки.
А метод ***trim*** удаляет пробелы и управляющие символы в начале и в конце строки:
" Hello\n\t".trim → Hello
Методы ***trimStart*** и ***trimEnd*** делают то же самое только для начала и только для конца строки, соответственно.

Метод ***split*** позволяет строки превращать в списки, разбивая строку по указанным знакам:
"alpha beta gamma".split(' ') → [alpha, beta, gamma]
"alpha beta gamma".split('a') → [, lph, bet, g, mm,]
Значит, сами заданные знаки при этом удаляются.

Метод ***splitLines*** превращает многострочный текст в список с элементами, соответствующими отдельным строкам:
"Ruby Scala\nGroovy\nFantom".splitLines → [Ruby Scala, Groovy, Fantom]

Этот перечень не исчерпывает весь список методов для работы со строками. Все эти методы позволяют обрабатывать тексты с использованием регулярных выражений — ничего нового тут Fantom не предлагает. В частности, имеется метод ***toRegex***, позволяющий заданную строку трансформировать в регулярное выражение.

2. Классы и миксины

2.1 Объявление классов

В первом примере мы уже немного познакомились с техникой создания класса и его экземпляров (объектов). Займёмся теперь деталями.

Конструкторы

Классы ***Boo*** и ***Foo*** не имели параметров. Однако, классы во всех языках ООП могут принимать параметры, и классы Fantom не исключение. Обычно для этой цели используют специальные функции, называемые конструкторами. Конструкторы в Fantom довольно специфичны. Посмотрим на примере:

```
class Foo  {
        Int x
        new fi(Int t) { this.x = t}
        Int f() { return x * x }
}
class Main  {
        Void main()  {
                p := Foo.fi(3)
                echo(p.f()) → 9
        }
}
```

Строка ***new fi(Int t) { this.x = t}*** называется конструктором. Функция ***fi***, принимает аргумент, передаваемый классу при создании его экземпляра. Идентификатор функции ***fi*** обычно называют именем конструктора. Выражение ***this.x*** означает, что переменная ***x*** является переменной экземпляра класса в общепринятой терминологии. Эта переменная видна в теле класса и может использоваться там без ограничения, но только не может быть инициирована другим значением. Функция ***fi*** передаёт значение параметра класса ***t*** переменной ***x*** для дальнейшего использования. Слово ***new*** в конструкторе является своего рода маркером. Кроме того, переменная экземпляра ***x*** предварительно должны быть объявлена, автоматически её тип не выводится из контекста.

Функция ***f()*** иллюстрирует использование переменной экземпляра класса, своих аргументов она не имеет.

Выражение ***p := Foo.fi(3)*** создаёт экземпляр класса с передачей аргумента в конструктор. Функцию ***fi*** (конструктор) компилятор способен использовать по умолчанию и выражение можно подсократить:

p := Foo(3)

Как видим, параметр ***t*** служит только для передачи значения в переменную экземпляра класса ***x*** и больше нигде не используется. Поэтому, нет смысла вводить для него свой идентификатор, целесообразнее использовать то же обозначение, что и для переменной экземпляра, не забывая при этом, что ***x*** и ***this.x*** – это не одно и то же и дважды повторённое ***Int x*** объявляет две разные переменные:

```
class Foo  {
```

```
        Int x
        new fi(Int x) { this.x = x}
        Int f() { return x * x }
}
class Main  {
        Void main()  {
                p := Foo(3)
                echo(p.f()) → 9
        }
}
```

Конструктор позволяет создавать классы с произвольным числом параметров:

```
class Foo  {
        Int x; Int y
        new csi(Int x, Int y) { this.x = x; this.y = y}
        Int f() { return x + y }
}
class Main  {
        Void main()  {
                p := Foo(3, 7)
                echo(p.f()) → 10
        }
}
```

Теперь мы дали новое имя конструктору — ***csi***; кажется, что оно нигде далее не используется, но это не так — компилятор использует его по умолчанию.

Интересно посмотреть, к какому типу относится созданный в программе экземпляр класса ***p***. Если добавить функции ***main*** строку ***echo(p.typeof)*** , то получим такой результат:

prob3_0::Foo

То-есть, ***p*** имеет тип ***Foo*** в поде ***prob3_0***. Здесь ***prob3*** — название файла, в котором я разместил программу, а ***(_0)*** добавил компилятор. Поскольку мы никаких подов не создавали, то имя для него сформировал компилятор. Однако практически это мало интересно, поды надо создавать самостоятельно и мы ещё рассмотрим это далее.

Есть ещё одна разновидность конструктора:

```
class Foo  {
        static new fi(Str x) { csi(x.toInt) }
```

```
        Int x
        new csi(Int x) { this.x = x}
        Int f() { return x + x }
}
class Main  {
        Void main()  {
                p := Foo("5")
                echo(p.f()) → 10
        }
}
```

К обычному конструктору здесь добавлена строка:
static new fi(Str x) { csi(x.toInt) }
в которой функция ***fi*** принимает аргумент типа ***Str***, и передаёт его конструктору ***csi***, при этом встроенный метод ***toInt*** трансформирует принятое значение к типу ***Int***. Вместо ***toInt*** можно применять другие аналогичные методы: ***toFloat, toBool*** и так далее, получая соответствующие типы. Надо, конечно, при этом изменить тип и в самом конструкторе. Такой приём позволяет вводить разные данные в виде текста и создавать из него переменные нужных типов, что иногда может быть полезным. Кроме маркера ***new*** использован ещё и модификатор ***static*** (без него генерируется ошибка). Разобраться с деталями всей этой конструкции, видимо, не так просто, приходится удовлетвориться только тем, что это работает.

Вообще-то для создания экземпляра в этом случае надо писать:
p := Foo.fi("5")
но синтаксический сахар позволяет опускать ***.fi***. Кроме того, конструктор позволяет вводить аргумент сразу того типа, что нам нужно, то-есть мы можем написать:
p := Foo(5)
и тогда функция ***fi*** просто не будет использована.

При наследовании классов в дочернем классе можно использовать конструктор родительского класса, но для этого надо выполнить «связывание» конструкторов с помощью директивы ***(:)*** и встроенной функции ***super***. Тут опять довольно сложный синтаксис, который лучше всего просто показать на примере:

```
class A  {
        Float x
        new make(Float x) { this.x = x}
```

```
        Float f() { return x + x }
}
class B : A  {
        Float y
        new make(Float y) : super(y) {this.y = y}
        Float g() { return y }
        Float q() { return x }
}
class Main  {
        Void main()  {
                p := B(6.3f)
                echo(p.f()) → 12.6
                echo(p.g()) → 6.3
                echo(p.q()) → 6.3
        }
}
```

Здесь я применил к конструктору имя ***make***, которое часто используется в Fantom для этой цели (это, конечно не имеет значения; не надо только ошибочно считать, что ***make*** – встроенная функция, поскольку слово это очень популярно и на самом деле есть и такая встроенная функция, мы её ещё увидим). Я думаю, что смысл связывания конструкторов понятен, хотя и выглядит довольно громоздким (для Fantom, базирующимся на Java, это вообще характерно). Как видим, в классе ***B*** переменные ***x*** и ***y*** представляют одну и ту же величину и было бы целесообразно использовать один идентификатор ***x***, а не вводить новую переменную ***y***.

В классе можно иметь несколько конструкторов, позволяющих создавать объекты с разным числом и типом параметров. Можно также использовать связывание конструкторов в одном классе с помощью ключевого слова ***this*** вместо ***super***. Например:

```
class Foo  {
  Str? x; Str y
  new fi(Str? x, Str y) {this.x = x; this.y = y}
  new csi(Str y) : this.fi(null, y) {this.y = y}
  Str? f() { return x + y }
  Str g() { return y }
 }
 class Main  {
```

```
    Void main() {
        p := Foo("Hello, ", "World!")
        echo(p.f()) → Hello, World!
        r := Foo("Liberty")
        echo(r.g()) → Liberty
    }
}
```

Здесь мы связываем конструктор ***csi*** с конструктором ***fi*** того же класса, меняя при этом число параметров. Используем ***nullable*** тип ***Str?*** для того, чтобы в конструкторе ***csi*** заменить не нужную нам переменную ***x*** нейтральным значением ***null***. При создании экземпляров ***p*** и ***r*** мы опустили названия конструкторов (хотя могли бы и указать), компилятор выбирает нужный конструктор, ориентируясь только на количество переданных параметров.

При связывании можно изменить не только количество, но и тип параметров конструктора, например:

```
class Foo {
  Str? x; Str y
  new fi(Str? x, Str y) {this.x = x; this.y = y}
  Int z
  new csi(Int z) : this.fi(null, y) {this.z = z}
  Str? f() { return x + y }
  Int g() { return z }
 }
 class Main {
    Void main() {
        p := Foo("Hello, ", "World!")
        echo(p.f()) → Hello, World!
        r := Foo(555)
        echo(r.g()) → 555
    }
}
```

Изменив тип второго параметра конструктора ***fi*** с ***Str*** на ***Int***, необходимо объявить тип переменной ***z*** заново.

Если конструктор класса отсутствует, компилятор генерирует свой конструктор, позволяющий создавать экземпляры класса без параметров.

Модификаторы

Классы Fantom могут иметь следующие модификаторы: ***public, internal, abstract, final, const***. Например, класс может быть объявлен так:

internal abstract class Foo {}

То-есть, у класса ***Foo*** два модификатора.

Модификаторы ***public*** и ***internal*** определяют область видимости: классы ***public*** общедоступны, а классы ***internal*** — только внутри своего пода. Если модификатор отсутствует — класс относится к ***public*** по умолчанию.

Абстрактные классы (***abstrac***t) не имеют возможности создавать экземпляры непосредственно — при вызове конструктора абстрактного класса компилятор выдаст ошибку. Тем не менее, абстрактные классы имеют конструкторы, которые могут использоваться дочерними классами. Абстрактные классы могут иметь (или не иметь) абстрактные методы. При этом класс с абстрактными методами может быть только абстрактным.

Финальные классы (***final***) не могут наследоваться, попытка создания дочернего класса вызовет ошибку.

Константные классы (***const***) являются ***immutable***, однажды созданный экземпляр такого класса не может менять своё состояние. Такие классы могут наследовать только константным классам (или ***sys::Obj***) и наоборот, не константные классы не могут наследовать константным. Многие базовые классы (***Int, Float, Str*** и так далее) являются ***const***, чем, собственно, и обеспечивается статическая типизация Fantom.

Классы ***const*** могут иметь только статические (***static***) поля и, как исключение, ещё несколько видов полей с особыми свойствами.

2.2 Миксины (mixins)

Миксины предназначены для объявления в их теле некоторого количества слотов, то-есть, полей и методов, для повторного их использования с помощью механизма наследования. В принципе, они подобны интерфейсам в других языках, но, конечно, со своими особенностями. Внешне миксины подобны классам и их qname имеет такой же вид:

MyPod :: myMixin

Перечислим свойства, отличающие миксины от классов:

- Миксины неявно всегда абстрактны (abstract)
- Миксины могут наследовать другим миксинам
- Миксины могут иметь только абстрактные и статические поля
- Миксины могут иметь конкретные, абстрактные и статические методы
- Миксины могут иметь статические конструкторы: ***static {}***

Объявляются миксины точно так же, как и классы, только с использование ключевого слова ***mixin***:
mixin MixinName {…}. Модификаторами миксинов могут быть ***public*** и ***internal***. Модификатор ***const*** тоже может применяться, но использовать такие миксины могут только классы типа ***const.***

Наследование миксину синтаксически задаётся тем же оператором двоеточие: ***class A : MixinName***.

2.3 Перечисления (Enums)

Enum – это разновидность класса, предназначенного для хранения некоторого количества конкретных значений и имеющего такие основные характеристики:

- Enum всегда неявно const и final (то-есть, перечисления immutable)
- Enum может наследовать другим enum и миксинам.
- Enum может иметь конструктор типа private
- Enum может иметь поля и методы

Объявляется enum со словами ***enum*** и ***class***. Список значений (в документации его называют *range*) должен быть всегда раньше полей и методов. Компилятор автоматически генерирует два вспомогательных слота:

- ***vals*** – коллекция (список) значений, которым присваиваются индексы
- ***fromStr*** – обеспечивает доступ по имени

Посмотрим на примере. Объявим enum, имеющий только поля:
enum class Color { red, blue, green }
Компилятор автоматически создаст такой класс:

```
class Color : Enum  {
    static const Color red   = make(0, "red")
    static const Color blue  = make(1, "blue")
    static const Color green = make(2, "green")
    static const Color[] vals = [red, blue, green].toImmutable
```

```
    static new fromStr(Str s) { ... }
    private new make(Int ord, Str name) : super(ord, name) {}
}
```

Покажем на примере, какие мы имеем варианты доступа к полям этих enums:

```
enum class Color {
        red, blue, green
        private new make() {}
}
class Main  {
        Void main()  {
                echo(Color.vals[1]) → blue
                echo(Color.red) →red
                p := Color.fromStr("green")
                echo(p) → green
                echo(p.typeof) → prob5_0::Color -
                r := Color("red")
                echo(r) → red
        }
}
```

(имя пода prob5_0)

Значит, поля записываются через запятую и без кавычек, хотя дальше они используются, как строки. В этом варианте конструктор не имеет ни аргументов, ни тела (объявлять его не обязательно), а экземпляр ***r*** представляет только поле, как и экземпляр ***p***, полученный с помощью метода ***fromStr***.

Имеются встроенные Enum. ***Weekday*** представляет семь дней недели. Он имеет поля и методы:

echo(Weekday.vals[0..6]) → [sun, mon, tue, wed, thu, fri, sat]

Методы ***increment*** и ***decrement*** позволяют получить следующий и предыдущий день.

```
d := Weekday("tue")
d.increment → wed
d.decrement → mon
```

Методы ***localeAbbr*** и ***localeFull*** позволяют получить сокращённое и полное название дня.

Enum ***Month*** содержит названия двенадцати месяцев года и имеет те же методы, что и ***Weekday***.

В некоторых ситуациях использование Enum эффективнее, чем применение списка.

3. Слоты (slots) — поля и методы

3.1 Общая характеристика

В ООП принято считать, что поля определяют состояние экземпляров класса, а методы — поведение. И поля и методы Fantom могут иметь следующие модификаторы:
abstract, native, override, static, virtual.
Кроме того, поля могут иметь ещё модификатор ***const***.

Для задания области видимости применяются модификаторы:
public – общедоступные slots
protected – видимы только в своём и в дочерних классах
internal – видимы только в классах своего пода
private – видимы только в своём классе

По умолчанию поля и методы всегда ***public***.

3.2 Методы

Сначала о модификаторах методов. Модификатор ***static*** мы уже применяли — для вызова static-метода не требуется создавать экземпляр класса и можно обращаться прямо к самому классу.
Виртуальные методы (с модификатором ***virtual***) могут быть подменены (перегружены, override) в дочернем классе. Для перегрузки метод должен быть объявлен с модификатором ***override***:

```
class A  {
      virtual Int f(Int x)  { return x *x }
}
class B : A  {
      override Int f(Int y) { return (y.toFloat.exp).toInt }
}
class Main  {
      Void main()  {
            p := A()
            echo(p.f(5))  → 25
            r := B()
            echo(r.f(2))  → 7
```

```
    }
}
```

Здесь метод ***f*** класса ***A*** перегружен в подклассе ***B***. При перегрузке полностью изменён алгоритм метода. При этом нельзя только изменить тип ни аргумента, ни результата. В примере вычисления выполняются для типа ***Float***, но аргумент и возвращаемый результат имеют тот же тип, что и у перегружаемого метода, то-есть, ***Int***. Для изменения типа аргумента и результата использованы методы преобразования типов ***toFloat*** и ***toInt***. Напоминаю, что метод ***toInt*** возвращает целую часть числа без округления.

Абстрактные методы (с модификатором ***abstract***) одновременно являются виртуальными, но они не могут иметь тела (называют методами без реализации). Абстрактный метод может принадлежать только абстрактному классу.

```
abstract class A  {
    abstract Float f()
}
class B : A  {
    Int x
    new make(Int x) { this.x = x }
    override Float f() { return x.toFloat.exp }
}
class Main  {
    Void main()  {
        r := B(2)
        echo(r.f()) → 7.38905609893065
    }
}
```

Как видим, абстрактный метод может иметь только тип результата, который не может быть изменён при перегрузке метода. В теле перегруженного метода можно использовать только переменные экземпляра и, значит, в дочернем классе обязательно должен быть объявлен конструктор.

Методы с модификатором ***native*** реализуются на языках, с которыми Fantom способен взаимодействовать. Обычно это ***Java*** или ***C#***. Native-методы не должны иметь тела, подобно абстрактным

методам. Технику использования средств других языков надо рассматривать отдельно.

В Fantom можно также использовать так называемый ***once*** метод. Внешне он мало отличается от обычного метода:

```
class A  {
        Float x; Float y
       new make(Float x, Float y) { this.x = x; this.y = y }
       once Str f() { return "x = $x, y = $y, x + y = ${x + y}" }
}
class Main  {
        Void main()  {
                p := A(2.7f, 12.5f)
                echo(p.f())  →  x = 2.7, y = 12.5, x + y = 15.2
        }
}
```

Особенность once-метода заключается в том, что после первого вызова этот метод кэширует свой результат и при следующих вызовах вычисления заново не выполняются, а просто извлекаются из памяти. Ясно, что применять его имеет смысл только при больших объёмах обрабатываемой информации или при затратных по времени вычислениях. Иными словами, такие методы применяются в так называемых ленивых алгоритмах.

Имеются ограничения по применению методов ***once***:

- не могут принадлежать константным классам и миксинам
- не могут быть static и abstract
- должны возвращать результат не Void
- не могут иметь параметров

Если при вызове once-метода будет иметь место исключение (ошибка), результат не будет кэширован и при следующем вызове метод будет пытаться выполнить вычисления.

В Fantom есть возможность применять так называемые статические инициализаторы — специальные методы (а также и поля), которые вызываются автоматически в момент создания экземпляра класса. При объявлении таких методов используется модификатор ***static***:

```
class Foo  {
        static const Int a := 10
```

```
        static { echo("1) a=$a b=$b") }
        static const Int b := 20
        static { echo("2) a=$a b=$b") }
        static { a = 30 }
        static { echo("3) a=$a b=$b") }
}
class Main  {
        Void main()  {
                p:= Foo()
        }
}
```

В результате получим:

```
1) a=10 b=0
2) a=10 b=20
3) a=30 b=20
```

Имеются особенности синтаксиса: методы и выражения должны быть в фигурных скобках, поля должны иметь модификатор const (и при этом их значение можно изменять). Все эти строки выполняются в том порядке, как они расположены в классе. По терминологии Ruby инициализаторы можно было бы назвать переменными и методами класса.

Как и в других языках, в Fantom можно применять параметры по умолчанию:

```
class Foo {
        Int add(Int a, Int b, Int c := 0, Int d := 0)  {
                        return a + b + c + d
        }
}
class Main  {
        Void main()  {
                p:= Foo()
                echo(p.add(3, 4, 5, 6)) → 18
                echo(p.add(3, 4, 5)) → 12
                echo(p.add(3, 4)) → 7
        }
}
```

Переменным c и d по умолчанию задано значение 0. При вызове метода ***add*** этим параметрам можно задать другое значение, или оставить по умолчанию. Переменные по умолчанию должны всегда располагаться в конце списка параметров.

Fantom поддерживает свойство, называемое ***covariance***, позволяющее перегружать методы родительского класса так, чтобы они возвращали тип (экземпляр) дочернего класса:

```
abstract class Animal  {
        abstract Animal daddy()
}
class Cat : Animal  {
        Str f() {"Barsic"}
        override Cat daddy() {return this}
}
class Main  {
        Void main()  {
                r := Cat().daddy.f()
                echo(r) →  Barsic
        }
}
```

Перегруженный метод ***daddy*** в классе ***Cat*** возвращает экземпляр класса ***Cat*** с помощью директивы ***this***. Вместо ***{return this}*** можно было бы применить ***{return Cat()}***, что одно и то же. Позже мы ещё применим это covariance на практике.

Есть также возможность указать тип возвращаемого методом результата как ***This*** (на самом деле это маркер, подобный ***Void***). И тогда можно получить то же самое covariance без перегрузки метода и без абстрактного класса:

```
class A  {
        This f() { return this }
        Str fi() {return "Hello"}
}
class B : A  {
        This g() { return this }
}
class Main  {
        Void main()  {
```

```
            p := B.make.f.g
            echo(p.typeof) → prob2_0::B
            echo(p.fi()) → Hello
        }
}
```

Выражение ***B.make.f.g*** возвращает экземпляр класса ***B***. Очевидно, что код стал более лаконичным и читабельным.

Хотя Fantom имеет статическую типизацию, есть возможность использовать и динамическую типизацию, когда проверка типов выполняется на этапе runtime. Для этого при вызовах метода надо вместо точечной нотации вида ***p.f(7)*** применить вызов с использованием стрелки: ***p -> f() = 7***. Например:

```
class A {
        Int f(Int x)  { return x * x }
}
class Main  {
        Void main()  {
            p := A()
            r := p -> f = 7
            echo(r) → 49
        }
}
```

Встречаются ситуации, когда статическая проверка типа фиксирует исключение, а динамическая — позволяет его обойти. Фактически компилятор вместо выражения ***p -> f = 7*** генерирует код, использующий встроенный метод ***trap***:

```
p.trap("f", [7])
```

При этом можно передавать и разное число аргументов согласно следующим соответствиям:

a->x - ***a.trap("x", [,])*** - без аргументов
a->x = b - ***a.trap("x", [b])*** - с одним аргументом
a->x(b) - ***a.trap("x", [b])*** - с одним аргументом, второй способ
a->x(b, c) - ***a.trap("x", [b, c])*** - с двумя аргументами, и так далее

Фокус заключается в том, что метод ***tra***p имеет способность к ***override***.

В Fantom все операторы (например, арифметические операции) на самом деле являются методами, а их краткая запись — это только

синтаксический сахар. Приведём соответствия хотя бы для части операций:

negate(a)	***-a***	unary
increment(a)	++a	unary
decrement(a)	***--a***	unary
a.plus(b)	***a + b***	binary
a.minus(b)	***a - b***	binary
a.mult(b)	***a * b***	binary
a.div(b)	***a / b***	binary
a.mod(b)	***a % b***	binary
a.get(b)	***a[b]***	binary
a.set(b, c)	***a[b] = c***	ternary

a.add(b) ***a { b, }*** - добавление элемента ***b*** в конец списка ***a***

Эти методы и их краткое представление — синонимы, и могут использоваться на равных правах. Однако, именованные операторы в некоторых ситуациях не допускают изменение раз установленного типа параметров, поэтому предпочтение должно отдаваться кратким представлениям. Только метод ***add*** предпочтительнее всегда.

3.3 Поля

В примерах мы уже использовали поля в разных ситуациях и довольно хорошо познакомились с ними. Теперь только осталось получше систематизировать характеристики полей. Начнём с простейшего примера:

```
class Foo  {
        Int x := 9
        Str? y
}
class Main  {
        Void main()  {
                p := Foo()
                echo(p.x)→ 9
                echo(p.y)  → null
                p.y = "Bob"
                echo(p.y)  → Bob
        }
```

```
}
```

Здесь переменная ***y*** относится к nullable типа Str и при объявлении без инициализации принимает значение ***null***. Все поля, объявленные на уровне класса, инициализируются при создании экземпляра класса без использования конструктора. Для доступа к полям применяется оператор — точка, но фактически компилятор автоматически генерирует специальные методы доступа: метод ***get*** по чтению и метод ***set*** по записи. Эти методы можно объявить и явно и они должны располагаться в блоке сразу за объявлением переменной:

```
class Foo
{
    Float x  {
        get { echo("get x"); return &x }
        set { echo("set x"); &x = it }
    }
}
class Main  {
    Void main()  {
        p := Foo()
        echo(p.x) → get x   0.0
        p.x = 77.09f → set x
        echo(p.x) → get x    77.09
    }
}
```

Строки ***echo("get x");*** и ***echo("set x");*** вставлены только для того, чтобы было явно видно, что методы ***get*** и ***se***t действительно вызываются. Не обязательно объявлять оба метода доступа сразу, можно только один, тогда второй будет создан автоматически. Выражение ***&x*** означает ссылку на переменную ***x***, то-есть на самом деле это адрес ***x***. Впрочем, тот факт, что в этих методах используется ссылочный тип пока нас не должен интересовать. В методе ***set*** используется встроенная переменная ***it***, которой автоматически передаётся значение, присваиваемое переменной ***x***. Иногда тут может применяться closure, который тоже использует ***it*** и тогда надо вводить локальную переменную, например, так:

```
set { a := it; 3.times { echo(a) }; &x = a }
```

В тело метода ***se***t вставлена анонимная функция (closure), которая трижды выводит на экран значение, присваиваемое переменной ***x***. Поскольку closure может неявно изменить встроенную переменную ***it***, пришлось применить локальную переменную ***a*** для сохранения вводимого значения.

Отметим также, что при использовании вместо переменной ***x*** ссылки ***&x*** методы доступа ***get*** и ***se***t не применяются. Обращение к переменным через ссылку применяется и для других целей (не только в методах доступа), но это другая тема.

Поля, объявленные с модификатором ***const*** – immutable. Посмотрим на примере, что с такими полями можно делать:

```
class Foo  {
        new make(Int x) { this.x = x }
        const Int x
}
class Main  {
        Void main()  {
                p := Foo(33)
                echo(p.x) → 33
                p = Foo(55)
                echo(p.x) → 55
                //p.x = 77 → если раскомментировать, будет ошибка
        }
}
```

Значит, const-полям можно присваивать новые значения многократно, но только через конструктор. При прямом обращении к константному полю по записи (***p.x = 77***) компилятор зафиксирует ошибку.

Fantom позволяет для доступа к полям класса применять так называемые it-блоки (it-block). Посмотрим на примере, что это такое:

```
class Foo  {
        new make(|This w|? w := null) { if (w != null) w(this) }
        Str x
}
class Main  {
        Void main()  {
                t := Foo { x = "Marta" ; echo(it.x)} → Marta
```

```
            echo(t.x) → Marta
            t { x = "Brain" }
            echo(t.x) → Brain
        }
}
```

С it-блоком применён несколько странный конструктор:

new make(\|This w\|? w := null) { if (w != null) w(this) }

В конструкторе с помощью локальной переменной (***w***) выполняется некоторая проверка и установка и требуется он только для полей типа ***Str***. Для nullable типа ***Str?*** и всех других типов полей, кроме Str, можно этот конструктор убрать; тогда будет использован неявно генерируемый конструктор. Сам it-блок записываются в фигурных скобках и первый раз должен быть передан конструктору, а затем может использоваться для чтения и записи полей (***t { x = "Brain" }*** в нашем примере). Если объявить переменную ***x***, как константную, выражение ***t { x = "Brain" }*** выдаст ошибку. Название it-блока связано с тем, что внутри него доступна встроенная переменная ***it***, а из примера

t := Foo { x = "Marta" ; echo(it.x)} → Marta

видно, что она представляет экземпляр класса. Очевидно, что в it-блоках могут применяться и closure с использованием ***it***.

Константные поля не используют методы доступа ***get*** и ***set***; попытка их объявления вызовет ошибку. Const-поля не могут иметь модификаторы ***abstract*** и ***virtual***. Тем не менее, они могут перегружаться виртуальными методами без параметров.

Статические (***static***) поля могут быть только константными и установка значений для них может быть только в статических инициализаторах. Доступ по чтению через название класса (без создания экземпляра) и в точечной нотации:

```
class Foo  {
        const static Int x := 555
        static const Str s := "Hello"
}
class Main  {
        Void main()  {
                echo(Foo.x) → 555
                echo(Foo.s) → Hello
```

```
            Foo.x = 777 →    ошибка
        }
}
```

Как видим, модификаторы могут стоять в любом порядке.

Для метода доступа по записи ***set*** можно ограничить область видимости (по умолчанию она ***public***), например:

```
class Foo  {
        Int x { protected set }
        Str? s { internal set { &s = it } }
}
class Main  {
        Void main()  {
            p := Foo()
            p.x = 55
            echo(p.x) → 55
            p.s = "Hello"
            echo(p.s) →  Hello
        }
}
```

Для переменной ***x*** метод доступа генерируется автоматически, а для переменной ***s*** мы его объявили явно, использовав it-блок. Кстати, Fantom не позволяет объявить без инициации переменную типа Str. Дело в том, что компилятор в этом случае пытается присвоить ей значение null и выдаёт ошибку — нельзя присвоить null non-nullable переменной. Допустим только вариант: ***Str? s***.

Ограничивать область видимости для метода доступа по чтению ***get*** не имеет смысла, она определяется областью видимости самого поля.

Если ограничить область видимости метода доступа по записи ***set*** модификатором ***private***, то получим поле только для чтения (Readonly). Одновременно можно ограничить область видимости ещё и для самой переменной, например:

```
class Foo   {
        Int x { private set }
        internal Int y { private set }
}
```

Переменную ***y*** можно только читать и только в классах своего пода (***internal***).

Виртуальные поля (объявленные с модификатором ***virtual***) позволяют перегружать методы доступа в дочерних классах:

```
class A  {
        virtual Int x := 0
}
class B : A  {
        override Int x  {
                get { echo("B.x get"); return super.x }
                set { echo("B.x set"); super.x = it }
        }
}
class C : A  {
        override Int x := 77
}
class Main  {
        Void main()  {
                p := B() → B.x set  B.x get
                p.x = 55
                echo(p.x) → 55
                r := C()
                echo(r.x) → 77
        }
}
```

При перегрузке применяется тот же синтаксис, что и при объявлении методов доступа; дополнительно используется только директива ***super*** и оператор ссылки ***&*** не применяется. При этом методы доступа для перегрузки вызываются в момент создания экземпляра класса, что мы можем видеть по выводу «B.x set B.x get». При перегрузке может использоваться только тот же тип поля, что и в суперклассе. Для инициации виртуального поля новым значением в дочернем классе (***C***) достаточно только одной директивы ***override***.

Поля класса могут быть абстрактными (объявлены с модификатором abstract). Прямой доступ к таким полям невозможен ни по записи, ни по чтению. При этом абстрактные поля могут перегружаться в дочерних классах, где они становятся доступными:

```
abstract class A  {
        abstract Int x
}
class B : A   {
        override Int x := 0
}
class Main  {
        Void main()  {
                p := B()
                p.x = 55
                echo(p.x) → 55
        }
}
```

Поле может быть использовано для перегрузки одноимённого с полем метода, не имеющего параметров, например:

```
abstract class Named  {
        abstract Str? name()
}
class Person : Named  {
        override Str? name
}
class Main  {
        Void main()  {
                p := Person()
                p.name = "Brian"
                echo(p.name) → Brian
        }
}
```

Здесь метод ***name*** класса ***Named*** является виртуальным и он возвращает результат типа ***Str?*** (non-nullable тип Str опять приводит к ошибке по той же причине, что и ранее). В дочернем классе ***Person*** метод перегружается к полю ***name***.

Миксины могут иметь только константные статические и абстрактные поля. Миксины с абстрактными полями практически только определяют сигнатуру методов доступа (выполняют типизацию). Доступ к полям миксинов чаще всего применяется в точечной нотации.

Native-поля применяются при включении элементов других языков (Java и C#). Правила использования нативных полей подобны абстрактным полям.

В целом можно констатировать, что методы и поля в Fantom обладают необычайно большим многообразием (пожалуй, даже избыточным).

3.4 Ветвления и циклы

Условный оператор ***if - else*** в Fantom не имеет ничего оригинального:

if (условие) {блок} else {блок} if …

Условие записывается в круглых скобках, блоки могут содержать любое количество выражений. Поскольку Fantom не позволяет автоматически рассматривать значения разных типов в качестве логических, то в условном выражении всегда применяются операторы сравнения. Можно повторять несколько раз ***if – else***, создавая цепочку проверок. Служебное слово ***then*** не применяется. Сам оператор ***if*** выражением не является в том смысле, что он ничего не возвращает и заканчивать блоки директивой ***return*** нельзя.

```
class Main  {
    Void f(Int x, Int y)  {
        s := ""
        if (x > y) { s = "x > y" } else {s = "y <= x"}
        echo(s)
    }
    Void main()  {
        f(5, 2) → x > y
        f(2, 3) → y <= x
    }
}
```

Ещё раз обращаю внимание на то, почему переменная ***s*** инициирована явно пустой строкой - неявная инициализация возможна только для типа ***Str? s***. Можно было бы, конечно, использовать и автоматический вывод типа: ***s := ""***.

Ветвь ***else*** в операторе ***if*** может отсутствовать и тогда, если условие вернёт значение ***false***, строка просто пропускается.

В этом примере мы впервые весь код поместили в один класс ***Main*** (название Main не обязательное, оно может быть любым). В теле функции ***main*** мы вызываем функцию ***f*** на правах метода класса ***Main***. Может также возникнуть вопрос — а нельзя ли объявить функцию ***f*** прямо в теле функции main? Нет, нельзя, Fantom вообще исключает вложенные функции; можно только использовать closure, но о них позже.

Имеется также тернарный условный оператор -краткая форма оператора if. В общем виде его можно представить так:
условие ? Выражение 1 : выражение 2
Если условие даёт ***true***, вычисляется ***выражение 1***, если ***false*** – ***выражение 2***. Например:
3 > 4? "yes" : "no" - оператор возвращает ***no***
Поскольку оператор возвращает значение, его можно использовать в правой части оператора присваивания:

```
Str s := 3 > 4? "yes" : "no"
echo(s) → no
```

Или так:

```
x := 2.76f; y := 0.79f
r := x = y ? (x + y)/2 : "Hello"
echo → Hello
```

Второй вариант интересен тем, что в зависимости от условия переменная ***r*** может получит тип ***Float*** или ***Str*** и тип нельзя задать, он может быть только выведен компилятором.

Допустима и сокращённая форма использующая только nullable типы (пресловутый elvis). Лучше всего просто показать на примере:

```
class Main  {
        Str? f(Str? s)  {
                Str r :=  s ?: "Hello"
                return r
        }
        Void main()  {
                echo(f(null)) →  Hello
                echo(f("aaa")) →  aaa
        }
}
```

Слева от оператора **?:** должна быть nullable переменная и если она равна ***null***, возвращается результат выражения справа. Если же переменная не равна ***null***, возвращается значение самой этой переменной.

Не обязательно тип должен быть только ***Str?***, можно и так:

```
class Main  {
        Int? f(Int? x)  {
                Int r :=  x ?: 555
                return r
        }
        Void main()  {
                echo(f(null)) → 555
                echo(f(777)) → 777
        }
}
```

Имеет Fantom и традиционный оператор <=>:

```
x := 3; y := 7
x <=> y → -1
y <=> x → 1
x <=> x → 0
```

Fantom поддерживает два вида операторов цикла: ***while*** и ***for***, которые тоже вполне типичны. Сначала пример на применение ***while***:

```
class Main  {
        Void main()  {
                Int i := 0
                while (i < 5)  {
                        echo("i = " + i)
                }
        }
}
```

Получим очевидный результат:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

Ясно, что блок оператора ***while*** также может содержать сколько угодно выражений. И также ясно, что невозможен выход из блока по директиве ***return***.

Теперь пример на цикл ***for***:

```
class Main  {
        Void main()  {
                for (i := 0; i < 5; ++i)  {
                        echo("i = " + i)
                }
        }
}
```

Результат будет тем же, что и в предыдущем примере. Теперь мы применили префиксную нотацию ***(++i)***, но замена на ***(i++)*** здесь ничего не меняет, хотя это не всегда так.

Любое из выражений ***(i := 0; i < 5; ++i)*** может быть опущено. Например, если отсутствует условное выражение:

```
for (i := 0; ; ++i)
```

(точка с запятой должна остаться), его значение принимается ***true*** и получаем бесконечный цикл. Если отсутствует оператор изменения переменной цикла:

```
for (i := 0; i < 5; )
```

получим бесконечный цикл с повторяющимся выводом ***i = 0***.

Для экстренного выхода из цикла по какому-нибудь дополнительному условию применяется директива ***break***:

```
class Main  {
        Void main()  {
                for (i := 0; i < 5; ++i)  {
                        if (i == 3) break
                        echo("i = " + i)
                }
        }
}
```

Теперь получим:

```
i = 0
i = 1
i = 2
```

Директива ***continue*** возвращает управление на начало цикла:

```
class Main  {
        Void main()  {
                Int i
                for ( i = 0; i < 5; i++)  {
                        if (i == 3) continue
                        echo("i = " + i)
                }
        }
}
```

Получим:

i = 0
i = 1
i = 2
i = 4

Когда ***i*** равно ***3*** выполнен переход на начало цикла и это значение не было выведено.

Если циклы вложенные, обе эти директивы ***break*** и ***continue*** имеют отношение к самому внутреннему циклу, а внешний цикл будет работать в обычном порядке.

Основным средством организации циклов в Fantom, как и в большинстве современных языков, являются итераторы. Приведу пример на использование итератора ***each***:

```
class Main  {
        Void main()  {
                s := [1, 2, 3, 4, 5]
                s.each{echo(it * it)}
        }
}
```

Здесь итератор ***each*** принимает список ***s*** и организует цикл, в котором выполняет приданный итератору блок (или closure) в котором встроенная переменная ***it*** последовательно принимает значения, равные элементам списка. В результате получим:

1
4
9
16
25

Позже мы ещё рассмотрим другие итераторы

Поддерживает Fantom и инструмент сопоставления с образцом ***switch-case*** с обычным синтаксисом:

```
class Main  {
	Str f(Str[] s, Int i)  {
		switch (s[i])    {
			case "Sat" :
			case "Sun" :
				Str a := "it's the weekend baby!"; return a
			default:
				return "back to work!"
		}
	}
	Void main()  {
		Str[] week := ["Mon", "Tu", "Wed", "Thu", "Fri", "Sat",
"Sun"]
		echo(f(week, 3)) →  back to work
		echo(f(week, 6)) →it's the weekend baby
	}
}
```

Если ***case*** возвращает ***true***, выполняется соответствующий блок кода. При этом блок этот не заключается в фигурные скобки. Несколько подряд расположенных ***case*** управляют одним и тем же блоком (в примере это ***case "Sat" :*** и ***case "Sun" :***). Как только сопоставление даст ***true*** и выполнится блок, дальнейшие проверки не выполняются. Если ни одно сопоставление не имеет успеха, выполняется блок директивы ***default:***.

Более эффектно сопоставление с образцом работает при использовании ***enum***. Рассмотрим на том же примере:

```
enum class Week  {
	Mon, Tu, Wed, Thu, Fri, Sat, Sun
}
class Main  {
	Str f(Week s)  {
		switch (s)  {
			case Week.Sat :
			case Week.Sun :
```

```
                    Str a := "it's the weekend baby!"; return a
                default:
                    return "back to work!"
            }
        }
        Void main() {
            echo(f(Week.Sun)) → it's the weekend baby
            echo(f(Week.Tu)) → back to work!
        }
}
```

Функции ***f*** здесь передаётся аргумент ***s*** имеющий тип ***Week***.

Иногда в условных выражениях применяется проверка типа переменной. В Fantom это можно делать с помощью методов ***is, isnot*** и ***as***, смысл которых понятен из их названий. Просто приведу примеры:

Str x := "aaaaa"
x is Str → true
x is Float → false
x isnot Int → true
x isnot Str → false

Метод ***as*** позволяет проверять только сопоставимые типы, например тип ***Obj*** сопоставим с любым типом:

Obj x := 123
x as Float → null при отрицательном ответе возвращается ***null***
x as Int → 123 при положительном ответе возвращается проверяемое значение.

4. Функции

4.1 Сигнатура функций

Кроме методов классов Fantom позволяет создавать и выполнять функции («настоящие» функции, если можно так сказать). Больше того, можно считать, что методы — это обёртка функций; методы базируются на функциях. Fantom имеет базовый класс (тип) ***Func***, к которому принадлежат функции. Функции позволяют программирование в функциональном стиле, а также используются в таком интересном средстве, как closure.

Сигнатура определяет типы аргументов и результата функций. Синтаксис сигнатуры покажем на примерах:

|Int a, Int b -> Str|

Эта сигнатура определяет функцию, которая принимает два аргумента типа ***Int*** и возвращает результат типа ***Str***. Использовать функцию можно, например, так:

```
class Main  {
        Void main()  {
                r := |Int x, Int y→Int| {x + y}.call(7, 8)
                echo(r)  →  15
        }
}
```

Выражение ***|Int x, Int y→Int| {x + y}*** является анонимной (безымянной) функцией или closure из класса ***Func***. К такой функции можно применить встроенный метод ***call***, который вызывает функцию и передаёт ей аргументы. Анонимно й функции можно присвоить имя:

```
class Main  {
        Void main()  {
                f := |Int a, Int b->Int| {a + b}
                r := f.call(7, 8)
                echo(r)  →  15
        }
}
```

Явно метод ***call*** можно не указывать, и тогда получим обычный вызов: ***f(7, 8)***. Есть также метод ***callList***, позволяющий передавать параметры в виде списка:

f.callList([7, 8])

Приведём ещё ряд примеров сигнатур для разных функций:

|Int a, Int b -> Str| два параметра типа ***Int*** и результат типа ***Str***

| -> Bool| нет параметров, результат типа ***Bool***

|Str s -> Void| один параметр типа ***Str***, ничего не возвращается

|Str s| то же самое, ***Void*** можно опускать

|->Void| нет параметров, нет результата

|→| то же самое

В примере тип функции ***f - Func*** выводится компилятором, но его можно объявить и явно. Методы ***Type.of*** и ***typeof*** возвращают не тип, а сигнатуру функции, что, конечно, более полезно:

```
class Main() {
    Void main()  {
        Func f := |Int x, Int y->Int|{x + y}
        echo(f(7, 8)) → 15
        echo(f.typeof) → |sys::Int,sys::Int->sys::Int|
    }
}
```

В блоке можно вводить локальные переменные и возвращать результат с помощью директивы ***return***:

```
class Main  {
    Void main()  {
        f := |Float x, Float y -> Str|  {
            r := x + y
            p := r.sin
            return p.toStr
        }
        echo(f(2f, 0.5f)) → 0.5984721441039564
    }
}
```

Функции могут быть вложенными (в примерах выше функции объявлялись в теле функции ***main***).

4.2 Функции и методы

Fantom имеет встроенный метод ***func***, позволяющий методы преобразовывать в функции. Эту способность можно рассматривать как некий мост между ООП и функциональным программированием. Для преобразований применяется своеобразный синтаксис.

Для начала выберем метод с одним аргументом, например, с сигнатурой ***|Float -> Str|***. Для этого годится встроенный метод ***toStr***, который можно применить к величине типа ***Float***, а на выходе получить значение типа ***Str***, например:

```
r := 2.5f.toStr
r.typeof → sys::Str
```

Преобразуем метод ***toStr*** к функции, например, с идентификатором ***ts***. Это можно сделать приблизительно так:

```
class Main  {
	Void main()  {
		ts := Float#toStr.func
		echo(ts(2.5f)) → 2.5
		echo(ts.typeof) → sys::Func
	}
}
```

Итак, для преобразования надо указать тип аргумента (обязательно в форме ***Float#***, не годится указать ***sys::Float***), затем идентификатор метода и применить метод ***func***. В результате получили функцию ***ts***, имеющую тип ***Func***, для вызова которой применим обычный синтаксис: ***ts(x)***.

Точно также можно преобразовать встроенный бинарный метод, например, ***x.plus(y)*** - синоним операции ***x + y***:

```
pl := Int#plus.func
echo(pl(7, 6)) → 13
```

Преобразуем ещё встроенный метод ***replace***, который работает так:

```
"Hello, World!".replace("Hello", "Hi") → Hi, World!
```

Назовём полученную функцию ***rp***:

```
rp := Str#replace.func
s := rp("Hello, World!", "Hello", "Hi")
echo(s) → Hi, World!
```

Некоторые особенности имеют место при преобразованиях пользовательских методов, например:

```
class A  {
	static Int pw(Int a, Int b) { return a.pow(b) }
}
class Main  {
	Void main()  {
		f := A#pw.func
		echo(f(2, 3)) → 8
		echo(f.typeof) → sys::Func
	}
}
```

Здесь мы трансформировали к функции метод ***pw***, который возводит в степень целое число, объявленный в классе ***A***. Если метод с модификатором ***static***, то в качестве типа метода надо указать класс (по-прежнему с синтаксисом ***A#***). В остальном всё также, как и для встроенных методов. Имеется дополнительный нюанс, если метод не имеет модификатора:

```
class A  {
        Int pw(Int a, Int b) { return a.pow(b) }
}
class Main  {
        Void main()  {
                p := A()
                f := A#pw.func
                echo(f(p, 2, 3)) → 8
        }
}
```

В этом случае надо создать экземпляр класса (***p := A()***) и передать его в качестве первого аргумента полученной функции. В остальном всё остаётся таким же. Затем можно объявить новую функцию, понизив арность функции ***f***:

```
fi := |Int x, Int y-> Int|{f(p, x, y)}
echo(fi(2, 3)) → 8
```

Или, объединив всё в одном выражении окончательно получим:

```
class A  {
        Int pw(Int a, Int b) { return a.pow(b) }
}
class Main  {
        Void main()  {
                p := A()
                f := |Int x, Int y-> Int|{ (A#pw.func)(p, x, y) }
                echo(f(2, 3)) → 8
                echo(f.typeof) → |sys::Int,sys::Int->sys::Int|
        }
}
```

Теперь ***f.typeof*** возвращает сигнатуру функции.

Отметим ещё, что методы с модификатором ***static*** при преобразовании дают ***immutable*** функции, а не статические — нет.

4.3 Анонимные (безымянные) функции (closures)

Слово closure переводится на русский, как замыкание. Перевод приблизительный, и термин не отражает всей сложности этого понятия. Для краткости я буду пользоваться английским вариантом, что мне больше нравится. Closure применяется в большинстве современных языков, и это весьма эффективное и интересное средство.

В примерах мы уже неоднократно использовали closure, в частности, при создании функций. Рассмотрим теперь эту тему более детально.

В общем виде синтаксис closure можно представить так:

сигнатура { блок }

Здесь блок представляет тело closure, например:

| Int x, Int y -> Int |{ return x * y }

Ранее мы уже видели, что closure можно использовать, как значение и если этим значением инициировать переменную, то она получит тип ***Func*** и будет представлять функцию:

```
class Main  {
        Void main()  {
                f := | Int x, Int y -> Int |{ return x * y }
                echo(f(3, 7)) → 21
        }
}
```

Посмотрим теперь на пример, иллюстрирующий ещё одно важное свойство closure:

```
class Main  {
        Void main()  {
                x := 0
                f := | -> Int |{ return x++ }
                echo(f()) → 0
                echo(f()) → 1
                echo(f()) → 2
                echo(x) → 3   - почему здесь 3?
        }
}
```

Согласно сигнатуре, функция ***f*** не имеет аргументов и возвращает целое число. Важным здесь является то, что переменная ***x*** объявлена во внешней по отношению к функции области и доступна в блоке closure, кроме того она сохраняет значение, так что при повторных вызовах мы получаем результаты, увеличивающиеся на единицу. Кстати, если применить префиксный инкремент ***++x***, то в результате получим числа ***1 2 3 3***. Считаю, что читатель сам должен разобраться, почему это так.

Итак, из этих двух примеров видим, что closure может рассматриваться, как значение. А если это так, то значит можно создать функцию, которая возвращает closure в качестве результата. Например:

```
class Main  {
        Func g() {
                return  |->| { return echo("Hello, world!") }
        }
        Void main() {
                f := g()
                echo(f.typeof) → |->sys::Void|
                f() → Hello, world!
        }
}
```

Функция ***g*** (или метод, поскольку точно так же мы раньше создавали методы) возвращает результат типа ***Func***, то-есть closure. Этот closure не имеет аргументов и ничего не возвращает (только выводит текст). Кстати, в блоке этого closure можно опустить директиву return: ***{ echo("Hello, world!") } -*** это можно делать тогда, клгда в блоке только одно выражение. В операторе присваивания

f := g()

мы инициируем переменную ***f*** результатом вызова функции ***g***, то-есть closure. Таким образом, мы получаем функцию ***f*** и выражение ***f.typeof*** возвращает её сигнатуру (мы уже ранее видели это). Выражение ***f()*** вызывает эту функцию без аргументов.

Необходимо сделать несколько замечаний относительно синтаксиса в этом примере. Во-первых, сигнатура |→| должна записываться только так, без пробелов — нельзя написать | → |, хотя в других случаях такие пробелы допустимы (не знаю, почему это

так). Иногда пустые скобки при отсутствии параметров у функции можно опускать и в этом примере можно было бы написать ***f := g***. Однако, при вызове функции ***f()*** пустые скобки опустить нельзя. Мне кажется, чтобы не делать лишних ошибок, лучше принять правило - пустые скобки при вызове функций без аргументов ставить всегда. Нельзя также написать ***f() := g()*** - функцию нельзя инициировать. Наконец, выражение ***echo(f)*** также выводит сигнатуру функции ***f()***.

Пожалуй, самое интересное и полезное качество closures – это возможность передавать их функциям в качестве аргумента. Рассмотрим пример на эту тему:

```
class A  {
        Func g1 := |Int x, Int y -> Int| { return x + y }
        Func g2 := |Float x, Float y -> Str| { return (x * y).toStr}
}
class Main  {
        Obj f(Num x, Num y, Func g) { return g(x, y) }
        Void main()  {
                p := A()
                r1 := f(2, 3, p.g1)
                r2 := f(2.1f, 3.4f, p.g2)
                echo(r1)→ 5
                echo(r2) → 7.14
        }
}
```

Здесь в классе A мы создали два closure с разными сигнатурами: ***g1*** принимает два целых числа и возвращает целое число, а ***g2*** принимает два числа типа ***Float*** и возвращает результат типа ***Str***. Теперь объявляем в классе ***Main*** функцию ***f***, принимающую два аргумента типа ***Num*** и один аргумент типа ***Func***. Тип ***Num*** является общим для типов ***Int*** и ***Float*** и переменная типа ***Num*** может принимать как значения типа ***Int***, так и значения типа ***Float***. Ну, а переменная типа ***Func*** может принимать closure. Возвращает функция ***f*** результат типа ***Obj***, который, как мы уже знаем, может иметь значения любых типов, а значит ***Int*** и ***Str*** в частности. Вычисляя переменные ***r1*** и ***r2***, мы передаём функции ***f*** два числа и соответствующий closure, получая нужные результаты.

Таким образом, мы создали функцию ***f***, обладающую широкими возможностями и действуя подобным образом, мы можем создавать очень гибкие и универсальные функции, максимально приспособленные для самых разных ситуаций.

Fantom имеет большой набор методов, которые могут принимать closure в качестве аргументов и к ним, в частности, относятся такие мощные методы, как итераторы. Например, итератор ***findAll*** применяется к списку и принимает в качестве аргумента closure. Этот closure должен иметь один параметр (в примере далее этот параметр я обозначил ***t***), которому в цикле передаются элементы списка. Кроме того, closure должен иметь условное выражение и возвращать значение типа ***Bool***, зависящее от параметра. В результате итератор отбирает из списка те элементы, для которых closure возвращает значение ***true***, и объединяет их в новом списке. Применим итератор ***findAll*** для отбора из списка чётных и нечётных чисел:

```
class A  {
        Func g1 := |Int t->Bool| { return t % 2 == 0 }
        Func g2 := |Int t->Bool| { return t % 2 != 0 }
}
class Main  {
        Void main() {
                p := A()
                s := [1, 2, 3, 4, 5, 6, 7]
                x := s.findAll(p.g1)
                y := s.findAll(p.g2)
                echo(x) → [2, 4, 6]
                echo(y) → [1, 3, 5, 7]
        }
}
```

В классе ***A*** мы создали два closure: ***g1*** возвращает значение ***true*** для чётных чисел, а ***g2*** – для нечётных.

Кроме того, на правах синтаксического сахара Fantom позволяет передавать closure итератору без ввода промежуточного идентификатора, просто записывая closure сразу после имени итератора. Именно такая форма итераторов практикуется в Ruby. В этом случае код нашего примера будет более кратким:

```
class Main  {
```

```
    Void main() {
        s := [1, 2, 3, 4, 5, 6, 7]
        x := s.findAll() |Int t->Bool| { return t % 2 == 0 }
        echo(x) → [2, 4, 6]
    }
}
```

Наконец, и это ещё не всё. Можно не писать сигнатуру и не вводить для итератора параметр самостоятельно, а использовать автоматический вывод типов и встроенную переменную ***it*** и тогда вместо closure будет простой it – блок, без директивы ***return***:

```
class Main  {
    Void main() {
        s := [1, 2, 3, 4, 5, 6, 7]
        x := s.findAll() { it % 2 == 0 }
        echo(x) → [2, 4, 6]
    }
}
```

Это практически полная копия кода на языке Ruby, за исключением того, что там не требуется создание класса и метода main, определяющего точку входа.

Отметим ещё, что и пустые скобки у метода ***findAll*** не обязательны: ***x := s.findAll { it % 2 == 0 }***.

Вообще-то Fantom имеет встроенные методы ***isEven*** и ***isOdd***, проверяющие числа на чётность:

```
8.isEven → true
```

Значит, в примере лучше писать так:

```
x := s.findAll() { it.isEven }
```

Кроме итератора ***findAll*** есть итератор ***find***, который выполняет поиск в списке только до первого элемента, удовлетворяющего заданному условию. Цикл останавливается и найденный элемент возвращается, как результат:

```
s.find { it.isEven } → 2
```

Имеется несколько встроенных методов, позволяющих определить некоторые характеристики функций. Для примера создадим какую-нибудь функцию с помощью closure, например:

```
f := |Int x, Int y->Str|{return (x+y).toStr}
```

Метод ***returns*** позволяет узнать тип возвращаемого функцией результата:

f.returns → sys::Str

Метод ***params*** позволяет узнать, какие параметры принимает функция:

f.params → [sys::Int x, sys::Int y]

4.4 Ещё немного об итераторах

При работе итераторов со списками есть дополнительные возможности. Покажем их на примере уже знакомого нам итератора ***each***.

При работе со списком итератор ***each*** может передавать в свой блок (closure) как значения элементов, так и индексы:

```
class Main  {
    Void main() {
        s := ["one", "two", "three"]
        s.each |Str x, Int i  -> Void| { echo("$i = $x") }
    }
}
```

Получим:

```
0 = one
1 = two
2 = three
```

Значит, указав сигнатуру, можем задать, что именно надо передать в блок. Можно при этом использовать и автоматический вывод типов, а также не указывать тип результата:

```
s.each | x, i | { echo("$i = $x") }
```

Не обязательно передавать оба параметра, если указать только один, передаваться будут значения:

s.each | x | { echo(x) } → one two three

При использовании сигнатуры, переменная по умолчанию ***it*** становится не доступной:

s.each | x | { echo(it) } - будет ошибка

Итератор ***each*** работает и с рангом, только теперь передать в блок индексы нельзя:

(2..5).each | x | { echo(x) } → 2 3 4 5

Имеется разновидность итератора ***each — eachr***, который обрабатывает список в обратном порядке:

```
class Main  {
        Void main() {
                s := ["one", "two", "three"]
                s.eachr | x, i | { echo("$i = $x") }
        }
}
```

Получим:

```
2 = three
1 = two
0 = one
```

Перечислим основные встроенные итераторы с краткими примерами.

Итератор, обрабатывающий только часть элементов списка, заданную с помощью ранга индексов ***eachRange***:

```
s := [5, 2, 9, 4, 7, 3, 1]
s.eachRange(2..4){ echo(it +it) } → 18 8 14
```

Поиск в списке первого элемента, не равного null - ***eachWhile***:

```
s := [null, null, 12, 5, 87]
s.eachWhile{ it } → 12
```

Этот метод можно применить и для отыскания первого элемента в списке, удовлетворяющего заданному условию:

```
s := [5, 2, 9, 4, 7, 3, 1]
s.eachWhile |x, i| {if (x<7) { return null } else { return [i, x] }} → [2, 9]
```

Здесь мы извлекли первый элемент, который больше *7*, и вернули индекс и значение элемента в форме списка.

Определение индекса для первого элемента в списке, удовлетворяющего заданному условию ***findIndex*** :

```
s.findIndex { it == 7 } → 4
```

Извлечение элементов из списка по заданному типу ***findType***:

```
s := [22, "Marta", true, 1.02f, "Peter"]
s.findType(Str#) → [Marta, Peter]
```

Тип должен быть указан только в такой форме ***Str#*** (нельзя sys::Str). Соответственно, и для ***Int#***, ***Float#*** и так далее.

Извлечение элементов, для которых блок возвращает false ***exclude***:

```
s := [4, 9, 2, 5, 1, 7]
```

s.exclude { it > 4 } → [4, 2, 1]

Проверка, есть ли в списке элементы, удовлетворяющие заданному условию ***any***:

s.any{ it >= 5 } → true

s.any{ it >= 23 } → false

Проверка, все ли элементы списка удовлетворяют заданному условию ***all***:

s := ["Java", "Fantom", "Scala", "Clojure"]

s.all { it.size > 3 } → true

s.all { it.size < 4 } → false

Итератор ***map*** аналогичен итератору **each**, только он из результатов вычислений в блоке формирует новый список:

s.map { it.size } → [4, 6, 5, 7]

Итератор ***reduce*** позволяет получить результат заданных операций над всеми элементами списка в совокупности: сумма, произведение и тому подобное:

s := [2, 5, 1]

s.reduce(0) |Int r, Int t->Int| { r + t*t } → 30

Этот итератор требует задания полной сигнатуры для closure блока. Параметр итератора ***reduce*** (в примере равен ***0***) присваивается первой переменной, указанной в сигнатуре.

s.reduce(1) |Int r, Int t->Int| { r*t } → 10 - дла произведения параметр итератора должен равняться единице.

Итератор ***min*** позволяет найти наименьший (по заданному критерию) элемент списка. При сравнении элементов используется оператор <=>:

s.min → 1

На самом деле тут выполняется такой код:

s.min |a, b| { a <=> b } → 1

Применим этот итератор для более сложного случая:

s := ["Java", "Fantom", "Scala", "Clojure"]

s.min |Str a, Str b->Int| { a.size <=> b.size } → Java

По критерию обычной сортировки строковых величин:

s.min → Clojure

Точно так же работает итератор ***max***:

s.max → Scala

Итератор ***unique*** удаляет из списка дублирующие элементы:

s := [3, 7, 2, 3, 5, 3, 3]
s.unique() → [3, 7, 2, 5]

Итератор ***union*** объединяет два списка в один при одновременном удалении дублирующих элементов:

s1 := [6, 3, 4, 3]
s.union(s1) → [3, 7, 2, 5, 6, 4]

Итератор ***intersection*** из двух списков создаёт один, в который входят элементы, присутствующие в обоих исходных списках. При этом удаляются дублирующие элементы:

s := [2, 5, 1, 2, 7, 2]
s1 := [3, 2, 9, 2, 5]
s1.intersection(s) → [2, 5]

Итератор ***sort*** позволяет сортировать списки. Без блока элементы сортируются по возрастанию:

s := ["Java", "Fantom", "Scala", "Clojure"]
s.sort → [Clojure, Fantom, Java, Scala]

С помощью блока можно задать критерий сортировки самостоятельно (как для итераторов ***min*** и ***max***):

s.sort |Str a, Str b-> Obj?| { a.size <=> b.size } → [Java, Scala, Fantom, Clojure]

Поменяв местами переменные ***a*** и ***b***, изменим порядок сортировки на обратный:

s.sort |Str a, Str b-> Obj?| { b.size <=> a.size } → [Clojure, Fantom, Scala, Java]

Есть также итератор ***sortr***, сортирующий по убыванию:

s := [4, 8, 1, 9, 3, 2]
s.sortr → [9, 8, 4, 3, 2, 1]

При желании можно и для этого итератора добавить блок.

Итераторы ***sort*** и ***sortr*** при всех модификациях производят сортировку «на месте», то-есть изменяют сортируемый список .

Итератор reverse меняет порядок элементов в списке на обратный:

s.reverse → [1, 2, 3, 4, 8, 9]

Отражения (Map) обрабатываются итератором ***each*** точно также, как списки, только роль индексов выполняют ключи:

```
class Main  {
      Void main()  {
            m := [1:"one", 3:"three", 5:"five"]
```

```
            m.each |Str x, Int t| { echo("$t : $x") }
            m.each |Str x| { echo(x) }
        }
}
```

Результат (для краткости я убираю перевод на новую строку):

1 : one 3 : three 5 : five
one three five

Приведём также перечень основных итераторов для Map. Все они по названиям не отличаются от тех, что применяются для списков. Есть только некоторые особенности их применения.

Например, итератор ***eachWhile***, как и для списков, позволяет найти первый элемент в Map, удовлетворяющий заданному условию. Надо только иметь в виду, что в сигнатуре ***|v, k|*** переменная на первом месте ***(v)*** представляет значение, а не ключ, как можно было ожидать.

m := [1:Fantom, 2:Scala, 3:Go]

m.eachWhile |v, k| {if (k<=2) { return null } else { return [k, v] }} → [3, Go]

Итератор ***find*** позволяет отыскивать элемент в Map, как по заданному значению, так и по заданному ключу:

m.find(|v, k|{ v == "Scala"}) → Scala
m.find(|v, k|{ k > 2}) → Go

В том и другом случае возвращается значение (не ключ).

Итератор ***findAll*** отыскивает все элементы, удовлетворяющие условию (как по значениям, так и по ключам) и результат возвращает в виде Map.

m.findAll(|v, k|{ k > 1}) → [2:Scala, 3:Go]

Нет необходимости приводить примеры использования всех итераторов для Map ввиду их полной идентичности итераторам для списков, далее я их просто перечислю: ***exclude, any, all, reduce, map, isRO, isRW, ro, rw.***

5. Поды, файлы, ввод-вывод

5.1 Поды (pods)

Давайте теперь разберёмся, что же это такое, как поды создавать и использовать. Как всегда лучше всего начать с простейшего примера. Для создания пода надо подготовить пару папок и поместить в них

файлы. В своей рабочей директории (там, где хранятся наши исходные fan-файлы) создадим две папки и два файла:

hello
- build.fan
fan
- main.fan

Файл build.fan содержит информацию (зависимости), необходимую для создания пода. Например, он может выглядеть так:

```
class Build : build::BuildPod  {
        new make()  {
                podName = "hello"
                summary = "hello world pod"
                depends = ["sys 1.0"]
                srcDirs = [`fan/`]
        }
}
```

Первая строчка обязательная — в файле должен быть класс ***Build*** с указанием его суперкласса. Обязателен и метод ***make()*** с маркером ***new***. Далее следуют:

"hello" - имя пода, только это имя должно совпадать с именем папки, в которой он расположен; имена всех остальных папок и файлов могут быть произвольными.

"hello world pod" - это произвольный комментарий, команда ***fanr*** позволяет его вывести на терминал.

["sys 1.0"] - указывает, что будет использоваться базовый под ***sys***, версии ***1.0***. Эта зависимость должна быть всегда и здесь же через запятую могут быть перечислены и другие поды, используемые в нашей программе.

[`fan/`] - указывает папку, в которой расположен исходный текст программы, то-есть ***main.fan***.

Файл ***main.fan*** содержит головную программу. Ещё раз уточняю: имя файла может быть произвольным. Для примера поместим в файл ***main.fan*** простейшую программу вывода ***"Hello world"***:

```
class Main  {
        static Void main() { echo("Hello world") }
}
```

Вся подготовительная работа закончена.

Теперь переходим в папку ***hello*** и компилируем файл ***build.fan*** обычной командой:

fan build.fan

Здесь появится в командной строке информация о результатах компиляции, в частности могут быть сообщения об ошибках, с которыми придётся разбираться. Если всё нормально, то в папке C:/fantom-1.0.69/lib/fan появится файл ***hello.pod***, который и обеспечивает все связи и зависимости. Предназначен этот файл для внутреннего использования Fantom-системой и анализировать его содержание можно после распаковки, как zip-файла. Конечно, не слишком хорошо , что в директории, где содержится библиотека Fantom, будут накапливаться файлы всех подов, которые мы будем создавать. Придётся самому заботиться об удалении тех, что станут не нужны. Можно, правда, изменять константы переменных сред, меняя адреса, но это особая тема.

Запуск программы возможен из любого места любой из следующих команд:

fan hello - по имени пода

fan hello::Main — по полному имени класса Main

fan hello::Main.main - по полному имени метода ***main***.

Итак, под ***hello*** создан и в принципе типы (классы) и слоты (поля и методы) этого пода доступны из других подов. К сожалению, наша программа ничего не содержит, что можно было использовать в других подах. Чтобы посмотреть, как это делается, рассмотрим ещё один пример, в котором создадим два пода. Теперь в рабочей директории создадим четыре папки и четыре файла, ну, хотя бы с такими названиями:

pod1
- pod1.fan
prob1
- foo.fan
pod2
- pod2.fan
prob2
- myrab.fan

Файлы могут иметь, например, такое содержание:

….pod1.fan:

```
class Build : build::BuildPod
{
  new make()
  {
    podName = "pod1"
    summary = "hello pod1"
    depends = ["sys 1.0"]
    srcDirs = [`prob1/`]
  }
}
```

….foo.fan:

```
class Boo  {
        Int x
        new make(Int x){this.x = x}
        Int f() { return x * x }
}
class Main {
        Void main()  {
                p := Boo(5)
                echo(p.f())
        }
}
```

….pod2.fan:

```
class Build : build::BuildPod
{
  new make()
  {
    podName = "pod2"
    summary = "hello pod2"
    depends = ["sys 1.0", "pod1 1.0"]
    srcDirs = [`prob2/`]
  }
}
```

….myrab.fan:

```
class Main  {
        Void main()  {
                p := pod1::Boo(7)
```

```
        echo(p.f())
    }
}
```

Файл ***foo.fan*** содержит вполне законченную программу, в которой есть класс ***Boo*** с конструктором и методом ***f***, возвращающим квадрат целого числа. Эту программу можно использовать самостоятельно без всяких ограничений. Файл ***pod2.fan*** отличается только тем, что в зависимостях мы указали два пода: ***sys*** и ***pod1*** (компилятор требует, чтобы были добавлены номера версий). Тем самым мы определяем, что в поде ***pod2*** будут ссылки на ***pod1***.

Файл ***myrab.fan*** содержит также самостоятельную программу с классом ***Main*** и методом ***main***. Строка:

p := pod1::Boo(7)

создаёт экземпляр ***p*** класса ***Boo*** из пода ***pod1*** с вызовом конструктора и передачей ему аргумента ***7***, а строка:

echo(p.f())

вызывает метод ***f*** экземпляра ***p*** и выводит результат на терминал.

Программа запускается командой ***fan pod2*** или её вариантами с выдачей результата ***49***. Кстати, можно выполнить и команду ***fan pod1*** и тогда получим результат ***25***. Конечно, файл ***foo.fan*** мог и не быть самостоятельной программой, то-есть, не содержать функции ***main***, а иметь только коллекцию классов. Для пода ***pod2*** это безразлично.

Fantom позволяет загрузить любой под с помощью команды ***using mypod***, где ***mypod*** – название пода (без кавычек). Эта команда должна находиться в начале программы, до объявления первого класса. Если команда ***using*** присутствует, к классам пода можно обращаться по краткому имени (без указания имени пода). При этом в программе будут доступны все классы пода. Можно также загрузить и отдельный класс какого-нибудь пода:

using mypod::myclass

С помощью команды ***using*** используются библиотечные модули Fantom.

Имеется несколько команд для дополнительных операций с подами: ***fanr, fanp, fant, jstub***. С помощью опции ***-h*** или ***-?*** можно получить справку по этим командам. Команда ***fanr*** позволяет, в частности, удалить не нужный под:

fanr uninstall pod2
После этого получим текст, записанный в комментарии в файле pod2.fan - ***"hello pod2"***, сообщение о дате создания пода и о занимаемой им памяти, а далее появится вопрос:
Uninstall pod2? [y/n]: позволяющий удалить под, или оставить.

Надо признать, что такая система подов сильно облегчает жизнь по созданию и использованию модулей и по формированию библиотеки; практически все наши программы автоматически становятся библиотечными.

5.2 Обмен информацией с файлами

Запись данных в файл, или вывод из файла на Fantom выполняются очень просто. Сначала надо выполнить такое присваивание:
f := File(`myfile.txt`)
где ***myfile.txt*** – название файла.
Переменная ***f*** получит тип ***sys::LocalFile***. При этом файл должен располагаться в рабочей директории или можно задавать полный путь. Файл может уже существовать, или будет создаваться вновь при записи.

Имеется несколько разновидностей команд записи — чтения. Записать заданный текст в файл можно командой:
f.out.printLine("hello").close
Текст ***"hello"*** будет записан в файл ***myfile.txt***. Если файл не существовал, он будет создан, а если это уже существующий файл, старое содержимое в нём будет уничтожено. Можно, конечно, и так:

```
class Main  {
        static Void main() {
        f := File(`myfile.txt`)
        s := "The Fantom\nProgramming\nLanguag"
        f.out.printLine(s).close
        }
}
```

Будет записан текст, содержащий три строки. Директива ***close*** выталкивает данные из буфера и закрывает файл.

Если после ***out*** добавить аргумент ***(true)*** заданный текст будет добавлен в конец существующего текста:

f.out(true).printLine("world").close

Чтение из файла выполняется так же просто:

r := f.readAllStr

Переменная ***r*** получит тип ***Str*** и будет содержать весь текст файла, как большую строку.

r := f.readAllLines

Теперь переменная ***r*** будет представлять список, элементы которого содержат строки текста из файла.

Имеется также итератор ***eachLine***, позволющий читать из файла строку за строкой и передавать её в блок для обработки:

f.eachLine |x| { echo(x) }

В этом варианте строки будут выводиться на терминал одна за другой. Можно, конечно, применить и it-блок:

f.eachLine { echo(it) }

В блоке над строками можно выполнять любые доступные действия.

Есть ещё разновидность методов для записи в файл и чтения из файла:

f.writeObj("Fantom language")

s := f.readObj

Особенность этих методов заключается в том, что они обрабатывают управляющие последовательности в тексте несколько иначе. При этом файл закрывается автоматически.

Fantom имеет большие возможности по организации потоков, но при первом знакомстве с языком их лучше пока отложить.

5.3 Ввод с клавиатуры, вывод на терминал

Fantom предлагает единственный оператор вывода на терминал — ***echo***. Всё, что умеет делать ***echo***, это автоматическое преобразование переменных к типу ***Str***, выполнение управляющих последовательностей и интерполяцию выражений. Всё это мы уже видели на рассмотренных примерах. Форматированный вывод для оператора ***echo*** авторы Fantom не предусмотрели и для его реализации приходится использовать дополнительные методы. Во-первых, есть метод ***padr***, который применяется так:

echo("aaaa".padr(7))

Для заданного текста отводится ***7*** позиций и текст прижимается к левому краю выделенной области. Есть также метод ***padl***, полностью аналогичный методу ***padr***, но текст в этом случае прижимается к правому краю области.

Второй метод — ***toLocale***, позволяющий выводить числа типа ***Float*** и ***Decimal*** с заданным числом знаков после десятичной точки. У этого метода тоже есть варианты:

echo(1234567.09876543f.toLocale("##,###,###.00000")) →
1 234 567,09877

Эта форма позволяет задавать количество знаков после десятичной точки (в старинном стиле вместо точки выводится запятая) и разделять тысячи пробелом.

echo(67.09876543f.toLocale("000000.00000")) → 000067,09877

В таком варианте задаётся и количество знаков перед десятичной точкой, но лишние позиции заполняются нулями. Как видим, в обоих случаях числа выводятся с округлением.

Эти средства позволяют запрограммировать вывод в нужном виде, например:

```
class Main {
        Void main()  {
                x := 123.45679876f
                s := "x =".padr(7)+(x.toLocale("#.000")).padr(15)
                echo(s+"End")
                x = 123456.987654f
                s = "x =".padr(7)+(x.toLocale("#.000")).padr(15)
                echo(s+"End")
        }
}
```

Получим вполне приемлемый результат:

```
x =    123,457         End
x =    123456,988     End
```

А если вместо ***padr*** применить ***padl***, то получим:

```
x =            123,457End
x =        123456,988End
```

Упоминавшиеся ранее методы ***writeObj*** и ***readObj*** можно также использовать для вывода на терминал и для ввода с клавиатуры. Покажем на примере:

```
class Main  {
        Void main()  {
                Env.cur.out.writeObj("Hello"); echo(" World")
                res := Env.cur.in.readObj
                rr := res.toStr
                rrr := Float(rr) * 3.2f
                echo(rrr)
        }
}
```

В строке ***Env.cur.out.writeObj("Hello");*** использован метод класса ***Env***, который предназначен для организации потоков. В данном варианте происходит вывод на терминал заданного аргумента метода ***writeObj***. Этот аргумент должен иметь тип ***Str*** и при выводе сохраняются кавычки (это может быть ***(")***, ***(""")*** и ***(`)***). Поскольку оператор ***echo*** кавычек не выводит, то в целом результат будет такой: ***"Hello" World***.

Очевидно, что метод ***writeObj*** не выполняет ни интерполяцию, ни управляющие символы и не преобразует типы.

Соответственно, строка ***res := Env.cur.in.readObj*** позволяет ввести текст с клавиатуры и инициировать введённым значением переменную ***res***. На этой строке происходит останов программы и ожидание ввода. Вводить можно любые символы, которые будут восприниматься в кодировке UTF-8. Кодировку можно изменить, но пока не будем вдаваться в такие детали. Переменная ***res*** получит тип ***Obj?***, который можно трансформировать в тип ***Str*** с помощью метода ***toStr***. Ну, а от типа ***Str*** можно затем перейти к любому другому типу, в примере я использовал для этого метод ***Float***. Если ввести с клавиатуры, например, число ***2.73f*** (букву ***f*** вводить не обязательно), то в результате получим ***8.736***. После ввода любого набора знаков, надо нажать клавишу ***Enter***, а затем ***ctr/z*** и два раза ***Enter***.

В целом, можно констатировать, что работе с терминалом и клавиатурой авторы Fantom не уделили много внимания. Зато Fantom имеет мощный графический интерфейс, видимо в основном надо на него и ориентироваться. Кроме того, Fantom предназначен прежде всего для веб-программирования, а там совсем другие возможности ввода-вывода информации.

6. Функциональный стиль

Fantom позволяет создавать чистые (без побочных эффектов) функции и допускает рекурсию. Этого достаточно, чтобы программировать в функциональном стиле. Посмотрим на пример реализации программы вычисления факториала числа в функциональном стиле:

```
class Main  {
        Int f(Int n)  {
                if (n<2) {return 1} else {return n * f(n-1)}
        }
        Void main()  {
                echo(f(5))  → 120
        }
}
```

Как видим, эта программа, с использованием хвостовой рекурсии в чистом виде, имеет ясный код. В принципе, её после несущественных изменений можно реализовать на любом компиляторе, допускающем функциональный стиль (например на Ruby).

Чаще функциональные языки предпочитают использовать технику сопоставления с образцом и оператор ***switch-case*** позволяет это делать на Fantom. Применим этот приём для той же задачи вычисления факториала:

```
class Main  {
        Int f(Int n)  {
                switch(n<2) {
                        case true: return 1
                        default : return n * f(n-1)
                }
        }
        Void main()  {
                echo(f(5))  → 120
        }
}
```

По существу, это то же самое, что и выше.

Вот как эта программа может выглядеть на Haskell:

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

Этот код несколько короче, особенно, если учесть, что объявлять сигнатуру не обязательно.

Пожалуй, главное место функциональные языки программирования отводят работе со списками. Посмотрим, как с этим обстоит дело на Fantom. Запрограммируем для начала вычисление суммы элементов списка:

```
class Main  {
        Int f(Int[] s)  {
                switch(s)  {
                        case Int[,] : return 0
                        default : return s.removeAt(0) + f(s)
                }
        }
        Void main()  {
                s := [1,2,3,4,5]
                echo(f(s)) → 15
        }
}
```

Тут, наверное, нужны некоторые пояснения. Метод ***removeAt*** удаляет из списка элемент с заданным индексом и возвращает удаляемый элемент. Таким образом, если список не пуст, то первый элемент списка складывается с суммой элементов списка без первого элемента.

Приведу ещё пример извлечения элемента с заданным индексом:

```
class Main  {
        Int f(Int[] s, Int n)  {
                switch(s.size-1)  {
                        case n : return s.last
                        default : s.pop; return f(s, n)
                }
        }
        Void main()  {
                s := [3,8,6,1,9]
```

```
            echo(f(s, 2)) → 6
        }
}
```

Если заданный индекс совпадает с индексом последнего элемента, этот элемент возвращается, как результат. Иначе из списка удаляется последний элемент с помощью метода ***pop*** и всё повторяется снова.

В примере вычисления чисел Фиббоначи функция рекурсивно вызывается дважды:

```
class Main   {
    Int fib(Int x)  {
        switch (x)  {
            case 0 : return 0
            case 1 : return 1
            default : return fib(x-1) + fib(x-2)
        }
    }
    Void main()  {
        echo(fib(7)) → 13
    }
}
```

Функциональный стиль предполагает возможность каррирования функций. Fantom позволяет это делать легко и естественно. Функции можно объявлять непосредственно:

```
Int f(Int x, Int y) { x+y }
Int g(Int z) { f(z, 6) }
echo(g(3))  → 9
```

или с помощью closure:

```
Func f := |Int x, Int y-> Int| { x+y }
Func g := |Int z->Int|{ f(z, 5) }
echo(g(2)) → 7
```

В этих примерах из функции двух аргументов ***f*** с помощью простого приёма получаем функцию одного аргумента ***g***. Мне кажется, нет необходимости рассматривать здесь примеры использования каррированных функций; таких примеров много в руководствах по программированию в функциональном стиле.

7. Заключение

Мне кажется, что приведённого материала достаточно для первого знакомства с языком Fantom. Вряд ли на этом этапе целесообразно рассматривать такие темы, как организация потоков и параллелизм, средства веб-проектирования, графический интерфейс. Fantom этими средствами располагает. В частности, для организации потоков и параллельных вычислений применяются такие инструменты, как сериализация и акторы, способные обмениваться сообщениями. Графический интерфейс мало отличается от подобных механизмов в других языках, например в Ruby. Не затронул я и тему программирования исключений, этот механизм полностью подобен соответствующим средствам любого другого языка.

В целом, на мой взгляд, язык Fantom привносит не слишком много новых идей, за исключением только системы подов. Система типов, например, в таком языке, как Scala, намного изощрённее, и с лихвой перекрывает возможности Fantom. Знакомство с этим языком, конечно, полезно, но если допустим, что некто занимается реальным программированием на Ruby, то ему вряд ли есть смысл переходить на Fantom, да пожалуй и на любой другой из группы языков, реализованных на виртуальной машине Java. Лично у меня эта неуклюжая, громоздкая, невероятно многословная машина вызывает стойкую антипатию. Как впрочем и IDE под названием Eclipse, обслуживающий все эти языки. Мне просто противно, когда при реализации программы из одной строчки, вроде helloworld, создаётся некая конструкция, размером в несколько мегабайт.