

## Тема 2. Вступ до функціонального програмування мовою Kotlin

Починаючи з Java 8, у даній мові розвиваються синтаксичні інструменти функціонального програмування. Мова програмування Kotlin теж володіє аналогічними можливостями: передача функцій у якості аргументів, лямбда-вирази, ліниві обчислення (lazy evaluation), функції вищого порядку тощо.



Доповідь

Техніки функціонального програмування  
Чисті та високорівневі функції, каррінг, мемоізація та ін.

В основу парадигми функціонального програмування лягає намагання використовувати функції без *побічних ефектів (side effects)*. Під побічним ефектом будемо розуміти будь-які зміни загального стану системи: вивід повідомлення на екран, зміна значення властивості об'єкту тощо. На відміну від функціонального програмування, об'єктно-орієнтована парадигма в цілому побудована на передачі повідомлень між об'єктами, що змінюють внутрішній стан цих об'єктів. Також більшість функціональних мов програмування реалізують концепцію *прозорості посилань (referential transparency)*: для однакових вхідних даних функція завжди повертатиме однакові вихідні дані. Якщо функція підлягає такому означенню, її називають *чистою*.

### Функціональні типи та анонімні функції в мові Kotlin

Однією з базових ідей, які лежать в основі функціонального програмування, є *першокласні (first-class)* функції. Вони передбачають можливість оперувати функціями таким же чином, як і іншими мовними елементами: передавати в якості аргументів інших функцій, повертати функції з функцій та присвоювати функції змінній. Функції, які отримують функцію в якості параметру або повертають функцію, називають *функціями вищого порядку (higher-order functions)*. Впровадження таких можливостей базується на *функціональних типах*, які записуються подібним чином: `(Int, Int) -> Float`. Такий тип передбачає функцію, яка приймає на вхід два цілих числа та повертає дробове значення. Якщо функція нічого не повертатиме, після оператора “->” слід ставити тип Unit.

Функції, які розглядалися у попередній темі, були іменованими, а для їх виклику потрібно було записувати відповідні назви. Проте якщо тіло функції невелике, зручніше вбудувати його в потрібному місці, а не створювати окрему функцію. Для такого випадку цей блок коду не потрібно іменувати, тобто функція або метод стають *анонімними*. Для демонстрації розглянемо функцію додавання двох цілих чисел. Її можна записати різними способами. Лістинг 2.1 пропонує дві форми запису іменованих функцій (рядки 1-3 та рядок 5), а також два варіанти запису анонімних функцій, які також називають *лямбда-виразами* (рядок 9 та

рядок 10). Усі виклики повернуть число 9. Таким чином, лямбда-вираз характеризується функціональним типом та є чудовим варіантом для передавання в якості параметру функції або повернення результату її роботи.

### **Лістинг 2.1. Демонстрація лямбда-виразів**

```
1 fun sum1(a:Int, b:Int):Int {  
2     return a+b  
3 }  
4  
5 fun sum2(a:Int, b:Int) = a+b  
6 fun main() {  
7     println(sum1(4,5))  
8     println(sum2(4,5))  
9     val c1 = {a:Int, b:Int -> a + b}  
10    val c2: (Int, Int)->Int = { a, b -> a+b }  
11    println(c1(4,5))  
12    println(c2(4,5))  
13 }
```

Парадигма функціонального програмування у своїй основі пропонує писати код шляхом подібних вкладених викликів функцій.

Чисті функції складатимуть фундамент функціонального програмування, саме на їх основі компонується програма. Функціональний код має декларативний характер та виглядає як ланцюг викликів одних функцій як параметрів інших функцій. Така послідовність завершується на викликах чистих функцій, які замкнені на собі. У свою чергу, функції, яким передаються або які повертають інші функції, називають **функціями вищого порядку**. До таких, зокрема, відносяться функції `map()`, `filter()`, `zip()` та ін., що будуть розглядатись далі в темі.

Для лямбда-виразів з одним параметром у мові Kotlin передбачена спрощена процедура запису – даний параметр замінюється на вбудовану змінну `it`. Це дозволяє ще більше спростити запис, прибравши стрілковий оператор:

```
val cube: (Int)->Int = {it * it * it}
```

Проте така можливість доступна лише у випадку лямбда-виразів з одним параметром.

Для зручності роботи з типами мова програмування Kotlin пропонує забезпечувати їх альтернативні назви (псевдоніми) за допомогою ключового слова `typealias`. Зокрема, можна дещо спростити оголошення однотипних лямбда-виразів:

```
typealias cubedI = (Int) -> Int
```

### **Лінійні обчислення та послідовності**

Однією з особливостей функціональної парадигми програмування є використання **лінійних обчислень** (*lazy evaluation*), суть яких зводиться до повноцінного створення об'єктів не відразу після оголошення, а лише в момент

необхідності їх обробки. Розглянемо відкладені обчислення на прикладі спеціального виду колекцій у мові Kotlin – *послідовностей (sequence)*. Вони є аналогами генераторів з мови програмування Python і можуть динамічно генерувати потрібну кількість елементів колекції. Продemonструємо це в лістингу 2.3.

### Лістинг 2.3. Генерація послідовності парних чисел

```


1 fun main() {
2     val numbers = generateSequence(0) {
3         println("Згенеровано число: ${it+2}")
4         it + 2
5     }
6
7     val evenNmbers = numbers.take(5)
8     for (i in evenNmbers) {
9         println(i)
10    }
11 }
```

Суть лінивих обчислень показано в виводі програми (рис. 2.1). Заповнення послідовності відбувається тільки під час здійснення з нею деяких операцій. У даному випадку, взяття перших п'яти елементів за допомогою методу take() у рядку 7 лістинга 2.3. Якщо прибрати вивід послідовності в рядках 8-10, у консоль нічого не буде виведено. Таким чином, насправді створюється нескінченна послідовність, кількість елементів якої уточнюється в подальшому.

```

0
Згенеровано число: 2
2
Згенеровано число: 4
4
Згенеровано число: 6
6
Згенеровано число: 8
8
```

Рис. 2.1. Згенеровані елементи послідовності парних чисел

 <p><b>Завдання</b></p>	<p>Поекспериментуйте з методом drop() для отримання елементів послідовності. Чому програма поводить себе так? Згенеруйте послідовність з 10 випадкових цілих чисел значенням від 1000 до 100000.</p>
--	--

Послідовності передбачають два доступних види операцій над собою: проміжні та термінальні. Аналогічно до потоків даних (streams) у Java, негайні оператори є лінивими, вони створюють нову послідовність у результаті свого виклику. Зверніть увагу, що послідовність не зберігається в пам'яті, а операції виклику будуть перераховувати кожний її елемент. Послідовності створюються на

базі ітератора з посиланням на проміжні операції, які слід виконати. Функції `map()`, `distinct()`, `groupBy()` та ін. є проміжними та повертають `Sequence`-об'єкт. Для послідовності вони ставляться в чергу передбачених операцій та не виконуються відразу. Це дозволяє створювати ланцюговий виклик проміжних операцій, поки не буде викликано термінальну операцію.

Термінальні операції отримують значення з послідовності. Вони виконують прохід по послідовності та переривають його, як тільки виконається задана умова. Функції `first()`, `toList()`, `count()` та ін. – термінальні та не повертають `Sequence`-об'єкт.

**Послідовності** більш доречні для великих колекцій з багатьма операціями, які потрібно виконати над ними. Колекції використовують вбудовані (`inline`) функції, тому байткод операцій та переданих лямбда-виразів вбудовується. Колекції створюють новий список для кожного перетворення, а послідовності містять посилання на перетворюючі функції. При невеликих колекціях з декількома операціями різниця продуктивності коду є несуттєвою, проте у роботі з великими списками швидкодія може значно знизитись. У такому випадку краще використовувати послідовності. Мова Kotlin дозволяє перетворити ітеровану колекцію в послідовність за допомогою методу `asSequence`:

```
val array = arrayOf('a', 'b', 'c')
val sequence = array.asSequence()
```

### Функції вищого порядку

Однією з найпростіших для розуміння функцій вищого порядку є метод `filter()`. Він спрямований на вибірку елементів колекції згідно з певним критерієм. “Функціональність” його використання полягає в тому, що критерії відбору задаються у вигляді іншої функції, яка стає параметром для `filter()`. Набір даних, який потрібно відфільтрувати, буде об'єктом, до якого прив'язується метод `filter()`. У лістингу 2.2 показано вибирання парних чисел зі списку (рядок 4).

#### Лістинг 2.2. Демонстрація роботи методів `filter()` та `map()`

```
1 fun main() {
2     val numbers =
3         mutableListOf<Int>(7, 10, 19, 16, 4, 231, 78)
4     val evenNumbers = numbers.filter { it % 2 == 0 }
5     println(evenNumbers)
6     val cube: (Int) -> Int = { it * it * it }
7     val cubedNumbers = numbers.map { cube(it) }
8     println(cubedNumbers)
9 }
```



#### Завдання

Створіть колекцію зі слів, які починатимуться з різних літер. Продемонструйте фільтрування колекції, відібравши з неї всі слова, які починаються з літер “a”, “v” або “y”.

Механізм [рефлексії](#) в мові Kotlin дозволяє отримувати посилання на класи, функції та властивості. Звідси, посилання на іменовану функцію можна передати в якості параметру іншій функції, що й є базовим механізмом функціонального програмування. Для цього в мові Kotlin присутній оператор посилання на функцію «::».

Продемонструємо простий відбір парних чисел зі списку, коли в якості параметру функції `filter()` виступає посилання на функцію `isEven()` (лістинг 2.4). У даному випадку `isEven` є об'єктом функціонального типу (`Int`) -> `Boolean`.

#### Лістинг 2.4. Передача іменованих функцій у якості параметрів

```
1 fun isEven(x: Int) = x % 2 == 0
2
3 fun main() {
4     val numbers = listOf(7, 12, -3, -8, 15, 16)
5     println(numbers.filter(::isEven))
6 }
```

Іншою корисною функцією вищого порядку є метод `map()`, який застосовує деяку функцію до кожного елементу колекції. Традиційний підхід передбачав використання циклу, в якому відбувалось ітерування по колекції та виклик функції до відповідного її елементу. У функціональному стилі програмування дані дії записуються одним рядком (лістинг 2.2, рядок 7). Таким чином, код стає більш модульним та компактним.

Мова програмування Kotlin пропонує багато можливостей для фільтрування колекцій, крім розглянутих вище функцій `take()`, `drop()`, `filter()`. Також можна виокремити (`slice`) частину колекції за допомогою функції `slice()`, відібрати унікальні елементи зі списку за допомогою функції `distinct()` та розбити ітеровані колекції (не працює з `Arrays`) на частини за допомогою функції `chunked()`. Приклад застосування цих методів наведено в лістингу 2.5.

#### Лістинг 2.5. Інші фільтраційні методи

```
1 fun main() {
2     val intArray = arrayOf(7, 1, 3, 3, 12, 6, 7, 1, 9, 12)
3     val dnaFragment = "ATGTTTCGAAACGATTTCCAA"
4     println(intArray.slice(1..8 step 2))
5     println(intArray.distinct())
6     println(dnaFragment.chunked(4))
7 }
```

У результаті запуску на екран буде виведено наступне:

```
[1, 3, 6, 1]
[7, 1, 3, 12, 6, 9]
[ATGT, TCGA, AACG, ATTT, CCAA]
```

Наступна функція вищого порядку – метод `zip()`. Він на основі двох колекцій формує нову колекцію з пар їх відповідних значень. Таким методом буде корисним, наприклад, за потреби об'єднати два списки у єдиний список. Для

демонстрації візьмемо задачу побудови телефонного довідника з окремих списків імен та відповідних телефонів (лістинг 2.6).

**Лістинг 2.6. Застосування методу zip()**

```
1 fun main() {  
2     val names = mutableListOf<String>()  
3     val phones = mutableListOf<Long>()  
4     for (i in 0..10) {  
5         names.add("Абонент$i")  
6         phones.add(380_000_000_000 + (Math.random() *  
7             1_000_000_000).toLong())  
8     }  
9     val abonents = names.zip(phones)  
10    for (abonent in abonents) {  
11        println("${abonent.first}->+${abonent.second}")  
12    }  
13 }
```

Для представлення телефонних номерів використовуються числа типу Long, значення яких формуються за допомогою генератора випадкових чисел. У результаті виконання методу zip() утворюється список типу List<Pair<String, Long>>. Приклад виводу програми з лістингу 2.6 наведено на рис. 2.2.

```
Абонент0->+380789291138  
Абонент1->+380055009278  
Абонент2->+380543752522  
Абонент3->+380193750133  
Абонент4->+380215294712  
Абонент5->+380109791106  
Абонент6->+380927456454  
Абонент7->+380205553823  
Абонент8->+380184511742  
Абонент9->+380194863566  
Абонент10->+380552115666
```

**Рис. 2.2. Програмний вивід для лістингу 2.4**

Метод flatMap() дозволяє «сплюснути» вкладені колекції, перетворивши багатовимірну колекцію в одновимірну. Більшість методів для роботи з колекціями націлені саме на одновимірні набори даних, тому подібне «сплюснення» часто корисне на практиці. Для прикладу продемонструємо усереднення квартальних прибутків (список з чотирьох значень у тисячах грн) для мережі магазинів (лістинг 2.7). У даному випадку, без «сплюснення» списку неможливо викликати метод average(), який усереднює значення. Метод flatMap() дозволяє значно спростити код, поелементно переписуючи дані в новий список. Такий же результат можна отримати, застосовуючи метод flatten().



**Лістинг 2.7. Застосування методу flatMap()**

```

1 fun main() {
2     val quartalRevenue = listOf(
3         listOf(312.78, 256.12, 451.76, 389.53),
4         listOf(332.28, 304.26, 461.06, 409.38),
5         listOf(320.71, 296.02, 444.77, 430.00),
6         listOf(352.80, 332.75, 503.16, 489.33)
7     )
8     val mutableTotal = mutableListOf<Double>()
9     // варіант 1 - застосувати map() з функцією
10    // перетворення в єдиний список
11    quartalRevenue.map {
12        for(i in it) {
13            mutableTotal.add(i)
14        }
15    }
16
17    // варіант 2 - використати flatMap
18    val total = quartalRevenue.flatMap { it }
19
20    val mutableAvg = mutableTotal.average()
21    println(mutableAvg)
22    val avg = total.average()
23    println(avg)
24 }

```

Аналогічно даний метод застосовується для мепів. Тоді можна отримати як список ключів через it.key, так і список значень, записуючи it.value



**Завдання**

Реалізуйте аналогічну програму для усереднення помісячних температур за 4 роки. Окремо відберіть всі температури, які перевищують 15°C.

Загальний перелік операцій з колекціями в функціональному стилі такий:

**Агрегатні операції**

**Операція any.** Повертає true, якщо принаймні один елемент відповідає заданому предикату

```

val list = listOf(1, 2, 3, 4, 5, 6)
println(list.any { it % 2 == 0 }) // true
println(list.any { it > 8 })     // false

```

**Операція all.** Повертає true, якщо всі елементи відповідають заданому предикату:

```
println(list.all { it < 8 })           // true
println(list.all { it % 2 == 0 })     // false
```

Операція *count*. Повертає кількість елементів, які відповідають заданому предикату:

```
println(list.count { it % 2 == 0 })    // 3
```

Операції *fold* та *foldRight*. Операція *fold()* виконує накопичення, розпочинаючи з початкового значення та застосовуючи операцію над усією колекцією від першого до останнього її елементу. Операція *foldRight()* здійснює те ж, що і *fold()*, проте рухається від кінця колекції до початку:

```
println(list.fold(4) { total, next -> total + next }) // 25
println(list.foldRight(4) { total, next -> total + next })
// 25
```

Операції *forEach* та *forEachIndexed*. Виконують задану операцію по чергово до всіх елементів колекції:

```
list.forEach { println(it-1) }
list.forEachIndexed { index, value ->
    println("У позиції $index міститься значення $value") }
```

Остання інструкція сформує наступний вивід:

```
У позиції 0 міститься значення 1
У позиції 1 міститься значення 2
У позиції 2 міститься значення 3
У позиції 3 міститься значення 4
У позиції 4 міститься значення 5
У позиції 5 міститься значення 6
```

Операції *min/minBy* та *max/maxBy*. Операція *min()* повертає найменший елемент колекції або *null*, якщо в колекції немає елементів. Операція *minBy()* дозволяє знаходити мінімум результатів виконання заданої функції над елементами колекції або *null*, якщо колекція порожня. Аналогічні дії пропонують операції *max/maxBy*, проте для знаходження максимуму:

```
println(list.min())           // 1
println(list.minBy { -it * it }) // 6
println(list.max())           // 6
println(list.maxBy { -it * it }) // 1
```



*Операція none.* Повертає true, якщо в колекції немає елементів, які відповідають заданому предикату:

```
println(list.none { it % 7 == 0 }) // true
```

*Операція sumBy.* Повертає суму всіх значень, утворених перетворюючою функцією з елементів колекції

```
println(list.sumBy { it % 2 }) // 3
```

### **Операції фільтрування**

*Операції drop/dropWhile/dropLast/dropLastWhile.* Операція drop повертає список, що містить всі його елементи за винятком перших n. У разі потреби відбирання перших елементів відповідно до заданого предикату слід використовувати dropWhile. Якщо буде потрібно здійснити аналогічні дії, проте з кінця колекції, застосовуйте dropLast та dropLastWhile:

```
println(list.drop(4)) // [5,6]
println(list.dropWhile { it < 3 }) // [3,4,5,6]
println(list.dropLast(2)) // [1,2,3,4]
println(list.dropLastWhile { it > 4 }) // [1,2,3,4]
```

*Операції take/takeWhile/takeLast/takeLastWhile.* Працює аналогічно до попередніх операцій, проте повертає не решту колекції, а відібрану її частину:

```
println(list.take(2)) // [1,2]
println(list.takeWhile { it < 3 }) // [1,2]
println(list.takeLast(2)) // [5,6]
println(list.takeLastWhile { it > 3 }) // [4,5,6]
```

*Операції filter/filterNot/filterNotNull.* Результатом операції filter є список з усіх елементів, які відповідають заданому предикату. У разі потреби відбору елементів, які йому не відповідають, застосовують операцію filterNot. Часто критерієм відбору є відсутність null-елементів, тому мова Kotlin також пропонує операцію filterNotNull:

```
println(list.filter { it % 2 == 0 }) // [2,4,6]
println(list.filterNot { it % 2 == 0 }) // [1,3,5]
val listWithNull = listOf(null, 1, 2, 3, null, 4, 5, 6)
println(listWithNull.filterNotNull()) // [1,2,3,4,5,6]
```

*Операція slice.* Повертає список елементів згідно з обраними індексами:

```
println(list.slice(listOf(1, 3, 4))) // [2,4,5]
```

### **Операції відображення**

Операції *map/mapIndexed/mapNotNull*. Результатом операції *map* буде список, утворений шляхом застосування перетворюючої функції до кожного елементу початкової колекції. Якщо потрібно паралельно працювати з індексами цієї колекції, корисною буде операція *mapIndexed()*. Перетворююча функція може повертати *null*-значення, тому мова Kotlin також пропонує більш строге відображення – *mapNotNull*:

```
println(list.map { it * 2 }) // [2,4,6,8,10,12]
println(list.mapIndexed { index, it
    -> index * it }) // [0,2,6,12,20,30]
val listWithNull = listOf(null, 1, 2, 3, null, 4, 5, 6)
println(listWithNull.mapNotNull {
    it?.times(2) }) // [2,4,6,8,10,12]
```

Операція *flatMap*. Здійснює сплюснення (flattening) багатовимірної колекції, тобто перетворення її в одновимірну. Для демонстрації сформуємо двовимірний список (список списків) та сплуснемо його в стандартний список:

```
println(list.map { listOf(it, it + 1) })
// [[1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7]]
println(list.flatMap { listOf(it, it + 1) })
// [1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7]
```

Операція *groupBy*. Повертає меп з елементів початкової колекції, погрупованих згідно з результатами виконання заданої функції:

```
println(list.groupBy {
    if (it % 2 == 0) "парні" else "непарні"
}) // {непарні=[1, 3, 5], парні=[2, 4, 6]}
```

### **Операції з елементами**

Операція *contains*. Повертає *true*, якщо елемент знайдено в колекції:

```
println(list.contains(2)) // true
```

Операції *elementAt/elementAtOrElse/elementOrNull*. Перша повертає елемент за переданим індексом або викидає виняток *IndexOutOfBoundsException*, якщо цей індекс знаходиться за межами колекції. Операція *elementAtOrElse* дозволяє додатково задати функцію, яка за умовчанням буде виконуватись при

виникненні винятку `IndexOutOfBoundsException`. Остання операція в переліку передбачає для цієї ж ситуації повернення `null`-значення:

```
println(list.elementAt(1)) // 2
println(list.elementAtOrElse(10, { 2 * it })) // 20
println(list.elementAtOrNull(10)) // null
```

Операції `first/firstOrNull`. Повертають перший елемент, який відповідає заданому предикату. У випадку відсутності такого операція `first` згенерує `NoSuchElementException`, а операція `firstOrNull` поверне `null`-значення:

```
println(list.first { it % 2 == 0 }) // 2
println(list.firstOrNull { it % 7 == 0 }) // null
println(list.first { it%7 == 0 }) // NoSuchElementException
```

Операції `indexOf/indexOfFirst/indexOfLast`. Результатом є знаходження індексу елемента із заданим значенням (`indexOf`) або предикатом (`indexOfFirst`, `indexOfLast`). Операції `indexOf/indexOfFirst` повертає індекс першого знайденого елемента або -1 за умови його відсутності. Операція `indexOfLast`, очевидно, - індекс останнього знайденого елемента або -1, якщо такий елемент не було виявлено:

```
println(list.indexOf(4)) // 3
println(list.indexOfFirst { it % 2 == 0 }) // 1
println(list.indexOfLast { it % 2 == 0 }) // 5
```

Операції `last/lastIndexOf/lastOrNull`. Перша операція з переліку дозволяє отримати останній елемент колекції, який відповідає заданому предикату. Операція `lastIndexOf` працює аналогічно до `indexOfLast`, проте замість предикату приймає значення шуканого елемента. Операція `lastOrNull` подібна до `firstOrNull`, проте повертає індекс останнього знайденого елемента відповідно до переданого предикату:

```
val listRepeated = listOf(2, 2, 3, 4, 5, 5, 6)
println(listRepeated.last { it % 2 == 1 }) // 5
println(listRepeated.lastOrNull { it % 7 == 0 }) // null
println(listRepeated.lastIndexOf(5)) // 5
```

Операції `single/singleOrNull`. Повертають єдиний елемент, який відповідає заданому предикату. У випадку застосування `single` викидається виняток, якщо кількість відповідних елементів у колекції не дорівнює 1. У разі використання `singleOrNull` в цій ситуації отримуємо `null`-значення:

```
val list = listOf(1, 2, 3, 4, 5, 6)
println(list.single { it % 5 == 0 })           // 5
println(list.singleOrNull { it % 7 == 0 })     // null

val listRepeated = listOf(2, 2, 3, 4, 5, 5, 6)
println(listRepeated.single { it % 2 == 1 })
// IllegalArgumentException
```

### **Генеруючі операції**

*Операція partition.* Розбиває початкову колекцію на пару колекцій, сформованих з елементів, для яких заданий предикат набуде значень true і false відповідно:

```
println(list.partition { it % 2 == 0 }) // ([2,4,6],[1,3,5])
```

*Операція plus.* Повертає список, у якому спочатку містяться елементи початкової колекції, а потім елементи переданої колекції. Паралельно з такою ж функціональністю працює оператор «+»:

```
println(list + listOf(7, 8))           // [1, 2, 3, 4, 5, 6, 7, 8]
println(list.plus(listOf(7, 8)))       // [1, 2, 3, 4, 5, 6, 7, 8]
```

*Операція zip.* Повертає список пар, утворених з елементів обох колекцій з однаковими індексами. Сформований список матиме довжину коротшої з колекцій:

```
println(list.zip(listOf(7, 8)))        // [(1, 7), (2, 8)]
```

### **Впорядковуючі операції**

*Операція reversed.* Повертає список з елементами в зворотному порядку:

```
val unsortedList = listOf(3, 2, 7, 5)
println(unsortedList.reversed())       // [5, 7, 2, 3]
```

*Операції sorted/sortedBy/sortedDescending/sortedByDescending.* Операції sorted/sortedDescending дозволяють отримати відсортовану колекцію в порядку зростання чи спадання відповідно. Аналогічно поведуть себе операції sortedBy/sortedByDescending, при цьому дозволяючи задати критерій сортування:

```
println(unsortedList.sorted())          // [2, 3, 5, 7]
println(unsortedList.sortedBy { it % 3 }) // [3, 7, 2, 5]
println(unsortedList.sortedDescending()) // [7, 5, 3, 2]
println(unsortedList.sortedByDescending { it % 3 })
```

// [2, 5, 7, 3]

### Функції області видимості

У мові Kotlin існують вбудовані функції вищого порядку, які працюють не лише з колекціями, а й з довільними об'єктами. У даному контексті під стандартними функціями розуміємо універсальні допоміжні функції зі стандартної бібліотеки мови програмування Kotlin, які приймають в якості параметрів лямбда-вирази для уточнення поведінки. Серед стандартних функцій мови програмування Kotlin найчастіше використовуються наступні: `apply()`, `let()`, `run()`, `with()`, `also()` і `takeIf()` [2]. Загалом дані функції регулюють поведінку переданих їм об'єктів (receiver objects – об'єкти приймачі). Такі функції також називають *функціями області видимості* (scope functions).

По суті, все эти функции делают одно и то же: выполняют блок кода для объекта. Отличие состоит в том, как этот объект становится доступным внутри блока и каков результат всего выражения. Общие случаи использования

- Преобразование объекта - `let()`
- Создание, передача и оценка - `also()`
- Инициализация и выполнение - `run()`
- Инициализировать объект для присвоения - `apply()`

<https://coderoad.ru/48218400/Scope-functions-apply-with-run-also-let-%D0%BE%D1%82%D0%BA%D1%83%D0%B4%D0%B0-%D0%B1%D0%B5%D1%80%D1%83%D1%82%D1%81%D1%8F-%D1%8D%D1%82%D0%B8-%D0%B8%D0%BC%D0%B5%D0%BD%D0%B0>

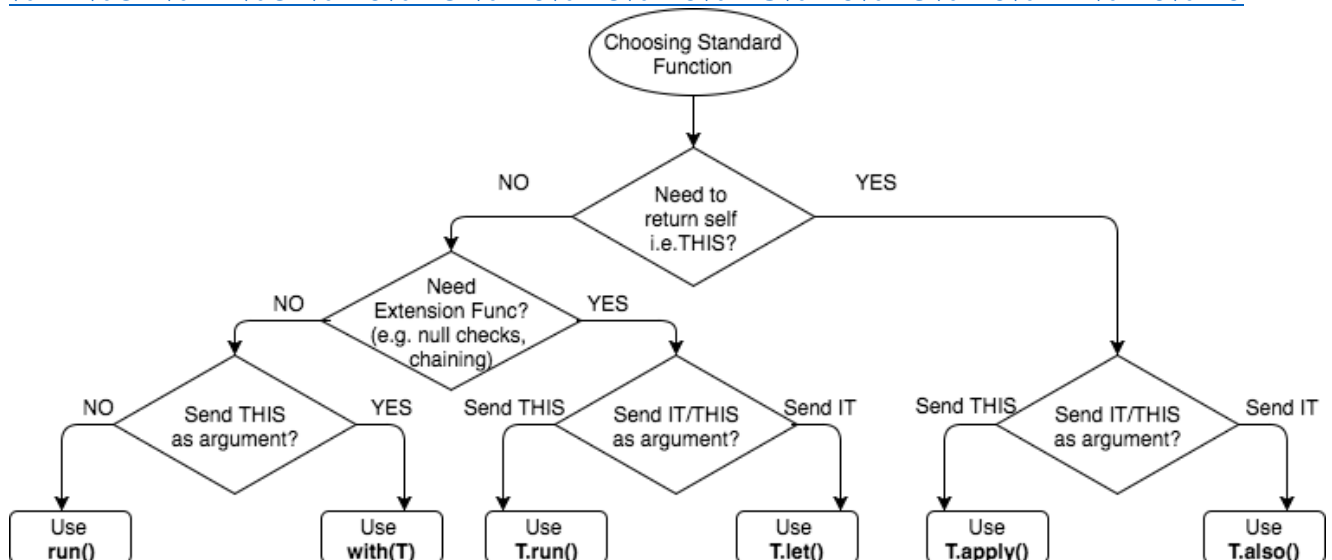


Рис. 2.3. Діаграма вибору

<https://medium.com/@brijesh1794/kotlin-standard-library-let-run-also-apply-with-bb08473d29fd>

<https://proandroiddev.com/kotlin-standard-functions-or-scoping-functions-let-apply-run-also-with-af1d93a444f1>

1) `let` takes the object it is invoked upon as the parameter and returns the result of the lambda expression and also returned as an argument. `let` also check for null; Use `let` whenever you want to define a variable for a specific scope of your code but not beyond.

2) `run` used to override values

3) `also` expressions do some additional processing on the object it was invoked. Unlike `let`, it returns the original object instead of any new return data. Hence the return data has always the same type.

4) `apply` is an extension function on a type. It runs on the object reference into the expression and returns the object reference on completion.

5) `with` is used to change instance properties without the need to call dot operator over the reference every time. it takes an argument.

Для прикладу, функція `let()` дозволяє зручніше працювати з нулабельними типами, зокрема, при порівнянні з ненулабельними значеннями (лістинг 2.8). Припустимо, потрібна перевірка довжини рядка, щоб працювати з текстом довжиною понад 3 символи. Змінна `text` у лістингу 2.8 не є локальною, тому окрема перевірка на рівність `null` не спрацює: значення можна змінити з іншого потоку програми.

#### *Лістинг 2.8. Застосування функції `let()`*

```
1  var text:String? = null
2  fun main() {
3      // варіант 1 - порівняння з compareTo()
4      if(text?.length?.compareTo(3) == 1)
5          println(text)
6
7      // варіант 2 - придушити попередження
8      if(text!!.length > 3)
9          println(text)
10
11     // варіант 3 - функція let()
12     text?.let {
13         if(it.length > 3)
14             println(text)
15     }
16 }
```

Серед запропонованих не спрацює варіант 2, оскільки придушення помилки при компіляції все-одно призведе до виникнення NPE під час виконання програми. Перший і третій варіанти рівноцінні, проте остання версія коду є більш читабельною та зрозумілою.

Функція `with()` повторює функціональність одноіменних менеджерів контексту з мови програмування Python. Виклик багатьох методів об'єкту вимагає постійного вказування назви цього об'єкта для доступу до них. Дещо спрощує код функція `with()`, яка дозволяє встановити об'єкт у якості контексту і викликати методи без згадування назви об'єкта або за допомогою ключового слова `this`



(лістинг 2.7). Зверніть увагу, що з використанням функції `with()` не потрібно дописувати “numbers” перед викликами вбудованих для списку методів у рядках 5-11 лістингу 2.9. Використання `this` у рядку 5 не є обов’язковим.

**Лістинг 2.9. Застосування функції `with()`**

```

1  fun main() {
2      val numbers = mutableListOf<Int>()
3      with(numbers) {
4          for(i in 0..20)
5              this.add((Math.random() * 100).toInt())
6          println(sum())
7          println(average())
8          println(first())
9          println(last())
10         println(min())
11         println(max())
12     }
13 }

```

Функція `with()` є різновидом функції `run()`, проте має інший протокол виклику: об’єкт передається в якості першого аргумента (для `run()` характерний запис, подібний до `apply()`).

Функція `apply()` налаштовує об’єкти для подальшого використання. Для прикладу розглянемо кнопку з графічного інтерфейсу Android-додатку з ідентифікатором `R.id.button`.

```

val button = findViewById<Button>(R.id.button)
button.apply{
    text = "Натисни мене!"
    textSize = 24
    setBackgroundColor(Color.RED)
}

```

Даний код пропонує звернутись до кнопки та налаштувати в коді її вигляд: напис, його розмір та колір заливки фону. Принцип роботи функції аналогічний до функції `with()`, проте на відміну від неї, `apply()` повертає переданий об’єкт. Інколи таку поведінку називають *обмеженням відносної області видимості (relative scoping)*, оскільки всі налаштування стосуються тільки переданого об’єкта. Таким чином, загальна синтаксична структура розглянутих функцій наступна:

```

val r: R = T().run { this.foo(); this.toR() }
val r: R = with(T()) { this.foo(); this.toR() }
val t: T = T().apply { this.foo() }
val t: T = T().also { it.foo() }
val r: R = T().let { it.foo(); it.toR() }

```

Визуальная заметка, чтобы показать различия:

	returns what?	How to access the receiver object?
<pre> run main() {     val citizen1: Citizen? = Citizen( name: "Hans", age: 30, residence: "Berlin")     val citizen2 = Citizen( name: "Tom", age: 24, residence: "Washington")      // Info 📢 There are 6 scoping functions : T.run, T.let, T.apply, T.also, with, run Functions     // T is the receiver object      // 1 - T.run - this, nothing -&gt; Lambda result - returns anything     val total: Int? = citizen1.run { this: Citizen         println("T.run - \${this.age} - \$age")         //println("The receiver string length: \${this.length}") // does the same         this.name.length + age + this.residence.length * run     }     println("return value of run : \$total") </pre>	returns anything	- this - directly
<pre> // 2 - T.let - it -&gt; Lambda result - returns anything val total2: Int = citizen1.let { it: Citizen     println("T.let - \${it.age}")     it.age + it.residence.length + it.name.length * let } println("return value of let : \$total2") </pre>	returns anything	- it
<pre> // 3 - T.apply - this or nothing -&gt; Context object - returns self(receiver object) val id: Int? = User().apply { this: User     id = 100     this.id = 200     println("T.apply - The receiver object is citizen4. id : \$id - id : \${this.id}") }.id </pre>	returns self(receiver object)	- this - directly
<pre> // 4 - T.also - it -&gt; Context object - returns self(receiver object) val citizenNew: Citizen = citizen2.also { it: Citizen     it.age = 40 } println("T.also returns T - name : \${citizenNew.age} + citizenNew.age") </pre>		- it
<pre> // 5 - with - this or nothing -&gt; Context object - returns anything val age: Int = with(citizen2) { this: Citizen     println("\$name - \$age \$residence")     age = this.age + age     residence = "Florida"     age++ * with } println("\${citizen2.name} - \${citizen2.age} - \$age - \${citizen2.residence}") </pre>	returns anything	- this - directly
<pre> // 6 - run - returns anything var mood = "I am sad" var mood2: String = run {     val mood = "I am happy"     println(mood) // I am happy     "I am excited" * run } println("mood : \$mood - mood2 : \$mood2") // I am sad - I am excited </pre>	returns anything	-

У контексті функціонального програмування також існує таке поняття, як **замикання (closure)**. Це функція, яка має доступ до змінних та параметрів, визначених за межами її області видимості. Для Java 8 така можливість нехарактерна, проте мова Kotlin дозволяє створювати подібні конструкції, як у лістингу 2.10. Змінна `res` всередині лямбда-виразу не є локальною для цього виразу, а отримується ззовні та після виконання дій отримує нове значення.

**Лістинг 2.10. Демонстрація роботи замикання**

```

1 fun main() {
2     fun addInts(x: Int, y: Int, operation: (Int, Int) -> Unit) {
3         operation(x, y)
4     }
5
6     var res = 0
7     addInts(5, 12) {a, b -> res = a + b}
8     println(res)
9 }

```

Цікавим наслідком такої поведінки є формування в функціональному коді конструкцій, подібних до об'єктів з об'єктно-орієнтованого програмування. Розглянемо в якості об'єкта деякого домашнього улюбленця, який має параметр

ситості (satiety) та три дії: їсти, гуляти та спати (лістинг 2.11). Створення перелічення (enum) доступних варіантів поведінки дозволяє застосовувати функції в потрібних випадках за допомогою оператора when та викликів у рядках 15-17.

**Лістинг 2.11. Створення аналога об'єкта на основі замикань**

```

1  enum class Action {feed, walk, sleep}
2
3  val pet = { action: Action ->
4      var satiety = 10
5      when (action) {
6          Action.feed -> { food: Int -> satiety += food;
7                          println(satiety) }
8          Action.walk -> { run: Int -> satiety -= 3*run;
9                          println(satiety) }
10         Action.sleep -> { time: Int -> println(satiety) }
11     }
12 }
13
14 fun main() {
15     pet(Action.feed) (5)
16     pet(Action.sleep) (4)
17     pet(Action.walk) (1)
18 }

```

Використання функцій вищого порядку може мати свої недоліки: кожна функція є об'єктом із замиканням, тому виділення пам'яті для функціональних об'єктів, класів та віртуальних викликів буде досить значним. У багатьох випадках можна позбавитись таких даремних витрат, вбудовуючи лямбда-вирази. Якщо оголосити функцію з ключовим словом inline, то компілятор «розгорне» цю функцію в послідовний код.

```

inline fun printSomething(action: () -> String) {
    print(action())
}

```

Економія полягає в тому, що для переданої в функцію лямбди не буде створено анонімний клас, тобто код повністю вбудовується в те місце, звідки була викликана функція. Також через вбудовування для функції не створюється повноцінний об'єкт. Проте кількість методів з точки зору компілятора не змінюється.

Якщо потреби вбудовувати лямбду немає, слід застосувати ключове слово `noinline`:

```

inline fun printSomething(noinline action: () -> String) {
    print(action())
}

```

Если вы захотите вызвать inline лямбду внутри другой лямбды, то для этого вам необходимо указать ключевое слово `crossinline` Пример:

```

inline fun printSomething(crossinline action: () -> String) {
    doSomething {
        action()
    }
}

```

}

Правило простое: если ваш код множество раз вызывает функцию, принимающую лямбду в качестве аргумента, лучше сделать ее inline-функцией. Также запомните следующие вещи:

- Kotlin 1.1 позволяет применять ключевое слово inline также к полям, в которых используются геттеры и сеттеры;
- ключевое слово return, вызванное из лямбды, переданной inline-функции, будет возвращать не из лямбды или inline-функции, а из функции, вызвавшей inline-функцию;

#### INLINE-функції

<https://kotlinlang.ru/docs/reference/inline-functions.html>

<https://tech-geek.ru/inline-function-kotlin/>

[https://polis-mail-ru.github.io/2019-android/10\\_kotlin/101\\_features/](https://polis-mail-ru.github.io/2019-android/10_kotlin/101_features/)

Таким чином, навіть без використання класів створюється змінна-об'єкт зі своїми характеристиками та поведінкою. У поєднанні з функціями вищого порядку це дозволяє переписати об'єктно-орієнтований код у функціональному стилі. Також для розробника доступні синтаксичні можливості створення власних функцій вищого порядку.

### Створення власних функцій вищого порядку

Функції вищого порядку приймають або повертають інші функції. Тому спочатку розглянемо функцію, яка прийматиме лямбда-вираз (анонімну функцію) в якості параметра (лістинг 2.12). Для демонстрації реалізуємо декорування навколо переданого тексту за допомогою символу '\*'. Зверніть увагу, що оператор return не застосовується в анонімній функції для повернення декорованого рядка. У той же час, рядки 3-8 можна винести в окрему функцію, що може зробити код дещо зручнішим.

#### Лістинг 2.12. Власна функція вищого порядку

```
1 fun main() {
2     val decoratedText = decorate("Привіт, світ!") {
3         val length = it.length + 4
4         var result = ""
5         for(i in 0 until length) { result += "*" }
6         result += "\n* " + it + " *\n"
7         for(i in 0 until length) { result += "*" }
8         result
9     }
10    println(decoratedText)
11 }
12 fun decorate(str: String, decor: (String)->String):
13     String {
14     return decor(str)
15 }
```

При запуску коду з лістингу 2.12 на екран буде виведено наступний текст:

```
*****
* Привіт, світ! *
*****
```

За допомогою функцій-розширень (extension function) можна доповнювати функціональність вбудованих та власних класів. Наприклад, таким чином було введено метод isEmpty() для текстових рядків. Якщо натиснути колесом миші на виклику цього метода, можна побачити його внутрішню реалізацію:

```
@kotlin.internal.InlineOnly
public inline fun CharSequence.isEmpty(): Boolean =
length > 0
```

Основне, на що слід звернути увагу, – прив'язка функції до класу CharSequence. Саме так визначаються функції-розширення. Розглянемо складніший приклад – реалізацію аналога функції with() в лістингу 2.13.

*Лістинг 2.13. Власна реалізація функції with()*

```
1 fun main() {
2     val obj = mutableMapOf<Int, String>(
3         1 to "Перший",
4         2 to "Другий",
5         3 to "Третій",
6         4 to "Четвертий"
7     )
8     customWith(obj) {
9         for (i in obj)
10            println("${i.key}: ${i.value}")
11    }
12 }
13 inline fun<T, R> customWith(obj: T, op: T.() -> R): R {
14     return obj.op()
15 }
```

Зверніть увагу на те, що функція with() має працювати з об'єктами довільного типу, тому для функції-розширення визначені параметризовані типи T та R, що позначають тип об'єкта та тип значення, яке повертається відповідно. Звідси, отримуємо тип функції op() – операції, передбаченої для об'єкта типу T – T.() -> R.



#### Завдання

Порівняйте реалізацію з лістинга 2.13 та вбудовану реалізацію функції with(). У чому полягають відмінності між ними?

**Практичні завдання**

1. *0,3 бала* Перетворіть список температур у градусах Фаренгейта на аналогічний список у градусах Цельсія.
2. *0,4 бала* Маючи список з 10 GPS-координат, визначте пару найближчих положень. Відстань між двома розташуваннями слід визначати за [формулою гаверсинуса](#).
3. *0,4 бала* Створіть хешмеп з абонентами та їх номерами мобільних телефонів. Використовуючи функціональний стиль програмування, виведіть списки абонентів відповідно до [коду](#) їх мобільного оператора.
4. *0,4 бала* Припустимо, що у зв'язку з підвищенням мінімальної заробітної плати в країні, всі робітники підприємства мають отримувати не менше заданої суми грошей у місяць. Напишіть програму, яка для переліку робітників з їх заробітними платами повинна підняти оплату праці тим, у кого на даний момент вона не перевищує мінімальної, а для працедавця – вивести суму додаткових витрат на фонд оплати праці.
5. *0,6 бала* Виконайте власні дослідження з приводу порівняння швидкодії списків та послідовностей. Проаналізуйте вже існуючі порівняння, на зразок таких, як [тут](#), [тут](#) і [тут](#).
6. *0,7 бала* Перетворіть нижче наведений код з імперативного стилю програмування у функціональний

```
fun findByTitle(query: String, collection: MutableList<Movie>): List<Movie>{  
    var results: MutableList<Movie> = mutableListOf()
```

```
    do {  
        val movie = collection.removeAt(0)  
        if (movie.title.contains(query)){  
            results.add(movie)  
        }  
    }  
    while (collection.size > 0)
```

```
    return results
```

```
}
```

Зберігайте проміжні коди, утворені в процесі переробки.

**Для виконання даної задачі слід:**

- 1) Створити клас, який міститиме об'єкт-помічник з даним методом всередині.
- 2) Перейти до процедурного програмування з прибраними побічними ефектами (без зміни стану за межами функції): винести оператор if в окрему «чисту» функцію. Ланцюговий виклик movie.title.contains(query) розіб'ється на окремі методи:
  - а. Отримання заголовку фільму: сигнатура (Movie) -> String.



- b. Перевірки вмісту за допомогою стандартного методу `contains()`: сигнатура `(String, String) -> Boolean`.
  - c. Порівняння заголовків та пошук співпадінь `(String, Movie) -> Boolean`.
- 3) Подальше прибирання побічних ефектів:
- a. заміна циклу `do-while` на цикл `for` (перехід до ітератору),
  - b. заміна `MutableList` на `List` та методу списку `add()` на `plus()`. У подальшому буде записана функція вищого порядку для додавання фільму до окремого списку фільмів,
  - c. виділення функції порівняння заголовків у предикат,
  - d. включення предикату в якості аргументу оновленого методу для умовного порівняння. Поаргументно:  
*предикат*: `(String, Movie) -> Boolean`  
*пошуковий запит*: `String`  
*фільм з колекції*: `Movie`  
*додавання в колекцію*: `[Movie] -> Boolean`

У результаті роботи методу повертається або виклик методу додавання фільму в колекцію (`Movie -> Boolean`), або `false` (у подальшому – порожній список фільмів: `(Movie) -> (List<Movie>) -> List<Movie>`).

- e. заміна методу додавання фільму до списку (список передається в якості явного аргументу).
- Мета – оформлення першокласних функцій та функцій вищого порядку.
- 4) Завершення прибирання побічних ефектів. Каррінг отриманих функцій: зведення багатоаргументних функцій до ланцюга одноаргументних. Спочатку застосовується до методу додавання до списку, який тепер прийматиме лише фільм. Мета – задіяти лямбда-вирази.
  - 5) Перетворити всі методи у лямбда-вирази, зберігши їх у відповідні змінні.
  - 6) Замінити код умовного порівняння на базовий метод `Collection.Filter`. У результаті отримається набір лямбда-виразів, орієнтовно – збережених у 5 змінних-функцій:
    - a. пошук за назвою: `(String) -> (MutableList<Movie>) -> List<Movie>`,
    - b. фільтр фільмів: `(Movie) -> Boolean) -> (List<Movie>) -> List<Movie>`,
    - c. порівняння фільмів за назвою: `(String) -> (Movie) -> Boolean`,
    - d. отримання назви: `(Movie) -> String`,
    - e. порівняння назв: `(String) -> (String) -> Boolean`.