1.6. Обобщенное программирование в Kotlin. Расширения в Kotlin

Сайт:	Samsung Innovation Campus	Напечатано::	Павел Степанов
Курс:	Мобильная разработка на Kotlin	Дата:	вторник, 31 октября 2023, 13:06
Книга:	1.6. Обобщенное программирование в Kotlin. Расширения в Kotlin		

Оглавление

Расширения функциональности типов

При печати списков в консоли функцией `println()` обычно все элементы разделены пробелом и запятой. Для ряда случаев это неудобно и нужно оформить список отдельными строками, например так:

- элемент 1
- элемент 2
- элемент 3

Возникает задача дополнить интерфейс `List` новой функцией, назовём её `toItemsByLines()`. В Java существующий класс/интерфейс нельзя изменять или дополнять. В Kotlin Вы можете определять новые методы в произвольных классах/интерфейсах - расширения функциональности. Более того, значительная часть стандартной библиотеки языка реализована с применением этого подхода. Функции расширения не имеют доступа к внутренней реализации класса и это является определённым неудобством при их определении.

Для нашего случая интерфейс `List` можно дополнить так:

```
fun List<String>.toItemsByLines(): String {
    return this.joinToString ( separator = "\n* ", prefix = "* ", postfix = "\n")
}
```

Здесь мы с помощью `this` ссылаемся на экземпляр класса, у которого вызван метод `toItemsByLines()` и объединяем его элементы разделителями. Подробно синтаксис `joinToString` описан в документации

Аналогичным образом можно, например, посчитать число пробелов в произвольной строке:

```
fun String.countSpaces(): Int {
    var n = 0
    for (l in this.toCharArray()) {
        if (l ==' ' ) {
            n ++
        }
    }
    return n
}
```

Функции расширения часто используются для проверки свойств, корректности данных и приведения к определённому формату. В примере ниже мы проверим, содержит ли строка корректный по форме IP адрес. Строго говоря, нужно проверять каждый октет адреса на принадлежность к диапазону, но эту задачу предлагаем читателю проделать самостоятельно.

```
fun String.isValidIPv4(): Boolean {
    val IPV4_PATTERN = "^[0-9]+.[0-9]+.[0-9]+.[0-9]$"
    val ipv4regex = Regex(IPV4_PATTERN)
    return this.matches(ipv4regex)
}
```

Поскольку функции расширения определены вне классов, бывает сложно отследить их использование. Можно ограничить область видимости функции расширения объявив её внутри класса или использовав модификатор `private`.

Функции области видимости

Понятие операторного блока

Операторным блоком (scope) во многих языках программирования называется фрагмент программы внутри которого определяются переменные. Типичными примерами блоков являются цикл и определение функции:

```
for (w in words) {
    if (w.length in lengths) {
        print(w)
    }
}

fun area(a: Int = 0, b: Int) {
    print("area: ${a * b}")
}
```

Переменные, определённые внутри таких блоков недоступны извне, поэтому часто операторный блок называют ещё областью видимости (scope). Обратите внимание, что лямбда-выражения тоже являются операторным блоком. В языке Kotlin определены шесть функций для операций над операторным блоком в целом. Вы часто встретите применение таких функций в программах на Kotlin, т.к. они позволяют сделать код лаконичней и наглядней.

let()

Функция `let()` может быть вызвана у любого объекта, в качестве параметра в фигурных скобках она принимает лямбда-выражение, которое будет применено к нему. При этом имя переменной в которой будет храниться ссылка на этот объект можно задать явно или опустить (тогда переменная будет называться `it`)

```
val name = "Petya"
name.let {
        n ->   println("$n")
}

name.let {
    println("$it")
}
```

Применение `let()` позволяет более гибко работать с nullable-типами. Если объект, у которого вызвана `let()` содержит `null`, то лямбда-выражение вызвано не будет.

```
val age: Int? = null
age.let {
        println("Your age is $it")
}
```

apply()

Второй функцией является `apply()`. Она позволяет упростить обращения к объектам с вложенной структурой и сократить объем кода. Часто в создании мобильных приложений приходится обращаться ко вложенным полям и методам, например:

```
val intent = Intent(this, SecondActivity::class.java)
intent.putExtra("name", "Vasya")
intent.putExtra("age", 21)
intent.putExtra("password", "Bender789")
```

Последние 3 строки можно более лаконично переписать, применив функцию `apply()`

```
intent.apply() {
    putExtra("name", "Vasya")
    putExtra("age", 21)
    putExtra("password", "Bender789")
}
```

Аналогичным образом удобно задавать параметры объекта, если этого не позволяет сделать конструктор, например для установки цвета и стиля отрисовки на канве (`Canvas`)

```
val paint = Paint().apply {
    color = Color.YELLOW
    style = Paint.Style.STROKE
    textSize = textHeadlinePx
}
```

run()

Уже знакомое нам поведение функций `let()` и `apply()` сочетается в `run()`: вызываемый объект, как и в случае с `apply()`, доступен как `this` в лямбда-выражении, а результат последней операции возвращается как значение всего лямбда-выражения. Тем не менее, поведение `run()` также похоже на `let()`, так как возвращает в качестве значения результат последнего выражения. Вызов `run()` полезен в случае вызова множества методов для одного объекта, но в результате возвращает не собственно объект, а результат последнего выражения. В примере ниже мы добавляем в список несколько строк и возвращаем строковое представление объекта.

```
data class Message(val text: MutableList<String> = mutableListOf()) {
    fun add(s: String) {
        text.add(s)
    }
}

fun main() {
    val msg = Message().run {
        add("This")
        add("is")
        add("text")
        toString()
    }
    println(msg)
}
```

Результат работы программы: `Message(text=[This, is, text])`

with()

Функция `with()` очень похожа на `run()`: обе передают вызываемый объект параметром в лямбда-выражение и обе возвращают значение, полученное в блоке. Отличие лишь в форме вызова функций: `run()` вызывается как функция самого объекта, а в `with()` объект передаётся в виде параметра. Сравните с примером выше для `run()` (используемый класс и вывод программ идентичен).

```
fun main() {
    val msg = with(Message()) {
        add("This")
        add("is")
        add("text")
        toString()
    }
    println(msg)
}
```

Главные особенности `with()`: в функцию передаётся объект, который доступен внутри выражения как `this`. Результат последнего выражения возвращается как итоговое значение `with()`

also()

Как и функция `run()`, `also()` сочетает в себе свойства `apply()` и `let()`. Объект, для которого вызвана `also()` становится доступен в качестве параметра в лямбда-выражении в виде переменной `it`. Как и `apply()`, `also()` возвращает полученный объект. Используем тот же пример, что для `with()` и `run()`, заметим, что отличие лишь в синтаксисе обращения к передаваемому объекту (используемый класс и вывод программ идентичен).

```
fun main() {
    val msg = Message().also {
        it.add("This")
        it.add("is")
        it.add("text")
        toString()
    }
    println(msg)
}
```

Начать тур для пользователя на этой странице