86,30
Рейтинг

Haulmont
Компания

i_osipov 1 ноября в 20:11

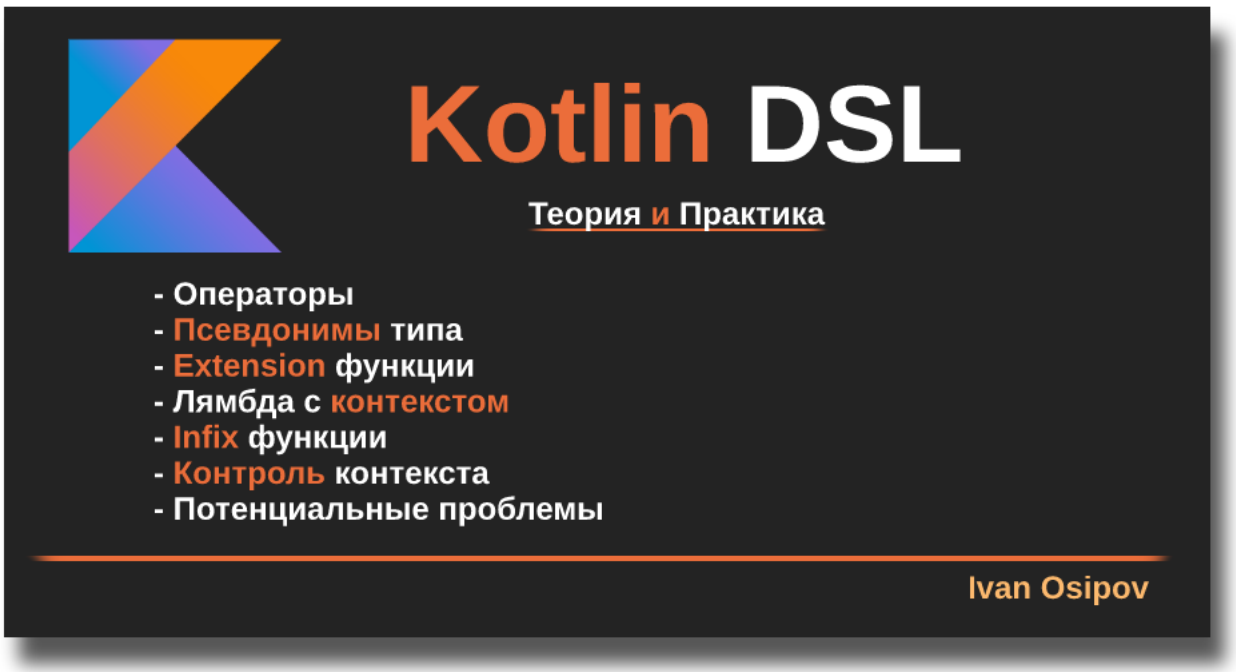
Kotlin DSL: Теория и Практика

Тестирование IT-систем, Программирование, Kotlin, Java, Блог компании Haulmont

Tutorial

Sql, RegExp, Gradle — что их объединяет? Всё это примеры использования проблемно-ориентированных языков или DSL (domain-s language). Каждый такой язык решает свою узконаправленную задачу, например, запрос данных из БД, поиск совпадений в тексте описание процесса сборки приложения. Язык Kotlin предоставляет большое количество возможностей для создания собственного проблемно-ориентированного языка. В ходе статьи мы разберемся, какие инструменты есть в арсенале программиста, и реализуе DSL для предложенной предметной области.

Весь синтаксис, представленный в статье, я объясню максимально просто, однако, материал рассчитан на практикующих инженер которые рассматривают Kotlin, как язык для построения проблемно-ориентированных языков. В конце статьи будут приведены недостатки, к которым нужно быть готовым. Используемый в статье код актуален для Kotlin версии 1.1.4-3 и доступен на GitHub.



Что такое DSL?

Языки программирования можно разделить на 2 типа: универсальные языки (general-purpose programming language) и предметно-ориентированные (domain-specific language). Популярные примеры DSL — это SQL, регулярные выражения, build.gradle. Язык умен объем предоставляемой функциональности, но при этом он способен эффективно решать определенную проблему. Это способ оп программу не в императивном стиле (как нужно получить результат), а в декларативном или близком к декларативному (описать текущую задачу), в таком случае решение проблемы будет получено исходя из заданной информации.

Допустим, у вас есть стандартный процесс выполнения, который иногда может меняться, дорабатываться, но в целом вы хотите использовать его с разными данными и форматом результата. Создавая DSL, вы делаете гибкий инструмент для решения различны задач из одной предметной области, при этом конечный пользоваель вашего DSL не задумывается о том, как решение задачи буд получено. Это некоторое API, виртуозно пользуясь которым, вы можете сильно упростить себе жизнь и долгосрочную поддержку системы.

В статье я рассмотрел построение "внутреннего" DSL на языке Kotlin. Такой вид проблемно-ориентированных языков реализуется основе синтаксиса универсального языка. Подробнее об этом вы можете прочитать по ссылке.

Область применения

Один из лучших способов применить и продемонстрировать Kotlin DSL, на мой взгляд, это тесты.

Предположим, что вы пришли из мира Java. Часто ли вам приходилось снова и снова описывать стандартные экземпляры сущност для довольно крупной модели данных? Вероятно, что для этого вы использовали какие-нибудь билдеры или, еще хуже, специальн утилитные классы, которые под капотом заполняли значения по умолчанию? Как много у вас перегруженных методов? Как часто в нужно «совсем немного» отклониться от значений по умолчанию и как много работы для этого приходится делать сейчас? Если н кроме негатива, у вас эти вопросы не вызывают, то вы читаете правильную статью.

Длительное время на нашем проекте, посвященном образовательной сфере, мы точно так же, с помощью билдеров и утилитных классов, покрывали тестами один из важнейших модулей системы — модуль построения учебного расписания. На смену этому по пришел язык Kotlin и DSL для формирования различных вариантов применения системы планирования и проверки результатов. Ни увидите примеры того, как мы воспользовались возможностями языка и превратили разработку тестов подсистемы планирования пытки в удовольствие.

В ходе статьи мы разберемся в конструкции DSL для тестирования небольшой демонстрационной системы планирования занятий между учеником и преподавателем.

Основные возможности

Давайте перечислим основные преимущества Kotlin, которые позволяют достаточно чисто писать на этом языке и доступны для построения собственного DSL. Ниже представлена таблица с основными улучшениями синтаксиса языка, которые стоит использо Просмотрите этот список внимательно. Если большая часть конструкций для вас не знакома, то желательно читать последовательн Однако если вы не знакомы с одним или двумя пунктами, то можете перейти сразу к ним. Если всё здесь для вас знакомо, то вы м перейти к обзору недостатков использования DSL в конце статьи. Если вы хотите дополнить этот список, то, пожалуйста, напишит свои варианты в комментариях.

Название функциональности	DSL синтаксис	Обычный синтаксис
Переопределение операторов	`collection += element`	`collection.add(element)`
Псевдонимы типа	`typealias Point = Pair<Int, Int>`	Создание пустых классов-наследников прочие костыли
Соглашение для get/set методов	`map["key"] = "value"`	`map.put("key", "value")`
Мульти-декларации	`val (x, y) = Point(0, 0)`	`val p = Point(0, 0); val x = p.first; y = p.second`
Лямбда за скобками	`list.forEach { ... }`	`list.forEach({...})`
Extention функции	`mylist.first(); // метод first() отсутствует в классе коллекции mylist`	Утилитные функции
Infix функции	`1 to "one"`	`1.to("one")`
Лямбда с обработчиком	`Person().apply { name = «John» }`	Нет
Контролирование контекста	`@DslMarker`	Нет

Нашли для себя что-то новое? Тогда продолжим.

В таблице намеренно пропущены делегированные свойства, так как, на мой взгляд, они бесполезны для построения DSL в том вид который мы будем рассматривать. Благодаря указанным возможностям вы сможете писать код чище, избавиться от большого количества "шумного" синтаксиса и при этом сделать разработку еще более приятным занятием ("куда уж приятнее?" — спросите Мне понравилось сравнение из книги Kotlin in Action, в натуральных языках, например, в английском, предложения построены из с грамматические правила управляют тем, как нужно объединять слова друг с другом. Аналогично в DSL, одна операция может быт

сложена из нескольких вызовов методов, а проверка типов обеспечит гарантию, что конструкция имеет смысл. Естественно, поря вызовов может быть не всегда очевиден, но это остается на совести проектировщика DSL.

Важно понимать, что в этой статье мы будем рассматривать «внутренний DSL», т.е. проблемно-ориентированный язык базируетс универсальном языке — Kotlin.

Пример финального результата

Прежде чем мы приступим к построению нашего проблемно-ориентированного языка, я хочу продемонстрировать результат того вы сможете построить после прочтения статьи. Весь код вы можете найти на GitHub репозитории по ссылке. Ниже рассмотрен DS тестирования поиска преподавателя для студентов по интересующим их предметам. В этом примере есть фиксированная времен сетка и мы проверяем, что занятия размещены в плане преподавателя и студента в одно и то же время.

```
schedule {
    data {
        startFrom("08:00")
        subjects("Russian",
                "Literature",
                "Algebra",
                "Geometry")
        student {
            name = "Ivanov"
            subjectIndexes(0, 2)
        }
        student {
            name = "Petrov"
            subjectIndexes(1, 3)
        }
        teacher {
           subjectIndexes(0, 1)
           availability {
             monday("08:00")
             wednesday("09:00", "16:00")
           }
        }
        teacher {
            subjectIndexes(2, 3)
            availability {
                thursday("08:00") + sameDay("11:00") + sameDay("14:00")
            }
        }
        // data { } doesn't be compiled here because there is scope control with
        // @DataContextMarker
    } assertions {
        for ((day, lesson, student, teacher) in scheduledEvents) {
            val teacherSchedule: Schedule = teacher.schedule
            teacherSchedule[day, lesson] shouldNotEqual null
            teacherSchedule[day, lesson]!!.student shouldEqual student
            val studentSchedule = student.schedule
            studentSchedule[day, lesson] shouldNotEqual null
            studentSchedule[day, lesson]!!.teacher shouldEqual teacher
        }
    }
}
```

Инструменты

Полный список инструментов для построения DSL, был приведен выше. Каждый из них был использован в примере и, исследуя ко ссылке, вы можете изучить построение таких конструкций. Мы не раз будем возвращаться к этому примеру для демонстрации различных инструментов. Важно отметить, что решения по построению DSL носят демонстративный характер, хотя вы можете повторить увиденное и в собственном проекте, это не означает, что представленный вариант единственно верный. Ниже мы дета рассмотрим каждый инструмент.

Некоторые возможности языка особенно хороши в совокупности с другими и первый инструмент в этом списке — лямбда вне ско

Лямбда вне скобок

Документация

Лямбда-выражения или лямбды — это блоки кода, которые можно передавать в функции, сохранять или вызывать. В языке Kotlin т лямбды обозначается следующим образом `(список типов параметров) -> возвращаемый тип`. Следуя этому правилу, самый примити вид лямбды это `() -> Unit`, где Unit — это аналог Void с одним исключением. В конце лямбды или функции мы не должны писать конструкцию «return ...». Благодаря этому, мы всегда имеем возвращаемый тип, просто в Kotlin это происходит не

Ниже приведен простейший пример того, как можно сохранить лямбду в переменную:

```
val helloPrint: (String) -> Unit = { println(it) }
```

Для лямбд без параметров компилятор способен самостоятельно вывести тип из уже известных. Однако в нашем случае один пар есть. Вызов такой лямбды выглядит следующим образом:

```
helloPrint("Hello")
```

В примере выше лямбда принимает один параметр. Внутри лямбды этот параметр по умолчанию имеет имя "it", но если параметр несколько, то вы должны явно перечислить их имена, либо использовать знак подчеркивания "_", чтобы проигнорировать его. При ниже демонстрирует такое поведение.

```
val helloPrint: (String, Int) -> Unit = { _, _ -> println("Do nothing") }

helloPrint("Does not matter", 42) //output: Do nothing
```

Базовый инструмент, который вы уже могли встретить, например, в Groovy, это лямбда вне скобок. Обратите внимание на пример самом начале статьи, практически каждое использование фигурных скобок, за исключением стандартных конструкций — это использование лямбд. Существует как минимум два способа сделать конструкцию вида `x { … }`:

- объект x и его унарный оператор invoke (этот способ обсудим позже);
- функция x, в которую передают лямбду.

Независимо от варианта, мы используем лямбды. Допустим, есть функция `x()`. В языке Kotlin действует следующее правило: если лямбда является **последним** аргументом функции, то её можно вынести за скобки, если при этом лямбда **единственный** парамет скобки можно не писать. В результате, конструкция `x({…})` может быть преобразована в `x() {}`, а затем, убрав скобки, мы получае `{}`. Объявление такой функции выглядит следующим образом:

```
fun x( lambda: () -> Unit ) { lambda() }
```

или в сокращенной форме для однострочных функций, вы можете записать так:

```
fun x( lambda: () -> Unit ) = lambda()
```

Но что если x — это экземпляр класса, объект, а не функция? Существует другое интересное решение, которое базируется на одн основополагающих концепций, используемой при построении проблемно-ориентированных языков, переопределение операторов Давайте рассмотрим этот инструмент.

Переопределение операторов

Документация

Kotlin предоставляет широкий, но ограниченный спектр операторов. Модификатор operator позволяет определять функции по соглашениям, которые будут вызываться при определенных условиях. Очевидным примером является функция plus, которая будет выполнена, при использовании оператора "+" между двумя объектами. Полный перечень операторов вы найдете по ссылке выше в документации.

Рассмотрим чуть менее тривиальный оператор "invoke". Главный пример этой статьи начинается с конструкции schedule { }. Назна конструкции — обособить блок кода, который отвечает за тестирование планирования. Для построения такой конструкции используется способ, немного отличающийся от рассмотренного выше: оператор invoke + "лямбда вне скобок". После определени

оператора invoke нам становится доступна конструкция schedule(...), при том, что schedule — это объект. Фактически, вызов schedule(...) интерпретируется компилятором как schedule.invoke(...). Давайте посмотрим на декларацию schedule.

```
object schedule {
    operator fun invoke(init: SchedulingContext.() -> Unit)  {
        SchedulingContext().init()
    }
}
```

Нужно понимать, что идентификатор schedule отсылает нас к единственному экземпляру класса schedule (синглтону), который по специальным ключевым словом object (подробнее о таких объектах, можно прочитать здесь). Таким образом, мы вызываем метод invoke у экземпляра schedule и при этом единственным параметром метода определяем лямбду, которую выносим за скобки. В ит конструкция schedule {... } равносильна следующей:

```
schedule.invoke( { код внутри лямбды } )
```

Однако если вы посмотрите внимательнее на метод invoke, то увидите не обычную лямбду, а "лямбду с обработчиком" или "лямбд контекстом", тип которой записывается следующим образом: `SchedulingContext.() -> Unit`
Пора разобраться с тем, что это такое.

Лямбда с обработчиком

Документация

Kotlin дает нам возможность установить контекст для лямбда-выражения. Контекст — это обычный объект. Тип контекста опреле вместе с типом лямбда-выражения. Такая лямбда приобретает свойства нестатического метода в классе контекста, но с доступо только к публичному API этого класса.
В то время как тип обычной лямбды определяется так: `() -> Unit`, тип лямбды с контекстом типа X определяется так: `X.()-> Unit` если первый тип лямбд можно запускать привычным образом:

```
val x : () -> Unit = {}
x()
```

то для лямбды с контекстом нужен контекст:

```
class MyContext

val x : MyContext.() -> Unit = {}

//x() //не скомпилируется, т.к. не определен контекст

val c = MyContext() //создаем контекст

c.x() //всё работает

x(c) //так тоже можно
```

Напомню, что в объекте schedule у нас определен оператор invoke (см. предыдущий параграф), который позволяет нам использов конструкцию:

```
schedule { }
```

Лямбда, которую мы используем, имеет контекст типа SchedulingContext. В этом классе определен метод data. В результате у нас получается следующая конструкция:

```
schedule {
    data {
        //...
```

```
    }
}
```

Как вы вероятно догадались, метод data принимает лямбду с контекстом, однако, контекст уже другой. Таким образом, мы получа вложенные структуры, внутри которых доступно одновременно несколько контекстов.

Чтобы детально понять как работает этот пример, давайте уберем весь синтаксический сахар:

```
schedule.invoke({
    this.data({
    })
})
```

Как вы видите, всё предельно просто.
Давайте взглянем на реализацию оператора invoke.

```
operator fun invoke(init: SchedulingContext.() -> Unit)  {
    SchedulingContext().init()
}
```

Мы вызываем конструктор для контекста: `SchedulingContext()`, а затем на созданном объекте (контексте) вызываем лямбду с идентификатором init, которую мы передали в качестве параметра. Это очень похоже на вызов обычной функции. В результате, в строке `SchedulingContext().init()` мы создаем контекст и вызываем переданную в оператор лямбду. Если вас интересуют другие примеры, то обратите внимание на методы apply и with из стандартной библиотеки Kotlin.

В последних примерах мы рассмотрели оператор invoke и его взаимодействие с другими инструментами. Далее мы сфокусируем другом инструменте, который формально является оператором и делает наш код чище, а именно на соглашении для get/set метод

Соглашение для get/set методов

Документация

При разработке DSL мы можем реализовывать синтаксис доступа к ассоциативному массиву по одному или более ключам. Взглян пример ниже:

```
availabilityTable[DayOfWeek.MONDAY, 0] = true
println(availabilityTable[DayOfWeek.MONDAY, 0]) //output: true
```

Чтобы использовать квадратные скобки, необходимо реализовать методы get или set в зависимости от того, что нужно (чтение ил запись) с модификатором operator. Пример реализации этого инструмента вы можете найти в классе Matrix на GitHub по ссылке. Э простейшая реализация обертки для работы с матрицами. Ниже часть кода, которая интересует нас.

```
class Matrix(...) {
    private val content: List<MutableList<T>>
    operator fun get(i: Int, j: Int) = content[i][j]
    operator fun set(i: Int, j: Int, value: T) { content[i][j] = value }
}
```

Типы параметров функций get и set ограничены только вашей фантазией. Вы можете использовать как один, так и несколько параметров для get/set функций и обеспечивать комфортный синтаксис для доступа к данным. Операторы в Kotlin привносят мног интересных возможностей, с которыми вы можете ознакомиться в документации.

К удивлению, в стандартной библиотеке Kotlin есть класс Pair, но почему? Большая часть сообщества считает, что класс Pair — это плохо, с ним пропадает смысл связи двух объектов и становится не очевидно, почему они в паре. Следующие два инструмента демонстрируют, как можно и осмысленность пары сохранить, и не создавать лишние классы.

Псевдонимы типа

Документация

Представим, что нам нужен класс-обертка для точки на плоскости с целочисленными координатами. В принципе, нам подходит кл
`Pair<Int, Int>`, но в переменной такого типа в один момент мы можем потерять понимание того, зачем мы связываем значения в
Очевидные пути исправления — это либо писать собственный класс, либо еще, что похуже. В Kotlin арсенал разработчика пополня
псевдонимами типа, которые записываются следующим образом:

```
typealias Point = Pair<Int,Int>
```

Фактически, это обычное переименование конструкции. Благодаря такому подходу, нам не нужно создавать класс Point, который
данном случае просто дублировал бы пару. Теперь, мы можем создавать точки следующим образом:

```
val point = Point(0, 0)
```

Однако у класса Pair есть два свойства, first и second, и как бы нам переименовать эти свойства так, чтобы стереть всякие различи
между желаемым классом Point и Pair? Сами свойства переименовать не удастся, но в нашем инструментарии есть замечательная
возможность, которую народные умельцы обозначили как мульти-декларации.

Мульти-декларации (Destructuring declaration)

Документация

Для простоты понимания примера рассмотрим ситуацию: у нас есть объект типа Point, как мы знаем из примера выше, это всего л
переименованный тип `Pair<Int, Int>`. Как видно из реализации класса Pair стандартной библиотеки, он помечен модификатором
это значит, что, среди прочего, в данном классе мы получаем сгенерированные методы componentN. Давайте о них и поговорим.

Для любого класса мы можем определить оператор componentN, который будет предоставлять доступ к одному из свойств объект
означает, что вызов метода point.component1 равносилен вызову point.first. Теперь разберемся, зачем нужно это дублирование.

Что такое мульти-декларации? Это способ "разложить" объект по переменным. Благодаря этой функциональности, мы можем нап
следующую конструкцию:

```
val (x, y) = Point(0, 0)
```

У нас есть возможность объявить сразу несколько переменных, но что окажется в качестве значений? Именно для этого нам и нуж
генерируемые методы `componentN`, в соответствии с порядковым номером, вместо N, начиная с 1, мы можем разложить объект на к
его свойств. Так, например, запись выше эквивалентна следующей:

```
val pair = Point(0, 0)
val x = pair.component1()
val y = pair.component2()
```

что в свою очередь равносильно:

```
val pair = Point(0, 0)
val x = pair.first
val y = pair.second
```

где first и second это свойства объекта Point.

Конструкция for в Kotlin имеет следующий вид, где x последовательно принимает значения 1, 2 и 3:

```
for(x in listOf(1, 2, 3)) { … }
```

Обратим внимание на блок `assertions` в DSL из основного примера. Для удобства часть его я приведу ниже:

```
for ((day, lesson, student, teacher) in scheduledEvents) { … }
```

Теперь всё должно быть очевидно. Мы перебираем коллекцию scheduledEvents, каждый элемент которой раскладывается на 4 свойства, описывающие текущий объект.

Extension функции

Документация

Добавление собственных методов к объектам из сторонних библиотек или добавление методов в Java Collection Framework — дав мечта многих разработчиков. И теперь у всех нас есть такая возможность. Объявление расширяющих функций выглядит следующ образом:

```
fun AvailabilityTable.monday(from: String, to: String? = null)
```

В отличии от обычного метода, мы добавляем название класса перед названием метода, чтобы обозначить какой именно класс мы расширяем. В примере `AvailabilityTable` это псевдоним для типа Matrix и, так как псевдонимы в Kotlin это только переименование результате такая декларация равносильна приведенной в примере ниже, что не всегда удобно:

```
fun Matrix<Boolean>.monday(from: String, to: String? = null)
```

Но, к сожалению, ничего с этим поделать нельзя, кроме как не использовать инструмент или добавлять методы только в определе класс контекста. Тогда магия появляется только там, где она нужна. Более того, вы можете расширять этими функциями даже интерфейсы. Хорошим примером будет метод first, расширяющий любой Iterable объект следующим образом:

```
fun <T> Iterable<T>.first(): T
```

В итоге, любая коллекция, основанная на интерфейсе Iterable, вне зависимости от типа элемента, получает метод first. Интересно что мы можем поместить extension метод в класс контекста и благодаря этому иметь доступ к расширяющему методу только в определенном контексте (см. выше лямбда с контекстом). Более того, мы можем создавать extension функции и для Nullable типов (объяснение Nullable типов выходит за рамки статьи, но при желании вы можете почитать здесь). Например, функция isNullOrEmpt стандартной библиотеки Kotlin, которая расширяет тип CharSequence?, может быть использована следующим образом:

```
val s: String? = null
s.isNullOrEmpty() //true
```

Сигнатура этой функции представлена ниже:

```
fun CharSequence?.isNullOrEmpty(): Boolean
```

При работе из Java с такими Kotlin функциями, extension функции доступны как статические.

Infix функции

Документация

Очередной способ подсладстить синтаксис — это infix функции. Проще говоря, благодаря этому инструменту мы получили возможность избавиться от лишнего зашумления кода в простых ситуациях.
Блок `assertions` из основного примера статьи демонстрирует использование этого инструмента:

```
teacherSchedule[day, lesson] shouldNotEqual null
```

Такая конструкция эквивалентна следующей:

```
teacherSchedule[day, lesson].shouldNotEqual(null)
```

Есть ситуации, когда скобки и точки излишни. Именно на этот случай нам нужен infix модификатор для функций.
В коде выше, конструкция `teacherSchedule[day, lesson]` возвращает элемент расписания, а функция `shouldNotEqual` проверяет, что элемент не равен null.

Чтобы объявить такую функцию необходимо:

- указать модификатор infix;
- определить ровно один параметр.

Вы можете комбинировать два последних инструмента, как в коде ниже:

```
infix fun <T : Any?> T.shouldNotEqual(expected: T)
```

Обратите внимание, что дженерик тип по умолчанию наследник Any (не Nullable иерархии типов), однако, в таких случаях, мы не м использовать null, по этому необходимо явно указать тип Any?.

Контроль контекста

Документация

Когда мы используем много вложенных контекстов, то на самом нижнем уровне получается гремучая смесь, так, например, без ка либо контроля может получиться следующая конструкция, не имеющая смысла:

```
schedule { //контекст SchedulingContext
    data { //контекст DataContext + внешний контекст SchedulingContext
        data { } //допустимо из-за отсутствия контроля контекста
    }
}
```

До версии Kotlin 1.1 уже существовал способ, как этого избежать. Создание собственного метода data во вложенном контексте DataContext, а затем пометка его аннотацией Deprecated с уровнем ERROR.

```
class DataContext {
    @Deprecated(level = DeprecationLevel.ERROR, message = "Incorrect context")
    fun data(init: DataContext.() -> Unit) {}
}
```

Благодаря такому поодходу мы могли исключить возможность недопустимого построения DSL. Однако, при большом количестве методов в SchedulingContext, мы получали определенное количество рутинной работы, отбивающей всё желание контролировать контекст.

В Kotlin 1.1 появился новый инструмент для контроля — аннотация @DslMarker. Она применяется на ваши собственные аннотации, которые, в свою очередь, нужны для маркирования ваших контекстов. Создадим свою аннотацию, которую пометим с помощью но инструмента в нашем арсенале:

```
@DslMarker
annotation class MyCustomDslMarker
```

Затем необходимо разметить контексты. В нашем основном примере это SchedulingContext и DataContext. Благодаря тому, что мы помечаем каждый из классов единым маркером DSL, происходит следующее:

```
@MyCustomDslMarker
class SchedulingContext { ... }

@MyCustomDslMarker
class DataContext { ... }

fun demo() {
    schedule { //контекст SchedulingContext
```

```
        data { //контекст DataContext + запрет на внешний контекст SchedulingContext
            // data { } //не компилируется, т.к. контексты помечены одним DSL маркером
        }
    }
}
```

Не смотря на всю восхитительность такого подхода, сокращающего кучу сил и времени, остается одна проблема. Если вы обратит внимание на наш главный пример, то увидите следующий код:

```
schedule {
    data {
        student {
            name = "Petrov"
        }
        ...
    }
}
```

В этом примере у нас появляется третий уровень вложенности и вместе с ним новый контекст Student, который, на деле, сущност класс, часть модели, а значит нам нужно пометить аннотацией @MyCustomDslMarker еще и сущностную модель, что, на мой взгля верно.

В контексте Student вызовы `data {}` всё так же запрещены, т.к. внешний DataContext никуда не делся, но эти конструкции остаютс валидны:

```
schedule {
    data {
        student {
            student { }
        }
    }
}
```

Пытаясь решить эту проблему с помощью аннотаций, у нас смешивается код для тестирования и бизнес код, а это в большинстве случаев нам не подойдет. Решения здесь три:

1. Использовать дополнительный контекст для создания студента, например, StudentContext. Это похоже на безумие и перестае оправдывать преимущества @DslMarker.
2. Создать интерфейсы для всех сущностей, например, IStudent (наименование здесь не важно), создать контексты-пустышки, наследующие эти интерфейсы, и делегировать реализацию объектам студентов, что тоже на грани бреда.

   ```
   @MyCustomDslMarker
   class StudentContext(val owner: Student = Student()): IStudent by owner
   ```

3. Воспользоваться аннотацией @Deprecated, как в примерах выше. В данном случае, пожалуй, это лучшее решение, которым мо воспользоваться.
   Просто добавляем deprecated extension метод для всех Identifiable объектов.

```
@Deprecated("Incorrect context", level = DeprecationLevel.ERROR)
fun Identifiable.student(init: () -> Unit) {}
```

В итоге, комбинируя разные инструменты, мы строим комфортный DSL для решения наших задач.

Минусы использования DSL

Попытаемся быть более объективными в применении DSL на Kotlin и разберемся, какие минусы есть у использования DSL в вашем проекте.

Переиспользование части DSL

Представим, что вам нужно переиспользовать часть своего DSL, вы хотите взять часть кода и дать возможность его легко повторит Хотя в самых простых случаях с единственным конктекстом мы можем спрятать повторяемую часть DSL в extension функцию, в большинстве ситуаций это нам не подходит.

Возможно, вы подскажете интересные варианты, но сейчас мне известно два решения этой проблемы: добавлять "именованные callback'и", как составляющую DSL, или плодить лямбды. Второй вариант проще, но его последствия могут превратиться в самый настоящий ад, когда вы пытаетесь отследить последовательность вызовов. Естественно, когда у нас появляется много императивн поведения подход с DSL начинает от этого страдать, отсюда и эта проблема.

This, it!?

Крайне легко потерять смысл текущего this и it в ходе взаимодействия со своим DSL. Если вы где-то используете it, как название параметра по умолчанию, и осознаете, что осмысленное название для этого параметра будет лучше, то просто сделайте это. Лучи немного очевидного кода, чем много неочевидных багов.

Наличие контекста может сбить с толку человека, который с ними никогда не работал. Однако теперь в вашем арсенале есть "лям контекстом" и вас стало еще труднее поставить в тупик появлянием странных методов внутри DSL. Помните, что на крайний случа можете присвоить контекст переменной, например, `val mainContext = this`

Вложенность

Эта проблема тесно переплетена с первым в нашем списке минусом. Использование вложенных во вложенные во вложенных конструкций двигает весь ваш осмысленный код вправо. Если это терпимо, то пусть так и остается, но в тот момент, когда вы сдвинулись "слишком сильно", разумно применить лямбды. Естественно, такой подход ухудшает читаемость DSL, но это некоторы компромисс, в том случае, когда DSL подразумевает не только создание структур, но и какую-то логику. При создании тестов на (кейс, который мы разбирали в ходе статьи), этой проблемы нет, т.к. данные описываются компактными структурами.

Где доки, Зин?

Если вы когда-либо подступались к чужому DSL, то у вас наверняка вставал вопрос: "Где документация?". На этот счет у меня ест мнение. Если вы пишете DSL, который будет использован не только вами, то лучшей документацией будут примеры использовани Сама по себе документация важна, но скорее в качестве дополнительной справки. Смотреть её довольно неудобно, т.к. наблюдат проблемно-ориентированного языка задается естественным вопросом: "Что мне нужно вызвать, чтобы получить результат?" и, по моему опыту, здесь эффективнее всего себя показывают примеры использования для схожих ситуаций.

Заключение

В статье мы рассмотрели инструменты, благодаря которым вы с легкостью построите собственный проблемно-ориентированный Теперь у вас не должно возникать сомнений о том, как это работает.

Возможно, я что-то ненамеренно пропустил, пожалуйста, напишите об этом в комментариях и статья будет дополнена. Важно пом что DSL не панацея. Когда получаешь такой мощный молоток, то всё подряд представляется гвоздём, но это не так.

Потренируйтесь "на кошках", как герой одного известного фильма, сделайте DSL для тестов, а затем, сделав множество ошибок, после появления опыта, рассмотрите и другие применения.
Желаю успехов в разработке проблемно-ориентированных языков!

Метки: dsl, kotlin, testing, программирование, обзор инструментов

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

20 сентября 2016 в 10:46

Платформа CUBA: Java RAD фреймворк с открытым кодом

+18 11,6k 45 9

Комментарии 11

bm13kk 02.11.17 в 12:12

Я не совсем понял. Где тут DSL? Я вижу Kotlin код, написаеном на поектно спецефичном фреймворке.

i_osipov 02.11.17 в 12:44

Вы верно заметили. Это обычный Kotlin код. Если быть точнее, это набор стандартных конструкций языка, благодаря которым мы строим проблемно-ориентированный язык, который специфичен для нашего проекта. Чем в вашем понимании «проектно специфичный фреймв отличается от DSL?

knekrasov 02.11.17 в 13:27

А где сам язык, который вы построили? Что он может, чем он удобен, в конце концов?

Я вообще не уверен, что хаки с синтаксическим сахаром и набор утилитных функций (которые не похожи на код, но компилируются! в прикольно!) — это уже DSL.

i_osipov 02.11.17 в 14:23

> А где сам язык, который вы построили?

Пример использования языка вы увидите в разделе «Пример финального результата»
Код доступен на GitHub по ссылкам в статье.

> Я вообще не уверен, что хаки с синтаксическим сахаром и набор утилитных функций (которые не похожи на код, но компилируют вот прикольно!) — это уже DSL.

Использование элементов языка общего назначения для получения специфичных для решаемой задачи конструкций — это один из вариантов построения DSL. Вы можете убедиться в этом сами по ссылке и в других источниках по запросу «internal domain-specific language»

burnashev 02.11.17 в 15:18

Возможно, в данном случае мы видим пример EDSL

Согласитесь, удобно описать сложный тест-кейс в виде кода
MyLittleDSL

```
schedule {
            data {
                startFrom("08:00")

                subjects("Russian",
                        "Literature",
                        "Algebra",
                        "Geometry")

                student {
                    name = "Ivanov"
                    subjectIndexes(0, 2)
                }
```

Альтернатива — многоэтажные if/case которые не всегда с первого раза прочтешь.

knekrasov 02.11.17 в 17:21

Да я понимаю. Spock Framework я сам пользовался для написания unit тестов, синтаксис куда более легковесный, чем в обычном JUnit (хотя под капотом используется именно он).

Я просто не понимаю помпы, с которой подается статья. Мы все делаем удобный для себя набор абстракций и mini-API для реше каких-то частных задач. Называть такой набор DSL и жаловаться на ограничения языка — ну это уж перебор. Хочешь свой язык чтобы тебя ничего не ограничивало — ANTLR в руки :-)

i_osipov 02.11.17 в 18:54

Далеко не все делают для себя «удобный mini-API». Kotlin DSL это легкий способ и API в терминах предметной области получ сил затратить мало, так еще и такой подход полностью совместим с вашим Java кодом. Не стоит принимать «минусы» как жа я хочу, чтобы у читателя было более объективное представление об инструменте до начала его использования. А про ANTLR д тестирования Java кода это вы отлично пошутили.

babylon 02.11.17 в 20:12

Можно на Kotlin реализовать JSONNET? Это же лучше любого DSL

knekrasov 03.11.17 в 11:55

Не шутил и не говорил такого. ANTLR — отдельно, Spock Framework со своим аккуратным синтаксисом — отдельно.

Pinsky 02.11.17 в 13:29

DSL — это, как мне кажется, чуть иная вещь, нежели библиотека/фреймворк под задачу. Это все таки — язык, со своим синтакисом и правилами.

Куда лучше рассматривать на роль DSL языки в Racket(например, тот же Scribble — DSL для оформления документов)

Aleosha 15.11.17 в 01:01

Непростая, но очень хорошая статья. При этом DSL'ы я не переношу, поскольку обычно они сводятся к «языку внутри языка», который понимает хорошо еще, если тот, кто его написал. Но примеры разобраны нетривиальные, и разобраны очень хорошо.

«Автоматический детектор спама». Или «О чем предупреждали Хемингуэй, Хаксли и Постман?»

+8 1,3k 19 2

Поджигаем голову как сосед или о социальном влиянии GT

+14 3,4k 17 7

Еще больше малого космоса. Британия хочет вернуться в стан космических держав GT

+10 1,3k 2 3

FPGA плата к Raspberry Pi GT

+21 5,5k 25 13

Рождение Software Tools: как и зачем появились GREP и AWK

+9 2,5k 26 2