

# Kotlin DSLを理解してみる



青柳 樹 (Itsuki AOYAGI) - シーサー株式会社  
Kotlin愛好会 Vol4 2018-09-20

# 自己紹介

- ▶ 青柳 樹 (Ituki AOYAGI)
- ▶ やぎにい (@yagi2, @yaginier)
- ▶ Androidエンジニア @ シーサー株式会社
- ▶ 普段は京都に居ます⛩
- ▶ Kotlin歴：1年と半年弱 🌱



# おしながき

- ▶ DSLとはなんぞや
- ▶ KotlinでDSLを作る時によく使う言語機能
- ▶ 実際にDSLを作ってみる

# DSL

Domain Specific Language  
(ドメイン固有言語)

特定の領域・分野の問題を解決するための言語。

身近なところだと

- ▶ SQL
- ▶ 正規表現
- ▶ make

# 内部と外部

## 内部DSL

アプリケーションから独立しているもの  
独自に構文解析を行う必要がある。

## 外部DSL

アプリケーション内で  
その言語を特定のドメイン向けに使う

以降の"DSL"は内部DSLについて喋っていきます

# KotlinでDSLを作るメリット

- ▶ ✨ 静的型付き言語なので  
コンパイル時に構文チェックが行われる ✨
- ▶ ✨ IDEでの補完が強い ✨
- ▶ ✨ Kotlinの言語機能が強い ✨

# Kotlin DSL Example

DSLは構造と文法を持つ。

普通に書いたケース

```
val html = createHtml()  
val table = createTable()  
  
html.addChild(table)  
  
val tr = createTr()  
table.addChild(tr)  
  
val td = createTd()  
tr.addChild(td)  
  
td.text = "セルです"
```

# Kotlin DSL Example

DSLは構造と文法を持つ。

DSLを用いたケース

```
val html = createHtml().  
    table {  
        tr {  
            td {  
                + "セルです。"  
            }  
        }  
    }
```

ぱっと見の圧倒的わかりやすさ 😅

[Kotlin/kotlinx.html: Kotlin DSL for HTML](#)

# DSL作成時良く使う言語機能

- ▶ 拡張関数・拡張プロパティ
- ▶ メソッドの引数の最後がラムダ式であれば、外側に出せる
- ▶ レシーバ指定のラムダ式
- ▶ 中置関数
- ▶ 演算子オーバーロード
- ▶ @DslMarkerアノテーション

# 拡張関数・拡張プロパティ

既存のクラスに新しいメソッドやプロパティを生やすことが出来る。

```
fun String.toPackedString() =
    this.replace("\n", "").replace(" ", "")

"""
{
    "name": "yagi2",
    "age": 24
}
""".toPackedString() // -> {"name": "yagi2", "age": 24}

var <T> List<T>.lastIndex: Int
    get() = size - 1

listOf(1, 2, 3).lastIndex // -> 2
```

# メソッドの引数の最後がラムダ式であれば 外側に出せる

```
fun hoge(num: Int, lambd: () -> String)  
  
hoge(100, { "this is lambda" })  
hoge(100) { "this is lambda" }  
  
fun piyo(lambda: () -> String)  
piyo({ "this is lambda" })  
piyo {  
    "this is lambda"  
}
```

ちょっとDSLっぽい見た目になる。

# レシーバ指定のラムダ式

ラムダ式にレシーバを指定して、ラムダ式の中のスコープのthisをそのクラスにする機能。

```
data class Fruit(val name: String, val num: Int)

fun hoge(lambda: Fruit.() -> Unit) {
    val apple = Fruit("りんご", 10)
    apple.lambda()
}

hoge {
    // このthisはFruitになっている
    println("${this.name}が${this.num}個あります。") // -> りんごが10個あります。

    // もちろんthisは省略可能
    println("${name}が${num}個あります。") // -> りんごが10個あります。
}
```

# レシーバ指定のラムダ式

## スコープの関数のwith

```
fun <T, R> with(receiver: T, block: T.() -> R): R { ... }
```

第1引数でTクラスを受け取り、そのTクラスを指定したラムダ式を第2引数で受け取る。

```
data class Fruit(val name: String, val num: Int)

val apple = Fruit("りんご", 10)
with(apple) {
    // apple(Fruitクラス)を渡しているので、ここ>thisはFruit
    println("${name}が${num}個欲しい!") // -> りんごが10個欲しい!
}
```

# 中置関数

メソッド呼び出し時に.`を`使わずにスペースで挟むこと（中置にすること）で呼び出せるようになる。`infix`をつけたメソッドが中置関数となる。

```
"a" to "b" // -> Pair<String, String>
```

これはTuples.ktに実装がある

```
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

# 中置関数

長澤太郎さんが作られているJUnitをシンプルに書けるライブラリ  
[ngsw-taro/knit: JUnit API set for Kotlin](#)での例

```
val actual = "hoge"
val expected = "hoge"

actual.should be expected
```

beが中置関数になっている。

```
interface Asserter<T> {
    infix fun be(expected: T)
}

internal class AsserterImpl<T>(override val target: T) : Asserter<T> {
    override fun be(expected: T) {
        target.should(`is`(expected))
    }
}
```

# 演算子オーバーロード

Kotlinでは既に用意されている数種類のオペレーターをオーバーロードして実装することができる。

## Operator overloading - Kotlin Programming Language

```
data class Point(val x: Int, val y: Int)
operator fun Point.plus(that: Point) = Point(this.x + that.x, this.y + that.y)

val p1 = Point(10, 10)
val p2 = Point(15, 20)

p1 + p2 // -> Point(25, 30)

// もちろんこれもOK
p1.plus(p2) // -> Point(25, 30)
```

# @DslMarkerアノテーション

プロックが入れ子になった際に、内側のブロックから外側のブロックにアクセスできるようになってしまふのが多々ある。  
それを防ぐためのアノテーション。

具体的には以下のようなケースを防ぐことが出来る。

```
html {  
    head {  
        // headの中のthisがthis@htmlになっている、this@headになってほしい  
        head { }  
    }  
}
```

[KEEP/scope-control-for-implicit-receivers.md at master · Kotlin/KEEP](#)  
[kotlinx.html/DSL-markers.md at dsl-markers · Kotlin/kotlinx.html](#)  
[Kotlin 1.1-M03 is here! | Kotlin Blog](#)

# Kotlin Gradle DSL

## build.gradle.ktsでのdependenciesの書き方

```
dependencies.implementation("androidx.constraintlayout:constraintlayout:1.1.3")  
  
dependencies {  
    implementation("androidx.constraintlayout:constraintlayout:1.1.3")  
}
```

それぞれの実装を見てみる

# 拡張関数のケース

```
dependencies.implementation("androidx.constraintlayout:constraintlayout:1.1.3")
```

dependenciesはDependencyHandlerクラスを返すgetDependencies()  
そのDependencyHandlerクラスに対して、拡張関数が生えている

```
fun DependencyHandler.`implementation`(dependencyNotation: Any): Dependency =  
    add("implementation", dependencyNotation)
```

# レシーバ指定のラムダ式をとるケース

```
dependencies {  
    implementation("androidx.constraintlayout:constraintlayout:1.1.3")  
}
```

dependenciesの実態は

```
class DependencyHandlerScope(val dependencies: DependencyHandler) : DependencyHandler by depen:  
    fun Project.dependencies(configuration: DependencyHandlerScope.() -> Unit) =  
        DependencyHandlerScope(dependencies).configuration()  
}
```

DependencyHandlerScopeクラスを指定したラムダ式を取っているため

```
dependencies {  
    // このthisはDependencyHandlerScope  
}
```

DependencyHandlerScopeはDependencyHandlerを継承しているので、その中で呼んでるimplementationはさっきの拡張関数。

# Anko

Ankoという、Androidアプリを制作を快適にするライブラリがある  
その中にAnko LayoutというDSLでレイアウトを組めるものがある

```
linearLayout {  
    padding = dip(30)  
  
    textView {  
        text = "A"  
    }.lparams {  
        margin = dip(30)  
    }  
}
```

Kotlin/anko: Pleasant Android application development

# 実際に作ってみる

AnkoのようなDSLでレイアウトをビルドするようなものを作ってみる  
(とりあえず汎用性は無視)

(時間がありそうならライブコーディングで作ります)

目指す完成形

```
linearLayout {  
    textView {  
        text = "Hello, Kotlin DSL!"  
        textColor = Color.RED  
    }.lparams {  
        margin = dip(10)  
    }  
}
```

LinearLayoutとTextViewを作れてLayoutParamsも指定できるように

# Layoutを作るDSLをつくる

```
linearLayout {  
    textView {  
        text = "Hello, Kotlin DSL!"  
        textColor = Color.RED  
    }.lparams {  
        margin = dip(10)  
    }  
}
```

外側から作っていく。

# LinearLayout

一番外側linearLayoutを作る

```
fun linearLayout(init: () -> Unit): LinearLayout =  
    LinearLayout(context)
```

これでここまで書けるようになった

```
linearLayout {  
}
```

# TextView

次にtextViewを同じように作っていく

```
fun textView(init: () -> Unit): TextView =  
    TextView(context)
```

これでここまで書けるようになった

```
linearLayout {  
    textView {  
    }  
}
```

# TextView

textView{}の中でtext = "text"を書きたい

→ レシーバ指定でtextView{}のラムダの中のthisをTextViewにする

```
fun textView(init: TextView.() -> Unit): TextView =  
    TextView(context).apply(init)
```

これでここまで書けるようになった

```
linearLayout {  
    textView {  
        text = "Hello, Kotlin DSL!"  
        setColor(Color.RED)  
    }  
}
```

# 親Viewへの追加

作ったTextViewを親(LinearLayout)に追加したい

→ textViewメソッドをLinearLayoutの拡張関数にする

```
fun LinearLayout.textView(init: TextView.() -> Unit): TextView {  
    val textView = TextView(context).apply(init)  
    addView(textView) // このメソッドのthisはLinearLayout  
    return textView  
}
```

# 親Viewへの追加

textViewメソッドをLinearLayoutの拡張関数にしたことで  
linearLayout{}の中でtextViewメソッドが呼べなくなった

→ linearLayout{}の中のthisをLinearLayoutにする

```
fun linearLayout(init: LinearLayout.() -> Unit): LinearLayout =  
    LinearLayout(context).apply(init)
```

これまでここまで書いて、レイアウトを表示することができるようになった

```
linearLayout {  
    textView {  
        text = "Hello, Kotlin DSL!"  
        setColor(Color.RED)  
    }  
}
```

# LayoutParamsの適用

lparamsメソッドを実装していく

```
fun TextView.lparams(init: () -> Unit): TextView {
    val lp = LinearLayout.LayoutParams(width, height)
    layoutParams = lp // this.layoutParams
    return this
}
```

これでここまで書けるようになった

```
linearLayout {
    textView {
        text = "Hello, Kotlin DSL!"
        setColor(Color.RED)
    }.lparams {
    }
}
```

# LayoutParamsの適用

lparams{}のラムダ内でmarginなどをセットしたい  
→ レシーバ指定してあげる

```
fun TextView.lparams(init: LinearLayout.LayoutParams.() -> Unit): TextView {  
    val lp = LinearLayout.LayoutParams(width, height)  
    layoutParams = lp // this.layoutParams  
    return this  
}
```

これでここまで書けるようになった

```
linearLayout {  
    textView {  
        text = "Hello, Kotlin DSL!"  
        setColor(Color.RED)  
    }.lparams {  
        setMargins(dip(10))  
    }  
}
```

# LayoutParamsの適用

LayoutParamsにwidthとheightを渡せるようにする (0のため)

```
fun TextView.lparams(  
    width: Int,  
    height: Int,  
    init: LinearLayout.LayoutParams.() -> Unit  
): TextView {  
    val lp = LinearLayout.LayoutParams(width, height)  
    layoutParams = lp // this.layoutParams  
    return this  
}
```

# LayoutParamsの適用

このままだと

```
lparams(WRAP_CONTENT, WRAP_CONTENT) {  
    setMargin(dip(10))  
}
```

という感じになるので、デフォルト値を与える  
そして引数のラムダを適用する

```
fun TextView.lparams(  
    width: Int = LinearLayout.LayoutParams.WRAP_CONTENT,  
    height: Int = LinearLayout.LayoutParams.WRAP_CONTENT,  
    init: LinearLayout.LayoutParams.() -> Unit  
) : TextView {  
    val lp = LinearLayout.LayoutParams(width, height).apply(init)  
    layoutParams = lp // this.layoutParams  
    return this  
}
```

# 9割完成

今までの実装で以下のように書いて表示することができるようになった

```
linearLayout {  
    textView {  
        text = "Hello, Kotlin DSL!"  
        setColor(Color.RED)  
    }.lparams {  
        setMargins(dip(10))  
    }  
}
```

setColor(),setMargins().....? 🤔

# プロパティのセット

setColor(),setMargins() を  
color = ..., margin =... で書きたい

理想の完成形

```
linearLayout {  
    textView {  
        text = "Hello, Kotlin DSL!"  
        color = Color.RED  
    }.lparams {  
        margin = dip(10)  
    }  
}
```

# setColorの実装

プロパティっぽくするならプロパティにするしかない

→ 拡張プロパティで解決

```
var TextView.textColor: Int
    get() = currentTextColor
    set(value) = setTextColor(value)
```

これでここまで書けるようになった

```
linearLayout {
    textView {
        text = "Hello, Kotlin DSL!"
        color = Color.RED
    }.lparams {
        setMargins(dip(10))
    }
}
```

# setMarginsの実装

marginも同じようにしていく

```
var LinearLayout.LayoutParams.margin: Int  
    set(value) = setMargins(value, value, value, value)
```

get().....🤔

# setMarginsの実装

setMargins()は4方向にセットするメソッドであるが  
当然だがgetMargin()は存在しない

→例外を投げよう！

```
var LinearLayout.LayoutParams.margin: Int
    get() = throw RuntimeException("This property does not have a getter.")
    set(value) = setMargins(value, value, value, value)
```

```
.lparams {
    val currentMargin = margin
}
```

と書けてしまう（実行時には落ちるが）  
いや、不親切すぎでは.....🤔

# setMarginsの実装

じゃあビルドできないようにしてあげよう  
→ @DeprecatedタグをERRORレベルでつけてあげる

```
var LinearLayout.LayoutParams.margin: Int
    @Deprecated("This property does not have a getter." level = DeprecationLevel.ERROR)
    get() = throw RuntimeException("This property does not have a getter.")
    set(value) = setMargins(value, value, value, value)
```

```
.lparams {
    // これはビルドできなくなる
    val currentMargin = margin
}
```

親切になった 😊👍

# 完成

これで最初の完成形が動いて、レイアウトが表示できるようになった

```
linearLayout {
    textView {
        text = "Hello, Kotlin DSL!"
        textColor = Color.RED
    }.lparams {
        margin = dip(10)
    }

    // コード側でレイアウトを組んでいるのでこういうのも書ける
    (1..10).forEach {
        textView {
            text = it.toString()
        }
    }
}
```

# 完成

DSLを作るときのコツ

- ▶ そのラムダ式の中のthisは何になっているのか
- ▶ 外側から作っていく

今回作ったDSLのリポジトリ  
[yagi2/LayoutDSLSample](#)

masterブランチが実装前で完成形のDSLが存在する  
answerブランチに実装が存在するので穴開きクイズで実装してみよう！

# おしまい

## Have a nice Kotlin!

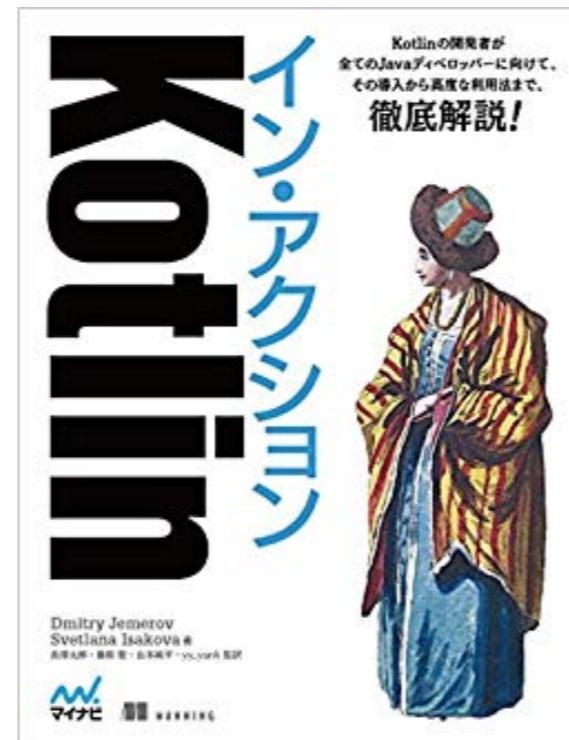


DSLを作るときのリファレンスとTry Kotlin  
[Type-Safe Builders - Kotlin Programming Language](#)  
[HTML Builder | Try Kotlin](#)

このスライドのリポジトリ  
[yagi2/love-kotlin-2018-09-20](#)

# Kotlin DSLをもっと知るには

Kotlinイン・アクションの11章がDSLについて詳しく書かれている



[Kotlinイン・アクション | Dmitry Jemerov, Svetlana Isakova, 長澤太郎, 藤原聖, 山本純平, yy\\_yank | 本 | 通販 | Amazon](#)