

Тема 3. Построение доменно-специфичных языков (DSL) в языке Kotlin.

Романов Владимир Юрьевич

3.1. ПОСТРОЕНИЕ DSL НА ОСНОВЕ СИНТАКСИЧЕСКИХ ОСОБЕННОСТЕЙ KOTLIN.

Внутренний *DSL*

Внутренним DSL мы называется код:

- предназначенный для решения конкретной задачи.
Например для конструирования **SQL** запросов или разметки **HTML**.
- реализованный в виде библиотеки на языке общего назначения.
Например на языке Kotlin.

Синтаксические особенности языка *Kotlin*.

Синтаксические особенности языка **Kotlin** для создания **DSL**:

- Функция - расширение
- Инфиксный вызов функции
- Перегрузка операторов
- Соглашение о методе ***get()***
- Лямбда - выражение вне круглых скобок
- Лямбда - выражение с получателем

Функция - расширение класса

Обычный синтаксис - создание объекта-утилиты ***StringUtil***:

```
object StringUtil {  
    fun lastIndex(s: String): Int = s.length - 1  
}
```

```
val i = StringUtil.lastIndex("Name")
```

Функция - расширение класса ***String***:

```
fun String.lastIndex(): Int = length - 1
```

```
val k = "Name".lastIndex()
```

Инфиксный вызов функции

Класс данных ***Point***:

```
data class Point(val x: Int, val y: Int)
```

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

```
val stop = Point(1, 1)
```

Расширение класса ***Point*** обычной функцией ***lineTo***.

```
fun Point.lineTo(p: Point): Point = p
```

```
val p1 = start.lineTo(stop)
```

Расширение класса ***Point*** инфиксной функцией ***line***.

```
infix fun Point.line(p: Point): Point = p
```

```
val p2 = start line stop
```

```
val p3 = start.line(stop)
```

Перегрузка операторов

Для класса ***Point*** может быть введен оператор сложения точек **+**:

```
data class Point(val x: Int, val y: Int)
```

```
operator fun Point.plus(p: Point)  
    = Point(x + p.x, y + p.y)
```

```
val point1 = Point(10, 20)
```

```
val point2 = Point(40, 30)
```

```
// Оператор + для сложения точек
```

```
val point3 = point1 + point2
```

Соглашение о методе *get()*

Методу ***get*** с параметрами соответствует оператор индексации ***[]***:

```
data class Rectangle (
```

```
    var x: Int,
```

```
    var y: Int,
```

```
    var width: Int,
```

```
    var height: Int)
```

```
data class Point(val x: Int, val y: Int)
```



```
operator fun Rectangle.get(p: Point): Rectangle? {  
    if (p.x < x ) return null  
    if (x + width < p.x) return null  
    if (p.y < y ) return null  
    if (y + height < p.y) return null  
    return this  
}
```

```
var rect = Rectangle(0,0, 100, 100)  
var p = Point(50, 50)
```

// Оператор индексации для метода get

```
var r = rect[p]
```

Лямбда - выражение внутри и вне круглых скобок метода

Лямбда выражение как параметр в круглых скобках:

```
collection.forEach({ e -> printf(e) } )
```

Последний параметр можно выносить из круглых скобок, если этот параметр - лямбда выражение.

Лямбда выражение вне круглых скобок:

```
collection.forEach{ printf(it) }
```

Лямбда - выражение без получателя

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

```
lambda("hello")
```

Лямбда с контекстом

```
val lambda: Context.() -> Unit = {  
    println(this)  
}
```

```
“Hello, world”.out()
```

Переопределение операторов

В DSL:

`collection += element`

Обычно:

`collection.add(element)`

Псевдонимы типа

В DSL:

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

Обычно:

Создание пустых классов-наследников

Соглашение для *get/set* методов

В DSL:

```
map["key"] = "value"
```

Обычно:

```
map.put("key", "value")
```

Мульти-декларации

В DSL:

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

Обычно:

```
val p = Point(0, 0);
```


Лямбда за скобками

В DSL:

```
list.forEach { ... }
```

Обычно:

```
list.forEach({...})
```

Extention функции

В DSL:

```
mylist.first();
```

// метод first() отсутствует в классе коллекции mylist

Обычно:

Утилитные функции

Infix функции

В DSL:

```
1 to "one"
```

Обычно:

```
1.to("one")
```

Лямбда с обработчиком

В DSL:

```
Person().apply { name = "John" }
```

Обычно:

Нет

HTML - ПОСТРОИТЕЛЬ

Пример использования тегов html, head, body, title, h1, p

```
fun main() {  
    val result =  
        html {  
            head {  
                title { +"заголовок файла" }  
            }  
            body {  
                h1 { +"H1-заголовок" }  
                p { +"Текст параграфа" }  
            }  
        }  
    println(result);  
}
```

Сгенерированный HTML текст

```
<html>  
  <head>  
    <title>  
      заголовок файла  
    </title>  
  </head>  
  <body>  
    <h1>  
      H1-заголовок  
    </h1>  
    <p>  
      Текст параграфа  
    </p>  
  </body>  
</html>
```

Вызов функции *html*

- Это вызов функции с одним параметром - лямбда выражением.

```
html {  
  // ...  
}
```

- У этого выражения есть получатель - экземпляр класса **HTML**:

```
fun html(init: HTML.() -> Unit): HTML {  
  val html = HTML()  
  html.init()  
  return html  
}
```


Объявление функции *html*

```
fun html(init: HTML.() -> Unit): HTML {  
    val html = HTML()  
    html.init()  
    return html  
}
```

- Получатель **HTML** — это класс, который описывает тэг **<html>**.
- Он определяет потомков, таких как **<head>** и **<body>**

Объявление функции *html* (2)

```
fun html(init: HTML.() -> Unit): HTML {  
    val html = HTML()  
    html.init()  
    return html  
}
```

- Функция принимает один *параметр-функцию* под названием **init**.
- Тип этой функции: **HTML.() -> Unit** — функциональный тип с объектом-приёмником.
- *Объект-приёмник* значит, что в функцию передается экземпляр класса **HTML** (*приёмник*).
- Через ключевое слово **this** можно обращаться к членам объекта в теле этой функции.

Обращение к членам класса-получателя *head* и *body*

Объявление членов класса **head** и **body**:

```
class HTML() : TagWithText("html") {  
    fun head(init: Head.() -> Unit)  
        = initTag(Head(), init)  
  
    fun body(init: Body.() -> Unit)  
        = initTag(Body(), init)  
}
```

Использование членов класса **head** и **body**:

```
html {  
    this.head { /* ... */}  
    this.body { /* ... */}  
}
```

(**head** и **body** — члены класса **HTML**)

this МОЖНО ВЫКИНУТЬ !!!

Использование членов класса **head** и **body**:

```
html {  
  this.head { /* ... */}  
  this.body { /* ... */}  
}
```

(**head** и **body** — члены класса **HTML**)

Использование **this** для членов класса **head** и **body**

НЕ ОБЯЗАТЕЛЬНО:

```
html {  
  head { /* ... */}  
  body { /* ... */}  
}
```

Вызов *head*

```
html {  
  head { /* ... */}  
  body { /* ... */}  
}
```

- Вызов функции **head** создаёт новый экземпляр класса **Head**
- Затем вызывается функция **init** переданная в параметре
- Функция **init** инициализирует этот экземпляр класса **Head**
- После этого функция **init** возвращает значение экземпляра класса **Head**.

```
fun head(init: Head.() -> Unit) : Head {  
  val head = Head()  
  head.init()  
  children.add(head)  
  return head  
}
```

Сходство и различие *html*, *head* и *body*

- Функции **head** и **body** в классе **HTML** похожи на функцию **html**.
- Есть отличие. Они добавляют отстроенные экземпляры в коллекцию **children** охватывающего экземпляра класса **HTML**:

```
fun head(init: Head.() -> Unit) : Head {  
    val head = Head()  
    head.init()  
    children.add(head)  
    return head  
}
```

```
fun body(init: Body.() -> Unit) : Body {  
    val body = Body()  
    body.init()  
    children.add(body)  
    return body  
}
```

Обобщение функций *head* и *body*

- Похожие функции можно обобщить **head** и **body**
- Обобщенная версия называется **initTag**:

```
protected fun <T : Element>
initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
}
```

- Упрощенные функции *head* и *body*:

```
fun head(init: Head.() -> Unit)
    = initTag(Head(), init)
```

```
fun body(init: Body.() -> Unit)
    = initTag(Body(), init)
```

Использование упрощенных функций *head* и *body*

```
fun head(init: Head.() -> Unit) = initTag(Head(), init)
```

```
fun body(init: Body.() -> Unit) = initTag(Body(), init)
```

Упрощенные функции используются для построения тэгов **<html>** и **<body>**.

```
html {  
    head {  
        title {+"XML кодирование с Kotlin"}  
    }  
    // ...  
}
```


Добавление строки в тело тега

```
fun String.unaryPlus() {  
    children.add(TextElement(this))  
}
```

- Для строки в тело тега приписывается **+** перед текстом
- **+** - это вызов *префиксной операции* **unaryPlus()**.
- Операция **+** определена с помощью *функции-расширения* **unaryPlus()**,
- *Функции-расширения* **unaryPlus()** - член абстрактного класса **TagWithText** (родителя **Title**).
- Префикс **+** оборачивает строку в экземпляр **TextElement** и добавляет его в коллекцию **children**.

JSON - ПОСТРОИТЕЛЬ

Пример на языке *JSON*

```
{  
  "language": "Kotlin",  
  "lecturer": "Romanov",  
  "time": {  
    "from": "11:30",  
    "to": "13:00"  
  }  
}
```

Отличия языка *Kotlin* от языка *JSON*

- В Котлине нет **двоеточий**
 - **Двоеточия** используются для разделения объекта и типа
 - Поменять их поведение не получится
- В Котлине нет оператора **запятая**
 - **Запятая** используется для разделения аргументов
 - Этим можно воспользоваться
- **Фигурные скобки** в Котлине для создания **лямбды**
 - Запустить **лямбду** на выполнение не просто
 - Для **лямбды** придется передавать явно контекст
 - Но этим можно воспользоваться

В Котлине нет *двоеточий*

Использование инфиксных функций вместо двоеточий :

```
{  
    "language" to "Kotlin",  
    "lecturer" to "Romanov",  
    "time" to {  
        "from" to "11:30",  
        "to" to "13:00"  
    }  
}
```

В Котлине нет оператора *запятая*

- *Запятая* используется для разделения аргументов
- В Kotlin запятые можно убрать

```
{  
    "language" to "Kotlin"  
    "lecturer" to "Romanov"  
    "time" to {  
        "from" to "11:30"  
        "to" to "13:00"  
    }  
}
```

Фигурные скобки в Котлине создают лямбду

- В Котлине есть функции с *неявным получателем (receiver)* который может быть *контекстом* лямбда выражению

```
class Json {  
  
    companion object {  
        fun obj(content: JsonObjectBuilder.() -> Unit)  
            : JsonObject {  
            val b = JsonObjectBuilder()  
            b.content()  
            return b.res  
        }  
    }  
}
```

- **JsonObjectBuilder** предоставляет вспомогательные функции
- **JsonObjectBuilder** строит объект **JsonObject**

Класс JsonObject

```
class JsonObject {  
    val map = HashMap<String, Any>()  
  
    fun add(key: String, value: Any) {  
        map[key] = value  
    }  
  
    override fun toString() = map.toString()  
}
```


Класс JsonObjectBuilder

- **JsonObjectBuilder** предоставляет вспомогательные функции
- **JsonObjectBuilder** строит объект **JsonObject**

```
class JsonObjectBuilder {  
    internal val res = JsonObject()  
  
    infix fun String.to(value: Any) {  
        res.add(this, value)  
    }  
}
```

Использование построителя

```
package dsl.json
```

```
import dsl.json.Json.Companion.obj
```

```
fun main() {  
    val json: JsonObject = obj {  
        "language" to "Kotlin"  
        "lecturer" to "Romanov"  
        "time" to obj {  
            "from" to "11:30"  
            "to" to "13:00"  
        }  
    }  
}
```

```
    println(json)  
}
```

```
// {lecturer=Romanov, language=Kotlin, time={from=11:30, to=13:00}}
```

Использование построителя

```
package dsl.json

import dsl.json.Json.Companion.obj

fun main(args: Array<String>) {
    val jsonArgs = obj {
        for ((i, arg) in args.withIndex())
            "arg$i" to arg
    }
    println(jsonArgs)
}
```

Массивы в Json

- Вместо **obj** использовать **array**

```
"language" to "Kotlin",  
"lecturer" to "Romanov",  
"date" to array {  
    "05.12.2020",  
    "12.12.2020"  
}
```

Как **оживить** строки

Строка "05.12.2020" представляет элемент массива

- Строка *не вызывает никакой функции*
- Строка *никак не видна контексту*

```
"language" to "Kotlin",  
"lecturer" to "Romanov",  
"date" to array {  
    "05.12.2020",  
    "12.12.2020"  
}
```

Использование перегрузки для оператора +

```
"language" to "Kotlin",  
"lecturer" to "Romanov",  
"date" to array {  
    +"05.12.2020",  
    +"12.12.2020"  
}
```

Дополнение к классу *Json*

```
class Json {  
  
    companion object {  
        fun array(content: JsonArrayBuilder.() -> Unit)  
            : JsonArray {  
            val arr = JsonArrayBuilder()  
            arr.content()  
            return arr.res  
        }  
    }  
}
```

Класс *JsonArray*

```
class JsonArray {  
    val array = ArrayList<Any>()  
  
    fun add(any: Any) {  
        array.add(any)  
    }  
  
    override fun toString() = array.toString()  
}
```


Класс *JsonArrayBuilder*

```
class JsonArrayBuilder {  
    internal val res = JsonArray()  
  
    operator fun Any.unaryPlus() {  
        res.add(this)  
    }  
}
```

Использование построителя

```
package dsl.json
```

```
import dsl.json.Json.Companion.array
```

```
import dsl.json.Json.Companion.obj
```

```
fun main() {  
    val json: JsonObject = obj {  
        "language" to "Kotlin"  
        "lecturer" to "Romanov"  
        "time" to array {  
            +"05.12.2020"  
            +"12.12.2020"  
        }  
    }  
    println(json)  
}
```

```
// {lecturer=Romanov, language=Kotlin, time=[05.12.2020, 12.12.2020]}
```