

# 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.