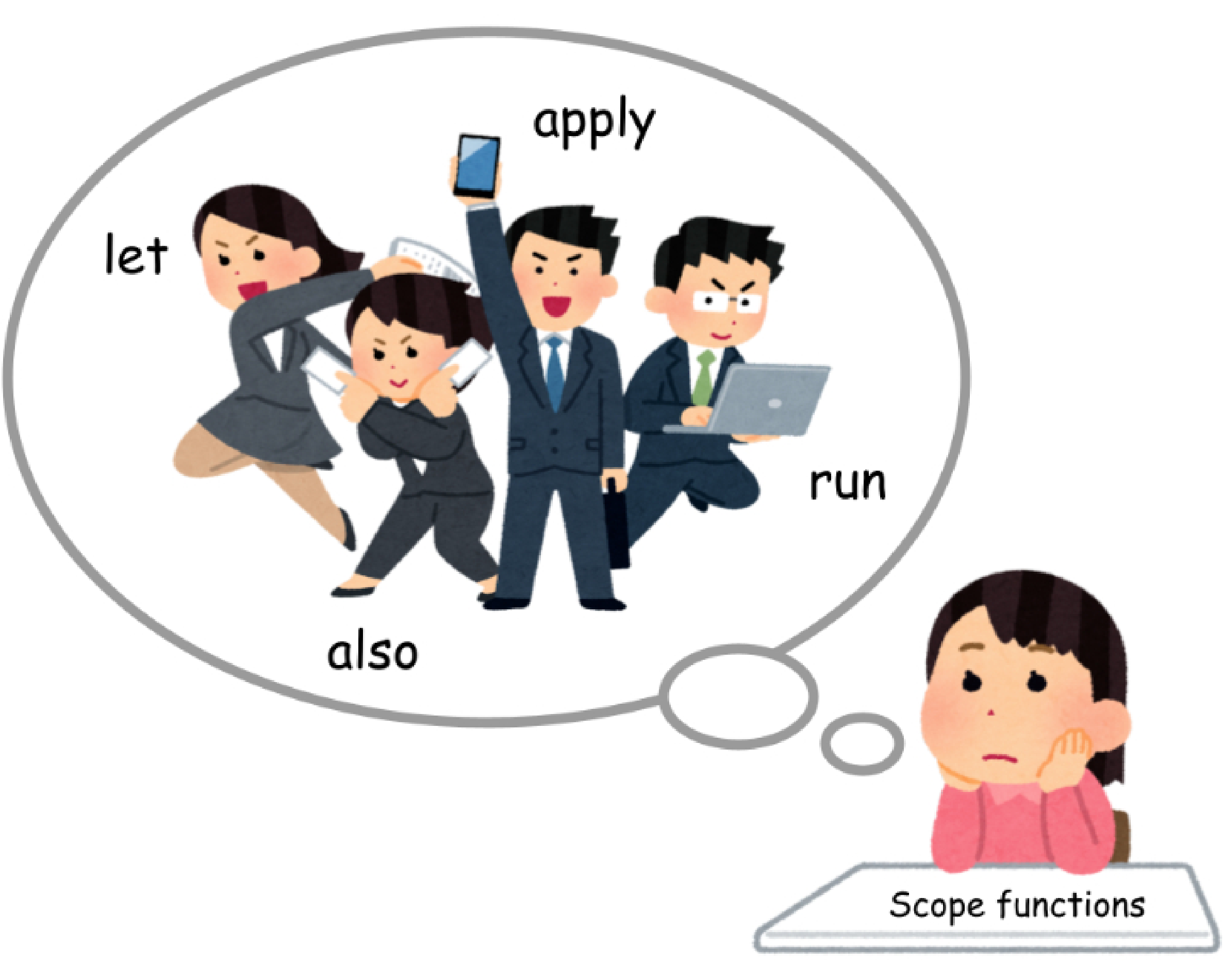
さて、Kotlin には標準ライブラリに便利なツールが搭載されており、その中でも今回はスコープ関数の紹介とその用途の考察を行います。



定義

Kotlin のスコープ関数 (Scope functions) は、オブジェクトのコンテキスト内でコードブロックを実行することを目的とした高階関数です。[*1]
標準ライブラリとして定義されており、あるオブジェクトに対して呼び出すと一時的スコープを生成し、そのオブジェクトに名前なしでアクセスすることができます。
`let` , `run` , `with` , `apply` , `also` の5つが存在しますが、実際に使うのは `let` , `run` , `apply` , `also` であり `with` はほぼ使う機会がないため今回は割愛します。

ブロック内で扱える対象オブジェクトとその戻り値は以下です。
公式がショートガイドとして提示している用途に応じた使い道も合わせて記載します。

関数定義	対象オブジェクト	戻り値	用途に応じた使い道
`let`	`it`	ラムダ式の結果（任意に指定可能）	NonNull オブジェクトに対してラムダ式を実行 ローカルスコープにて式を変数として導入
`run`	`this`	ラムダ式の結果（任意に指定可能）	オブジェクト設定及び結果の算出
`apply`	`this`	オブジェクト自身	オブジェクト設定
`also`	`it`	オブジェクト自身	追加効果

使い方

let

`let` は対象オブジェクトをブロック内にてラムダ引数 `it` として扱うことができ、戻り値は任意に指定可能です。[*2]
以下の例では仮の変数 `d` を経由して変数 `path` を取得していますが、 `let` でまとめることで途中処理はブロック内で行われるので、実際に必要な変数がわかりやすくなります。

```
val dir = File(dirPath)

// スコープ関数を使わない場合
val d = File(dir, "child")
if (!d.exists()) d.mkdir()
val path = d.absolutePath

↓

// let を使用した場合
val path = dir.let {
    val d = File(it, "child")
    if (!d.exists()) d.mkdir()
    d.absolutePath
}
```

また、NULL でない場合の If 文処理は `let` で書くとよいです。
以下の例では Nullable なクラスフィールドに対して分岐内で `!!` を使ってNonNull 型に変更した上で処理していますが、 `!!` は対象が NULL ならば Exception が発生する安全ではないキャスト方法であり、使用は控えたいところです。
ここで `let` を使えば NULL でなかった場合に NonNull 型として処理を安全に実行することが可能です。

```
private var dir: File? = null

// スコープ関数を使わない場合
if (dir != null) {
    val d = dir!!    // アンセーフなキャスト
    if (d.exists()) d.delete()
}

↓

// let を使用した場合
dir?.let {
    if (it.exists()) it.delete()
}
```

run

`run` は対象オブジェクトをブロック内にてラムダレシーバー `this` として扱うことができ、戻り値は任意に指定可能です。[*3]
私は主に NULL チェックの Return カットによく使用しています。
以下の例では `dir.listFiles()` が NULL ならば `run` のブロック処理に入り Return で処理がカットされるので、 `files` は NonNull 型が保障されます。
`run` ブロック内ではエラーログやコールバックなども自由に入れられることが可能です。

```
val dir = File(dirPath)

// スコープ関数を使わない場合
val files = dir.listFiles()
if (files == null) {
    println("Cannot get listFiles.")
    return
}

↓

// run を使用した場合
val files = dir.listFiles() ?: run {
    println("Cannot get listFiles.")
    return
}
```

apply

`apply` は対象オブジェクトをブロック内にてラムダレシーバー `this` として扱うことができ、戻り値はオブジェクト自身を返します。[*4]
以下の例のようにビルダークラス生成時に使うと、パラメータ設定がブロック内に収まり見やすくなります。

```
// スコープ関数を使わない場合
val request = HttpRequest.newBuilder()
    .uri(URI.create("https://www.optim.co.jp"))
    .GET()
    .setHeader("User-Agent", "xxxxxx")
    .timeout(Duration.ofMinutes(1))
    .version(HttpClient.Version.HTTP_1_1)
    .build()

↓

// apply を使用した場合
val request = HttpRequest.newBuilder().apply {
        uri(URI.create("https://www.optim.co.jp"))
        GET()
        setHeader("User-Agent", "xxxxxx")
        timeout(Duration.ofMinutes(1))
        version(HttpClient.Version.HTTP_1_1)
    }.build()
```

also

`also` は v1.1 から追加された関数で、対象オブジェクトをブロック内にてラムダ引数 `it` として扱うことができ、戻り値はオブジェクト自身を返します。[*5]
以下の例のように自前でビルダークラスを実装した場合、パラメータを追加するごとにセッターメソッドにていちいち `return this` を書く必要があります。
ここで `also` を使えば、戻り値はオブジェクト自身であるため `return this` を書く手間を省けて、スッキリ 1 行で書けます。

```
class Builder {
    private var userName: String? = null

    // スコープ関数を使わない場合
    fun setUserName(userName: String): Builder {
        this.userName = userName
        return this
    }

↓

    // also を使用した場合
    fun setUserName(userName: String): Builder = also { it.userName = userName }
```

考察

用途に応じた使い道の考察

`let` と `run` 、 `apply` と `also` ー それぞれの違いは、ブロック内で扱える対象オブジェクトがラムダ引数 `it` からラムダレシーバー `this` かだけです。
つまり、先程の `let` で書かれた例題は `run` で書くことができますし、その逆もまた可能です。
これは `apply` と `also` でも同様のことが言えます。

ここで疑問に思うのが「どれをどのような用途で使えばよいか？」です。
「どちらでも書くことができるならば、片方だけでよいのでは？」と考えるのも一理あります。
事実 `let` , `also` を使うほうがよいという考え方もあります。
`run` , `apply` は対象オブジェクトが `this` であるが故に、クラス参照の `this` と区別しづらいためです。

以下の例のように `Parent` , `Child` というクラスにそれぞれ `print()` という同じ名前のメソッドが存在し、 `Parent` は `Child` をフィールドとして保持していたとします。
`child.run` とした場合の `print()` は "Child" と出力されるのに対して、単純に `run` を呼び出した場合は "Parent" と結果が異なります。
また、 `child.run` にて `Parent` 側を呼び出す場合には `this@Parent` とクラスを明示する必要があります。

```
class Child {
    fun print() = println("Child")
}

class Parent {
    private val child = Child()

    fun print() = println("Parent")

    fun test() {
        child.run {
            print()              // Child
            this.print()         // Child
            this@Parent.print()  // Parent
        }
        run {
            print()              // Parent
            this.print()         // Parent
            child.print()        // Child
            this.child.print()   // Child
        }
    }
}
```

では上記の考えをもとに `let` と `also` だけ使うとした場合に、「 `run` と `apply` のほうがメリットになるパターンも存在するのでは？」という疑問に再度行き当たります。
わかりやすい例として、先ほどの `apply` の例を `also` に書き直します。
この場合、パラメータ設定ごとに毎回 `it` を付与する必要があるためちょっと冗長感があり、スコープ関数を使わないほうが見やすいとさえ感じてしまいます。

```
// also で書き直した場合
val request = HttpRequest.newBuilder().also {
        it.uri(URI.create("https://www.optim.co.jp"))
        it.GET()
        it.setHeader("User-Agent", "xxxxxx")
        it.timeout(Duration.ofMinutes(1))
        it.version(HttpClient.Version.HTTP_1_1)
    }.build()
```

結局のところ、複数人が同じ実装を書くにしても各々によってコードが異なるのと同じで、個々人の好みや思考思想に左右されるものであり正解はありません。
とは言うものの、これだと結論付けに投げやり感があるので、最初に言及した公式のショートガイドを軸に私個人のアレンジを加えた使い分けが以下になります。
あくまで私個人の解釈の範疇内でしかないためこれが正しいというわけではありませんが、ちょっとした参考程度にしていただければ幸いです。

- ブロック処理内で対象オブジェクトを使うことが前提で重要度が高い → `let`
- ブロック処理内で対象オブジェクトは使わない or そこまで重要度は高くない → `run`
- ブロック処理内で対象オブジェクトにパラメータを複数設定 → `apply`
- ブロック処理内で対象オブジェクトに設定を 1 つ追加 → `also`

複数のスコープ関数を組み合わせた場合の考察

スコープ関数は使い方で紹介した通り便利なものです。
もちろん、複数のスコープ関数を組み合わせることも可能です。

例えば、先ほど `let` にて If 文代わりの書き方を紹介しましたが、 `?.let { ... } ?: run { ... }` と連携させることで If-Else 文を表現することが可能です。
しかしスコープ関数を 2 つ立て続けに連携しているためか、少し見づらい印象を受けます。

```
private var dir: File? = null

// let と run を連携した場合
dir?.let {
    println("dir is not null. ${it.absolutePath}")
    if (it.exists()) it.delete()
} ?: run {
    println("dir is null.")
}
```

また、以下のように扱い方次第では可読性がかなり低下してしまいます。
以下の例では `let` のネストが増えた＋ `run` の Else 処理が下になったことが起因して、わかりにくいコードになっています。
この場合はシンプルに `run` を使用して、上から順に Return カットするとわかりやすいコードに改善されます。

```
private var inputFile: File? = null
private var outputFile: File? = null

// 良くない例
inputFile?.let { inFile ->
    outputFile?.let { outFile ->

        ...

    } ?: run {
        println("outputFile is null.")
    }
} ?: run {
    println("inputFile is null.")
}

↓

// 改善後
val inFile = inputFile ?: run {
    println("inputFile is null.")
    return
}
val outFile = outputFile ?: run {
    println("outputFile is null.")
    return
}
...
```

こうやって見ると複数のスコープ関数を連携するのは控えるべきで、なるべく 1 つだけに絞ってシンプルに書くのがよさそうですね。
何事もほどほどに、過剰は良くないということです。

Return カットの効率的な実装方法

これは考察というよりは効率的な実装にするための小技になります。
先ほど `run` にて Return カットに使用する方法を紹介しました。
ここでお勧めしたいのが、複数のクラスから呼び出される可能性がある共通メソッドを実装する際に、戻り値を `Boolean` ではなく Nullable 型にすることです。

例えばファイル A から B へコピーするようなメソッドが存在したと仮定します。
この場合、処理結果を `Boolean` （成功時に `true` , 失敗時に `false` ）で書くと以下のようになります。
大抵は呼び出し元で失敗時に Return カットすると思いますが、カットの条件 `!` を付与する必要があることから、該当の共通メソッドを使用する度に条件を反転し損ねるリスクがあり、もれなくバグに繋がってしまいます。

```
// 以下のような共通メソッドを想定
fun copyFile(inputFile: File, outputFile: File): Boolean {
    ...
}

// 呼び出し側での処理、`!` を忘れると条件が反転してしまいバグる
if (!copyFile(inputFile, outputFile) {
    println("Failed to copy file.")
    return
}
...
```

ここであえて Nullable 型（成功時に `outputFile` そのもの、失敗時に `null` ）を返すようにすると、呼び出し元で `?: run { ... }` で連携可能でです。
この書き方を採用すれば、先ほどの反転条件忘れによるバグ発生のリスクがなくなります。

```
// File? で返す共通メソッドを定義
fun copyFile(inputFile: File, outputFile: File): File? {
    ...
}

// 先ほどのバグリスクがなくなる
val resultFile = copyFile(inputFile, outputFile) ?: run {
    println("Failed to copy file.")
    return
}
...
```

まとめ

今回は Kotlin スコープ関数の使い方の紹介とその用途を考察しました。

用途に応じた使い道は「これはこうだ」という確固たる答えがなく、個々人の解釈によって意見が異なるところです。
私自身執筆するにあたって深掘りすればするほど思考が発散してしまい、発散したものを一つの記事として収束させるのにだいぶ苦労しました。
しかし答えはないと言うものの、少なくとも各プロジェクトごとに一定の決まり事を定めておくことで、コード全体の統一感を持たせるべきかと思います。

また、スコープ関数は便利であるものの、扱い方によってはかえって可読性が低下した見づらいコードになってしまいます。
処理は上から順番に、バグ発生の低い書き方を採用しつつ、シンプルイズベストで書きましょう。