Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Розробка програмного забезпечення мобільних пристроїв

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2025

УДК 004.43
Р64

Укладачі: *Недашківський Олексій Леонідович*, д-р техн. наук, доц.
Гусєва Ірина Ігорівна, канд. екон. наук, доц.
Голець Владислав Олександрович
Гнатишин Михайло Степанович
Пироговська Тетяна Володимирівна
Передера Віктор Романович

Рецензенти *Жураковський Б.Ю.,* д-р техн. наук, проф., професор кафедри інформаційних систем та технологій Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського»

Відповідальний редактор *Стативка Ю. І.*, канд. техн. наук, доцент

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 9 від 26.06.2025 р.)
за поданням вченої ради Навчально-наукового інституту атомної та теплової енергетики
(протокол № 12 від 03.06.2025 р.)

Р64 **Розробка програмного забезпечення мобільних пристроїв**. [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці» спец. 121 Інженерія програмного забезпечення / КПІ ім. Ігоря Сікорського ; уклад.: О. Л. Недашківський та ін. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2025. – 316 с.

Навчальний посібник призначений для здобувачів ступеня бакалавр за спеціальністю 121 Інженерія програмного забезпечення освітньої програми «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці» Національного технічного університету України «Київського політехнічного інституту імені Ігоря Сікорського».
У посібнику приділено увагу вивченню мови Kotlin для програмування мобільних застосунків. Навчальний посібник буде також корисний тим, хто цікавиться розробкою програмного забезпечення мовою програмування Kotlin та навчається за іншими спеціальностями галузі знань 12 Інформаційні технології.

УДК 004.43

Реєстр. № НП 24/25-593. Обсяг 10,9 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

ВСТУП

Навчальний посібник «Розробка програмного забезпечення мобільних пристроїв» присвячено мові програмування загального призначення Kotlin від компанії JetBrains, яка призначена насамперед для розробки мобільних застосунків.

Обсяг матеріалу становить необхідний мінімум при підготовці слухачів за освітньою програмою «Інженерія програмного забезпечення кіберфізичних систем в енергетиці» спеціальності 121 Інженерія програмного забезпечення під час вивчення дисципліни «Розробка програмного забезпечення мобільних пристроїв», предметом якої є методи та засоби розробки програмного забезпечення мобільних пристроїв при реалізації алгоритмів та програмних застосунків мовою Kotlin.

Матеріал, викладений в навчальному посібнику, буде корисний при вивченні дисциплін «Кросплатформна розробка мобільних застосунків», «Мульти- та кросплатформне програмне забезпечення», «Крос-платформне програмування» та інших, де використання мови програмування Kotlin є доцільним.

В навчальному посібнику подано загальний та практичний матеріал, опис програмних засобів та приклади створення програм мовою Kotlin, наведені рекомендації для поглибленого вивчення як самої мови програмування так і інструментальних засобів розробки та тестування.

Основним завдання навчального посібника, згідно з вимогами програм зазначених навчальних дисциплін, є формування у студентів після засвоєння кредитного модуля наступних знань та умінь.

Знання:

- основних процесів, фаз та ітерацій життєвого циклу програмного забезпечення;
- фундаментальних концепцій, парадигм і основних принципів функціонування мовних, інструментальних і обчислювальних засобів інженерії програмного забезпечення;
- прийомів застосування інформаційних технологій обробки, зберігання та передачі даних;
- методів та засобів створення мобільних додатків, крос- та мульти-платформного програмування, зокрема, для кібер-фізичних та енергетичних систем;
- основних принципів застосування мови Kotlin;
- структури Kotlin програми;

Уміння:

- вибирати вихідні дані для проектування, керуючись формальними методами опису вимог та моделювання;
- застосовувати інформаційні технології обробки, зберігання та передачі даних;
- організовувати, налаштовувати та програмувати у комп’ютерних мережах;
- аналізувати, вибирати, застосовувати засоби забезпечення інформаційної безпеки, зокрема в енергетиці.
- проєктувати компоненти мобільних застосунків мовою Kotlin;
- застосовувати засоби мови програмування, опису інформаційних ресурсів, специфікацій під час проєктування та створення інформаційних систем;
- програмно реалізувати алгоритми розв'язування задач;
- проєктувати компоненти програмного забезпечення;
- виконувати аналіз коректності програм.

1. МОВА KOTLIN. ПОЧАТОК РОБОТИ

1.1. Мова Kotlin. Перша програма

Kotlin є сучасною, статично типізованою мовою програмування, що швидко розвивається. Мова Kotlin створена і розвивається компанією JetBrains. Kotlin можна використовувати для створення різних програм. Це і програми для мобільних пристроїв на базі операційних систем (ОС) Android та iOS [1-6]. При цьому Kotlin дозволяє писати кросплатформний код, який застосовуватиметься на різних платформах. Це і веб-застосунки, причому як серверні програми, які працюють на стороні сервера (бекенд), так і браузерні клієнтські програми (фронтенд). Kotlin також можна застосовувати для створення десктопних програм, для Data Science і таке інше.

Таким чином, коло платформ, для яких можна створювати програми на Kotlin, надзвичайно широке – це Windows, Linux, Mac OS, iOS та Android.

Найпопулярнішим напрямком, де застосовується Kotlin, є насамперед розробка під ОС Android. Причому настільки популярним, що компанія Google на конференції Google I/O 2017 проголосила мову Kotlin однією з офіційних мов для розробки під Android поряд з Java і C++, а інструменти для роботи з цією мовою були включені до функціоналу середовища розробки Android Studio починаючи з версії 3.0.

Офіційним сайтом мови є **https://kotlinlang.org/**[1] [7], де можна знайти останню та детальну інформацію з мови.

Перша версія мови вийшла 15 лютого 2016 року. Хоча сама розробка мови велася з 2010 року. Поточною версією мови є версія 1.5, яка вийшла 5 травня 2021 року.

Мова Kotlin зазнала впливу багатьох мов: Java, Scala, Groovy, C#, JavaScript, Swift та дозволяє писати програми як у об'єктно-орієнтованому, так і у функціональному стилі. Вона має ясний і зрозумілий синтаксис і є досить легкою для опанування.

Але мова Kotlin – це не просто чергова мова програмування. На сьогоднішній день це ціла екосистема, яку показано на рисунку 1.1.

[1] https://kotlinlang.org/

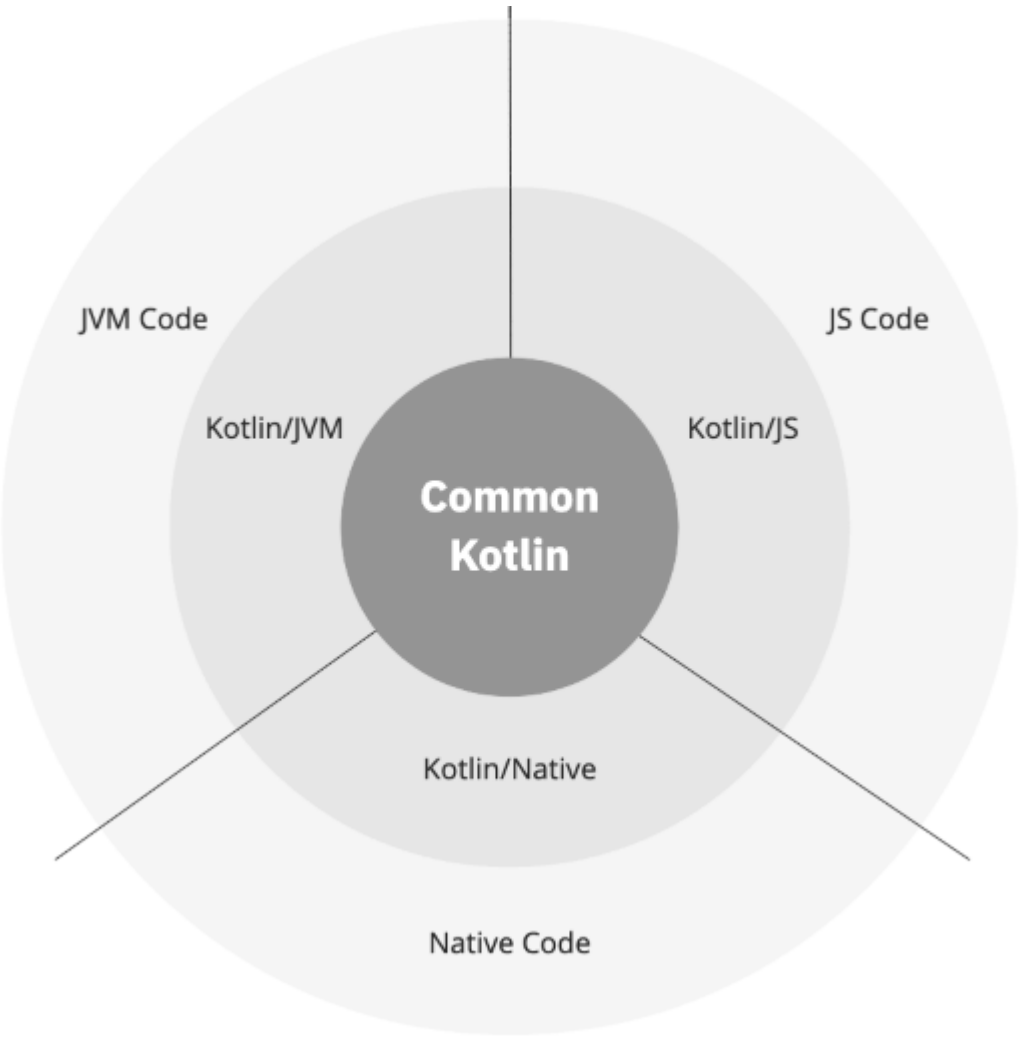

Рисунок 1.1. Екосистема Kotlin

Ядро цієї екосистеми є Common Kotlin, яке включає в себе власне мову, основні бібліотеки та базові інструменти для побудови програм.

Для взаємодії з конкретною платформою існують спеціально призначені для цієї платформи версії Kotlin: Kotlin/JVM, Kotlin/JS та Kotlin/Native. Ці специфічні версії є розширеннями для мови Kotlin зі специфічними для конкретної платформи бібліотеками та інструментами розробки.

У майбутньому вся ця екосистема буде об'єднана в єдину платформу Kotlin Multiplatform, яка зараз перебуває на стадії альфа-версії.

Також варто зазначити, що Kotlin розвивається як відкрита системи (opensource), вихідний код проєкту можна переглянути у репозиторії на github за адресою **https://github.com/JetBrains/kotlin/**[1] [8].

[1] https://github.com/JetBrains/kotlin/

1.1.1. Перша програма на Kotlin

Для написання програмного коду знадобиться текстовий редактор. Це може бути будь-який тестовий редактор, наприклад Notepad++ або Visual Studio Code. Для компіляції програми потрібний компілятор.

Крім того, необхідно встановити JDK (Java Development Kit). Завантажити пакети JDK для встановлення можна із сайту компанії Oracle за посиланням **http://www.oracle.com/technetwork/java/javase/downloads/index.html**[1] [9].

Завантажити компілятор безпосередньо для мови Kotlin можна за адресою **https://github.com/JetBrains/kotlin/releases/latest/**[2] [10]. Внизу сторінки можна знайти загальну версію компілятора, версії компілятора Kotlin/Native для різних операційних систем, а також вихідний код. Отже, завантажимо файл **kotlin-compiler-1.5.0.zip,** як показано на рисунку 1.2.

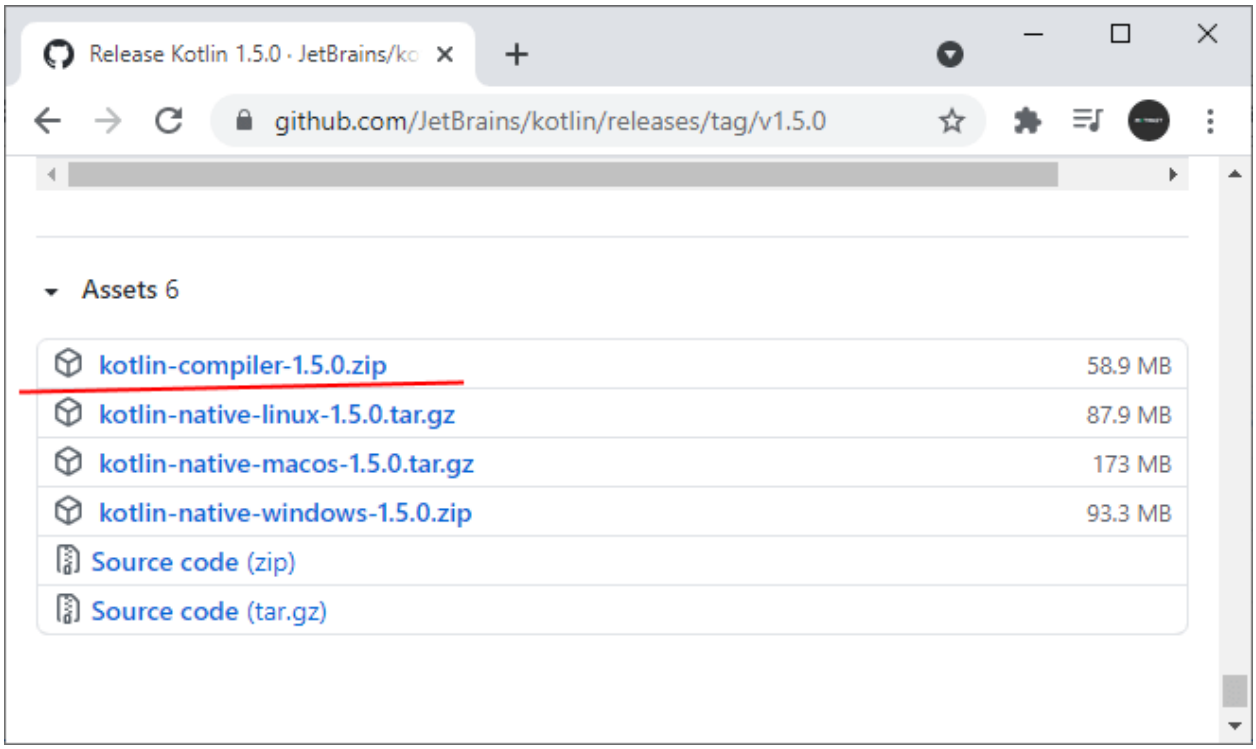

Рисунок 1.2. Вікно завантаження компілятора для мови Kotlin

За вказаною вище адресою можна знайти архів. Завантажуємо та розпакуємо з архіву теку **kotlinc**. У розпакованому архіві, як показано на рисунку 1.3 в теці **bin** можна знайти утиліту **kotlinc**, за допомогою якої буде здійснюватися компіляція:

[1] http://www.oracle.com/technetwork/java/javase/downloads/index.html

[2] https://github.com/JetBrains/kotlin/releases/latest/

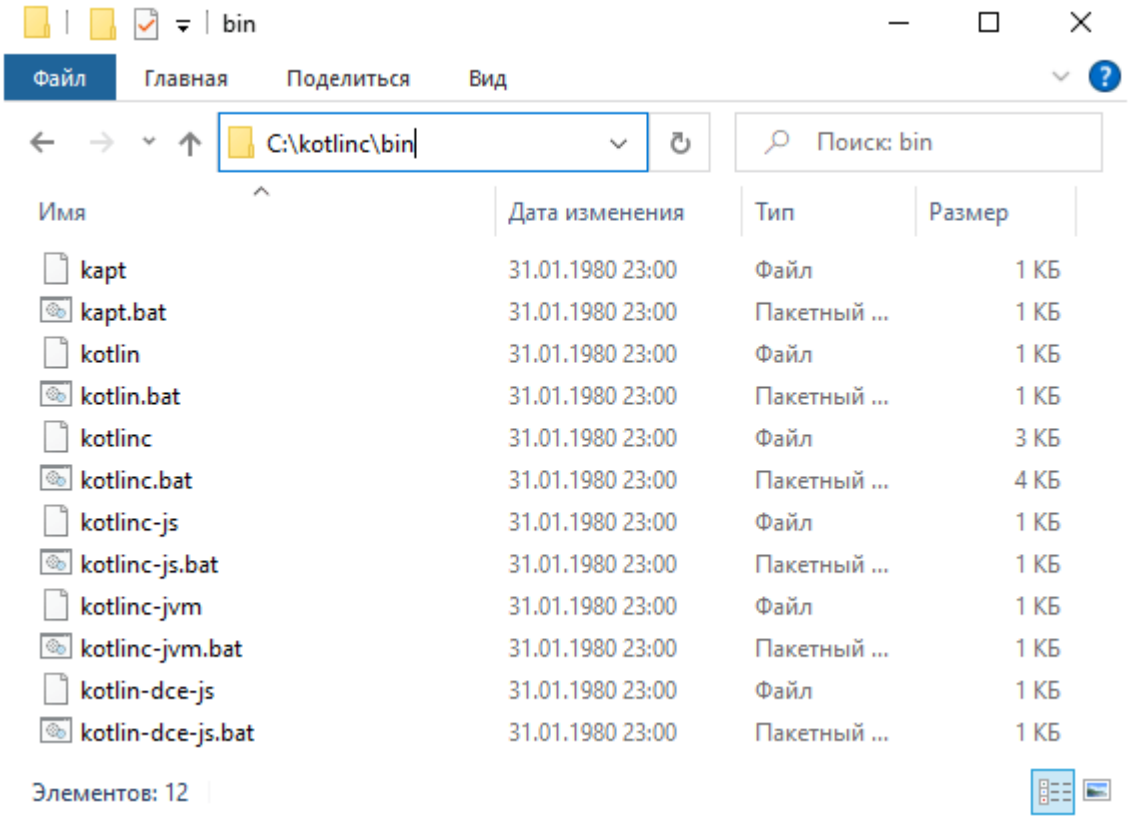

Рисунок 1.3. Вигляд теки, де розташована утиліта kotlinc

Тепер визначимо на жорсткому диску каталог файлів з вихідним кодом, наприклад **c:/kotlin**. У цьому каталозі створимо текстовий файл та перейменуємо його на **app.kt**. Розширення **.kt** – це розширення файлів мовою Kotlin.

Далі запишемо в цьому файлі код, який виводитиме деяке повідомлення на консоль:

```
1   fun main(){
2       println("Hello Kotlin")
3   }
```

Точкою входу в програму Kotlin є функція **main**. Для визначення функції застосовується ключове слово **fun**, після якого йде назва функції – тобто **main**. Ця функція не приймає жодних параметрів, тому після назви функції вказуються порожні дужки.

Далі у фігурних дужках визначаються ті команди, які виконує функція **main**. У наведеному прикладі, як видно з рисунку 1.4, всередині функції **main** виконується інша функція - **println()**, яка виводить деяке повідомлення на консоль.

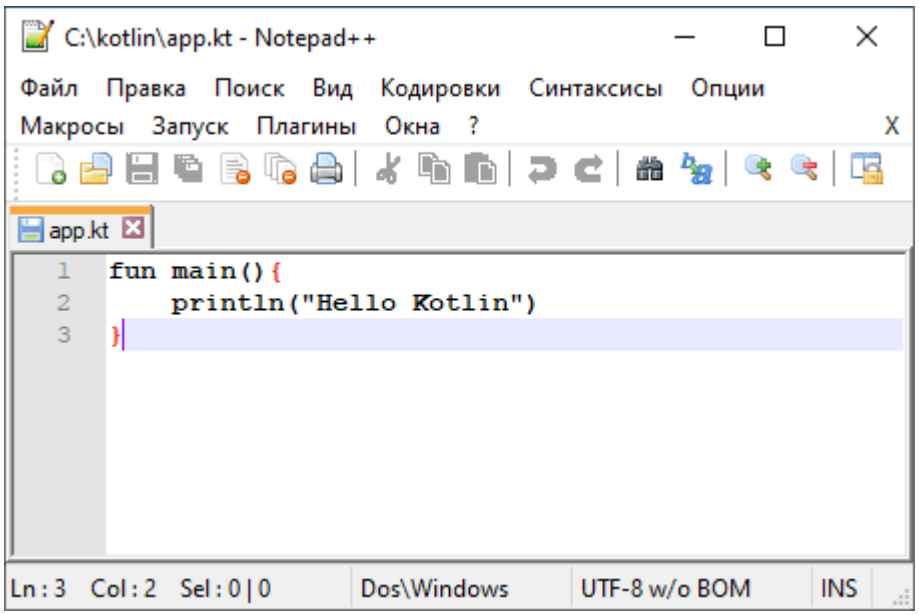

Рисунок 1.4. Вигляд вікна текстового редактора **Notepad++**

Тепер відкриємо командний рядок. Спочатку за допомогою команди **cd** перейдемо до теки, де розташований файл **app.kt**. Потім для компіляції програми вводимо в командному рядку команду, як показано на рисунку 1.5.

```
c:\kotlinc\bin\kotlinc app.kt -include-runtime -d app.jar
```

Рисунок 1.5. Вигляд командного рядка

В наведеному прикладі, компілятору **c:\kotlin\bin\kotlinc** для компіляції передається файл **app.kt**. Щоб не писати повний шлях до компілятора, шлях до нього можна додати у змінну **PATH** в змінних середовища. Далі за допомогою параметра **-include-runtime** вказується, що створюваний файл включатиме середовище Kotlin. А параметр **-d** вказує, як буде називатися створюваний файл програми, тобто це буде **app.jar**.

Після виконання цієї команди буде створено файл **app.jar**. Тепер можна запускати його на виконання. Для цього необхідно ввести команду, як показано на рисунку 1.6.

```
java -jar app.jar
```

Рисунок 1.6. Вигляд командного рядка

В цьому прикладі вважається, що шлях до JDK, встановленому на комп'ютері, прописаний у змінній **PATH** у змінних середовища. Інакше замість **"java"** доведеться писати повний шлях до утиліти **java**.

У результаті при запуску файлу на консолі з’явиться рядок **"Hello Kotlin"**, як показано на рисунку 1.6.

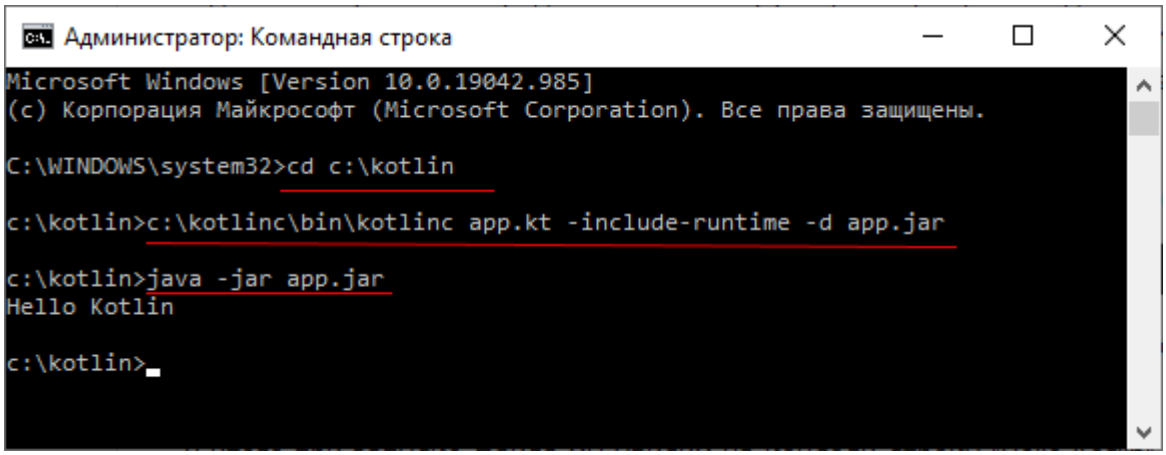

Рисунок 1.6. Результат роботи програми

1.2. Перша програма на Kotlin у IntelliJ IDEA

Для розробки мобільних та інших застосунків мовою Kotlin можна використовувати таке середовище розробки як IntelliJ IDEA від компанії JetBrains. Завантажити його можна за посиланням **https://www.jetbrains.com/idea/download/**[1] [11]. Це середовище доступне як для Windows, так і для MacOS та Linux. Є також безкоштовна версія – Community, і платна – Ultimate. Отже, завантажуємо та встановлюємо безкоштовну версію IntelliJ IDEA Community, або якщо є бажання та відповідні засоби, то можна використовувати і платну версію Ultimate.

1.2.1. Установка IntelliJ IDEA

Для установки IntelliJ IDEA необхідно запустити програму інсталяції, як показано на рисунку 1.7.

[1] https://www.jetbrains.com/idea/download/

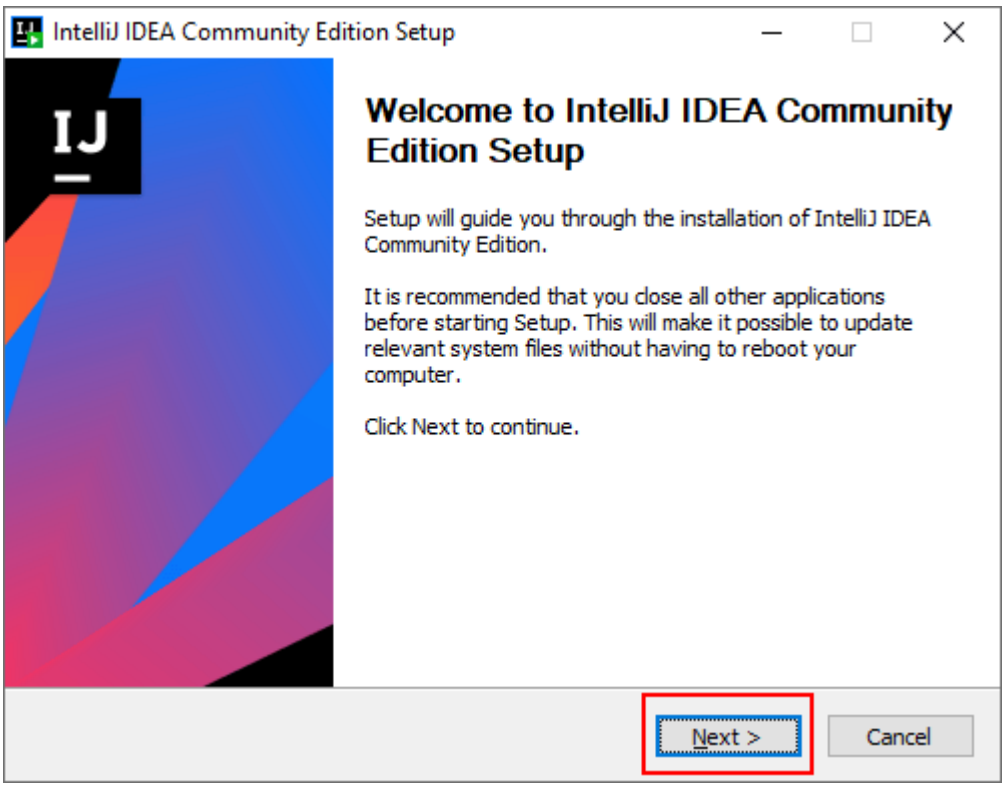

Рисунок 1.7. Вітальне вікно запуску програми інсталяції

У вітальному вікні натискаємо кнопку **Next**. Далі відобразиться шлях, за яким встановлюватиметься середовище розробки, як показано на рисунку 1.8.

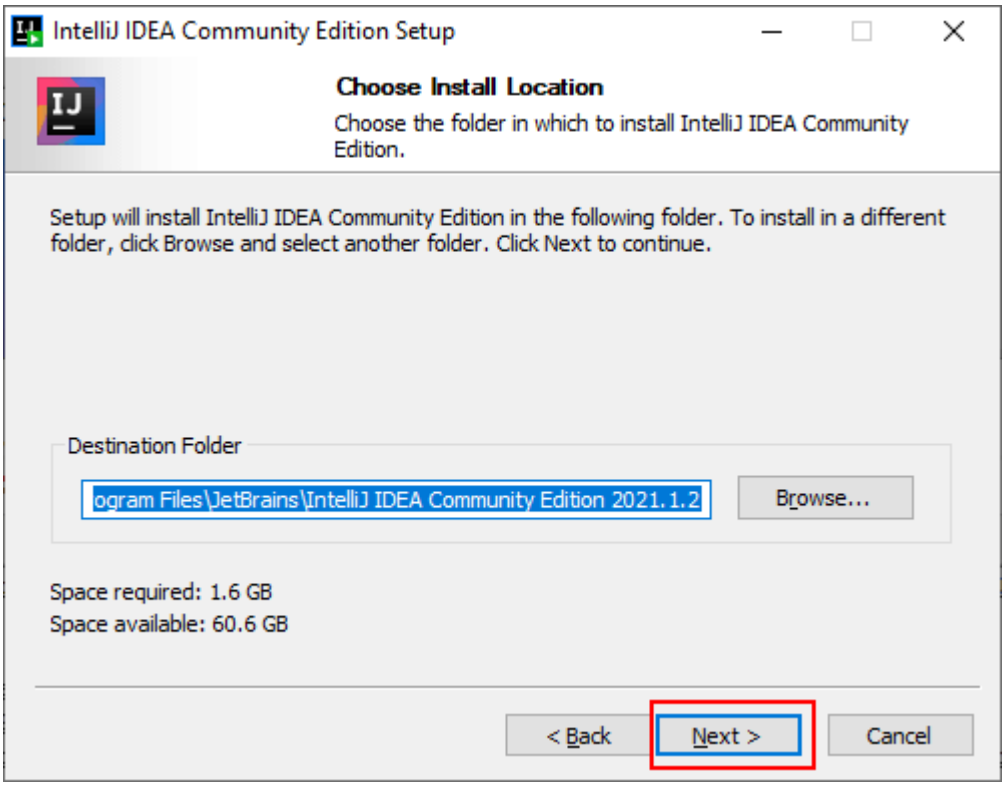

Рисунок 1.8. Вікно вибору розташування середовища розробки

Розташування середовища розробки можна залишити за замовчуванням, а можна змінити. І далі натискаємо кнопку **Next**.

Потім з'явиться вікно, яке показано на рисунку 1.9, деяких налаштувань конфігурації, де можна, наприклад, зв'язати середовище з типами файлів або налаштувати створення іконок середовища на робочому столі. Але на початку просто натискаємо кнопку **Next**.

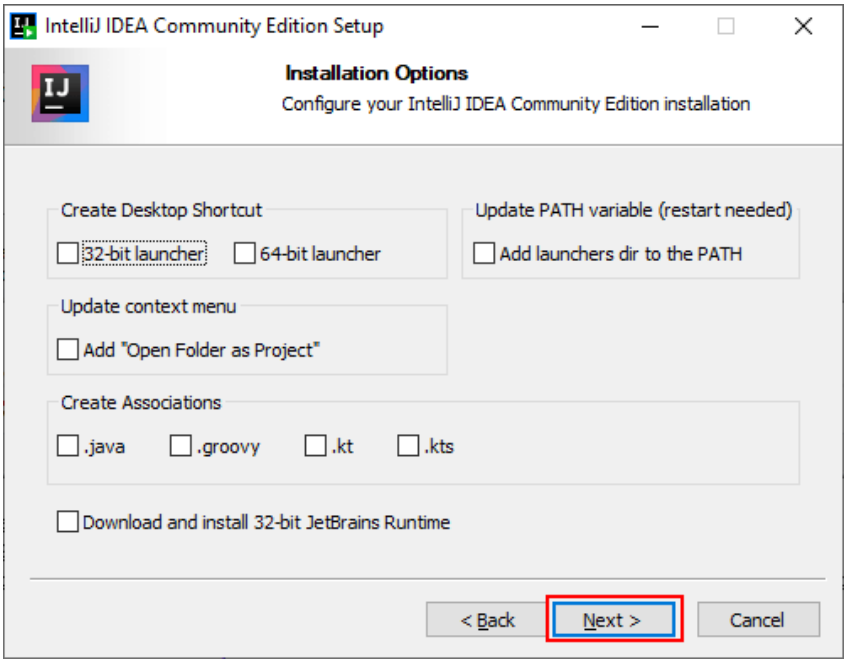

Рисунок 1.9. Вікно налаштування параметрів конфігурації

Далі відкриється вікно, яке показано на рисунку 1.10, для вибору каталогу у меню **Пуск**, де можна буде потім швидко запускати програму.

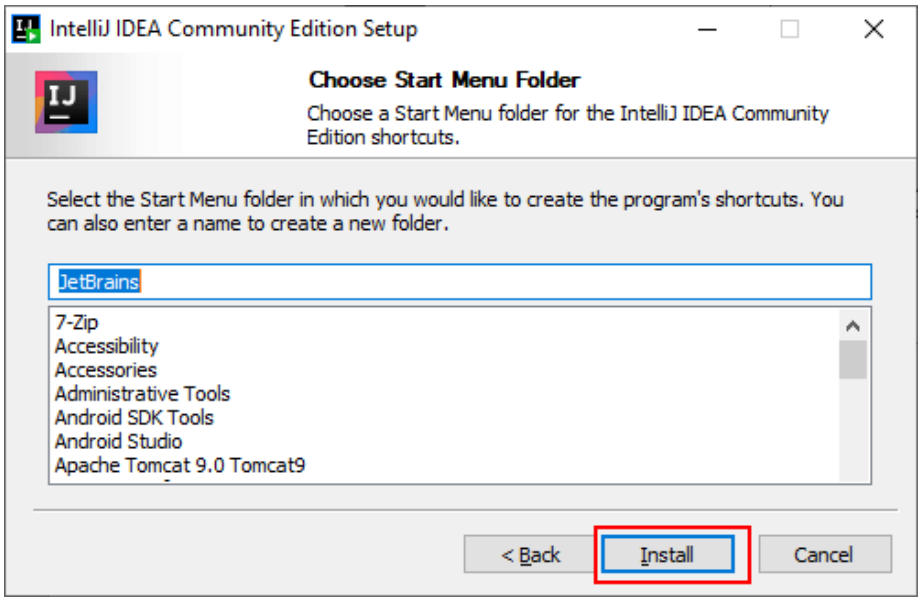

Рисунок 1.10. Вікно вибору каталогу, де буде розташовано програму

Залишимо значення за замовчуванням і натиснемо кнопку **Install**. Процес інсталяції, як показано на рисунку 1.11, буде запущено.

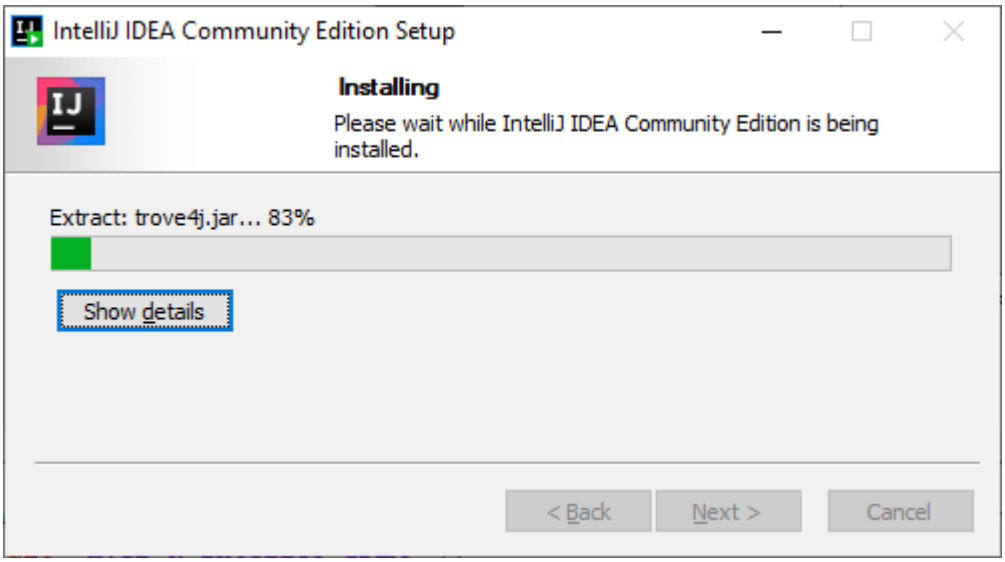

Рисунок 1.11. Вікно, що візуалізує процес інсталяції

Після закінчення інсталяції можна запускати середовище розробки. Для цього позначимо у фінальному вікні пункт **Run IntelliJ IDEA Community Edition** і натиснемо кнопку **Finish**, як показано на рисунку 1.12.

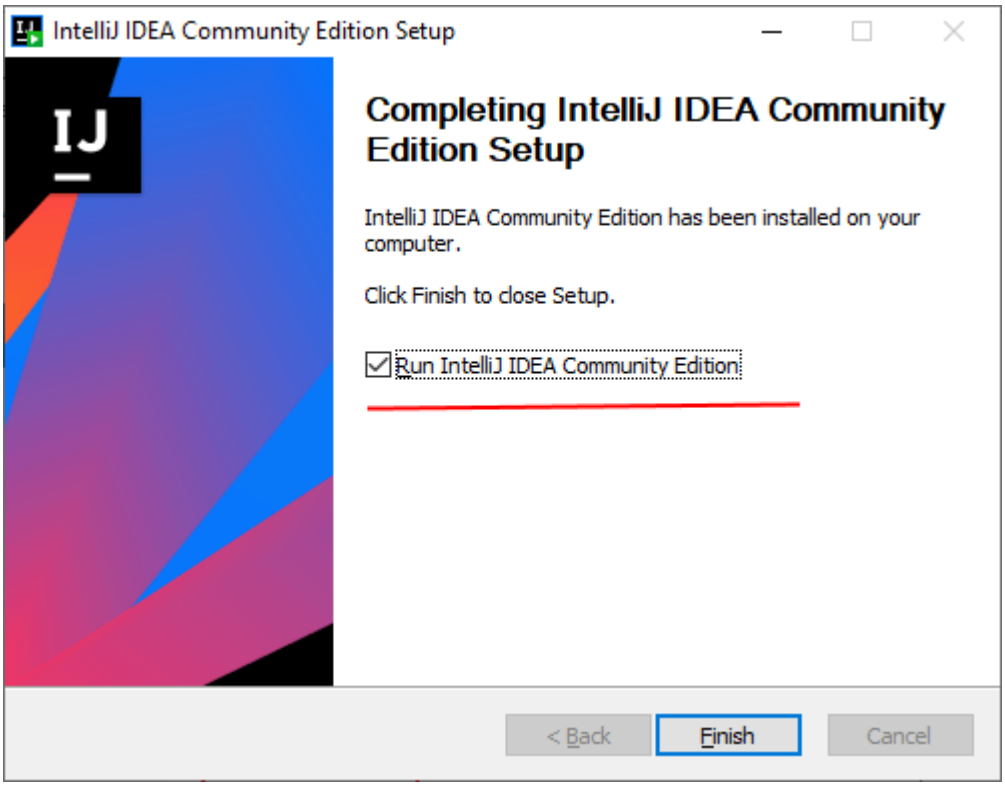

Рисунок 1.12. Вікно завершення інсталяції програми

1.2.2. Створення проєкту

Запустимо IntelliJ IDEA. При цьому повинно відкритися стартове вікно програми, як показано на рисунку 1.13.

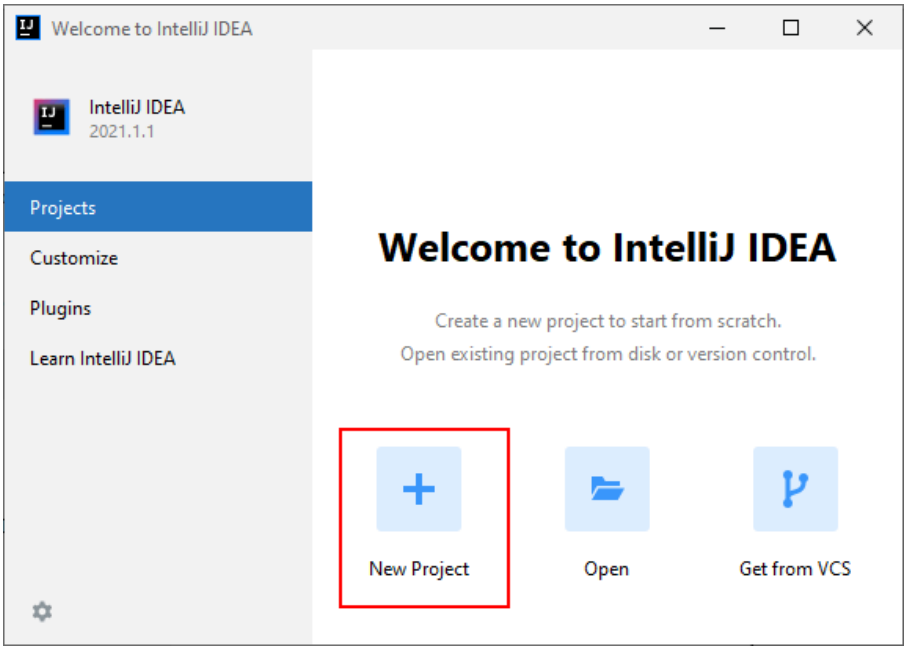

Рисунок 1.13. Стартове вікно програми

Обираємо у ньому пункт **New Project**. Після цього відкриється вікно вибору шаблону проєкту та вибору його налаштувань, як показано на рисунку 1.14.

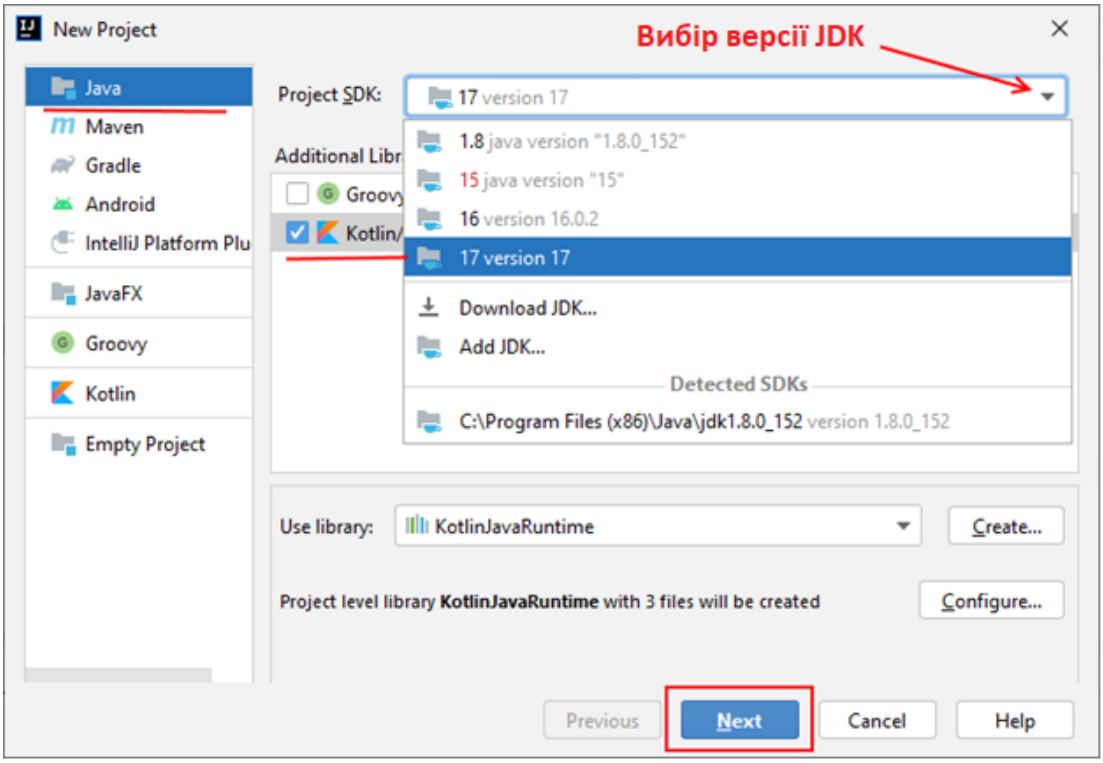

Рисунок 1.14. Вікно вибору шаблону проєкту та вибору його налаштувань

IntelliJ IDEA пропонує низку типів проєктів для створення програм на мові Kotlin. Обираємо найпростіший тип. Отже, ліворуч у вікні створення проєкту виберемо **Java**, а праворуч як шаблон проєкту виберемо **Kotlin/JVM**. Тобто проєкт використовуватиме JVM (віртуальну машину Java). Тому у верхній частині вікна в полі **Project SDK**: встановимо використовувану версію JDK. Для цього потрібно зазначити каталог, де встановлено JDK. І після цього натиснути кнопку **Next**.

Потім з'явиться вікно, яке проказано на рисунку 1.15, для визначення назви проєкту. В цьому прикладі проєкт називатиметься **HelloKotlin**.

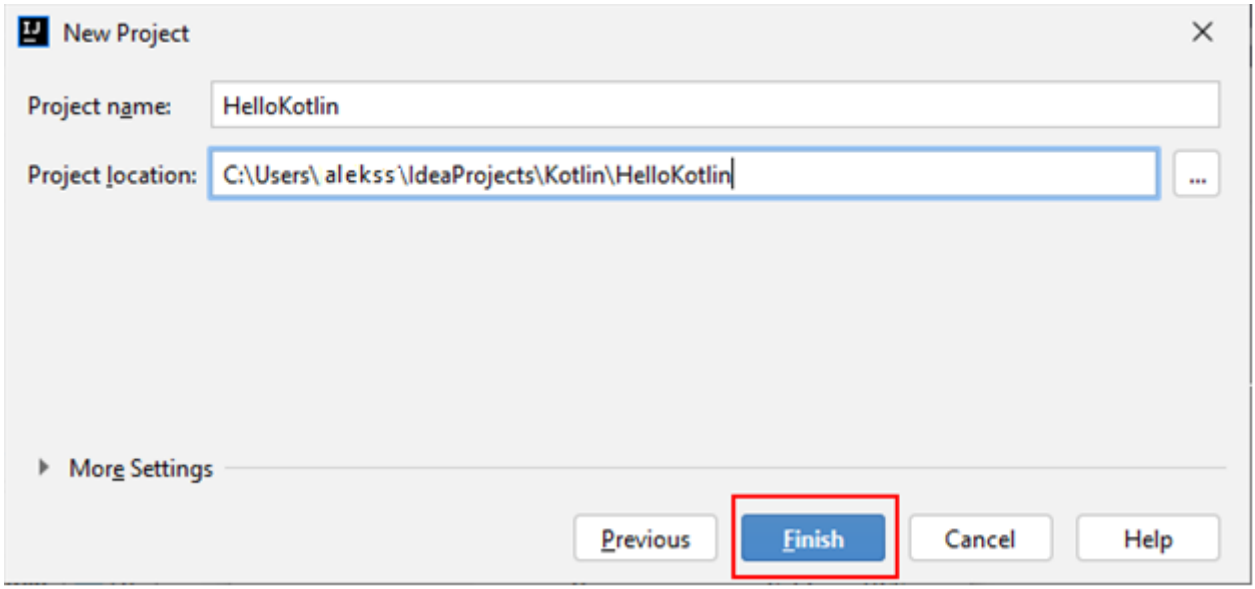

Рисунок 1.15. Вікно визначення імені проєкту

Після натискання на кнопку **Finish** буде створено новий проєкт. За замовчуванням проєкт не містить жодного коду, він порожній. І тому додамо до нього файл із кодом мовою Kotlin. Для цього натиснемо правою кнопкою миші на теку **src** і в контекстному меню, що з'явиться, як показано на рисунку 1.16, обираємо **New -> Kotlin File/Class**.

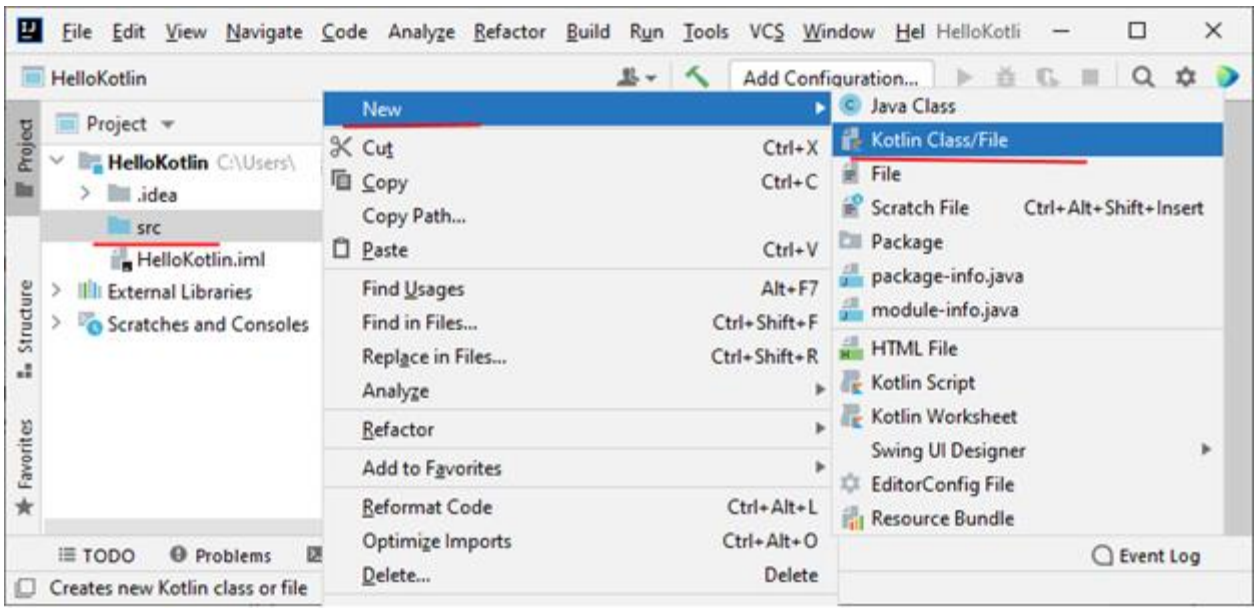

Рисунок 1.16. Вікно додавання файлу мовою Kotlin

Далі назвемо файл, наприклад, **app**, як показано на рисунку 1.17.

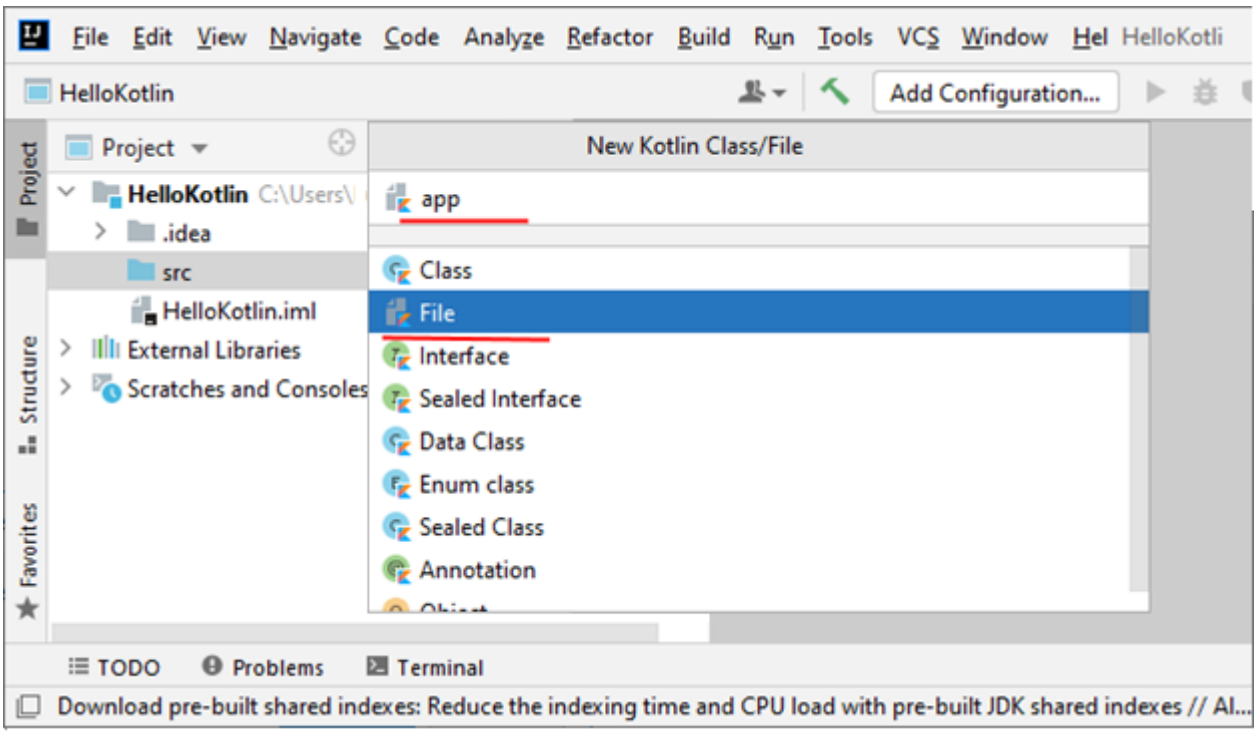

Рисунок 1.17. Вікно введення назви файлу програми

В теку **src** буде додано файл **app.kt** (**kt** – розширення для файлів мовою Kotlin). За замовчуванням цей файл відкривається у центральній частині IntelliJ IDEA. Але файл порожній. Тому додамо в нього наступний код:

```
1   fun main(){
2       println("Hello Kotlin")
3   }
```

Точкою входу в програму Kotlin є функція **main**. Для визначення функції застосовується ключове слово **fun**, після якого йде назва функції – тобто **main**. Ця функція не приймає жодних параметрів, тому після назви функції вказуються порожні дужки.

Далі у фігурних дужках визначаються ті команди, які виконує функція **main**. У наведеному прикладі, як показано на рисунку 1.18, всередині функції **main** виконується інша функція - **println()**, яка виводить деяке повідомлення на консоль.

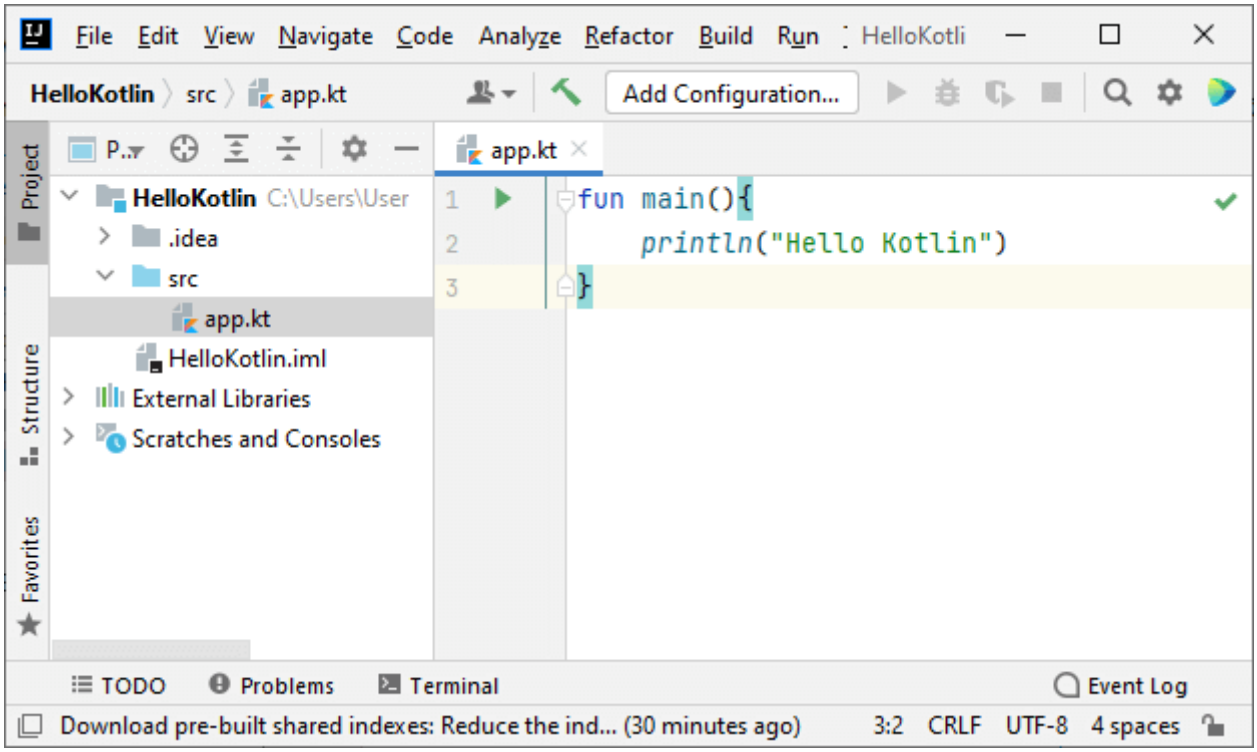

Рисунок 1.18. Вікно ведення коду програми

Запустимо цю програму на виконання. Для цього натиснемо на значок **Kotlin** поряд з першим рядком коду або на назву файлу і обираємо в меню пункт **Run 'AppKt'**, як показано на рисунку 1.19.

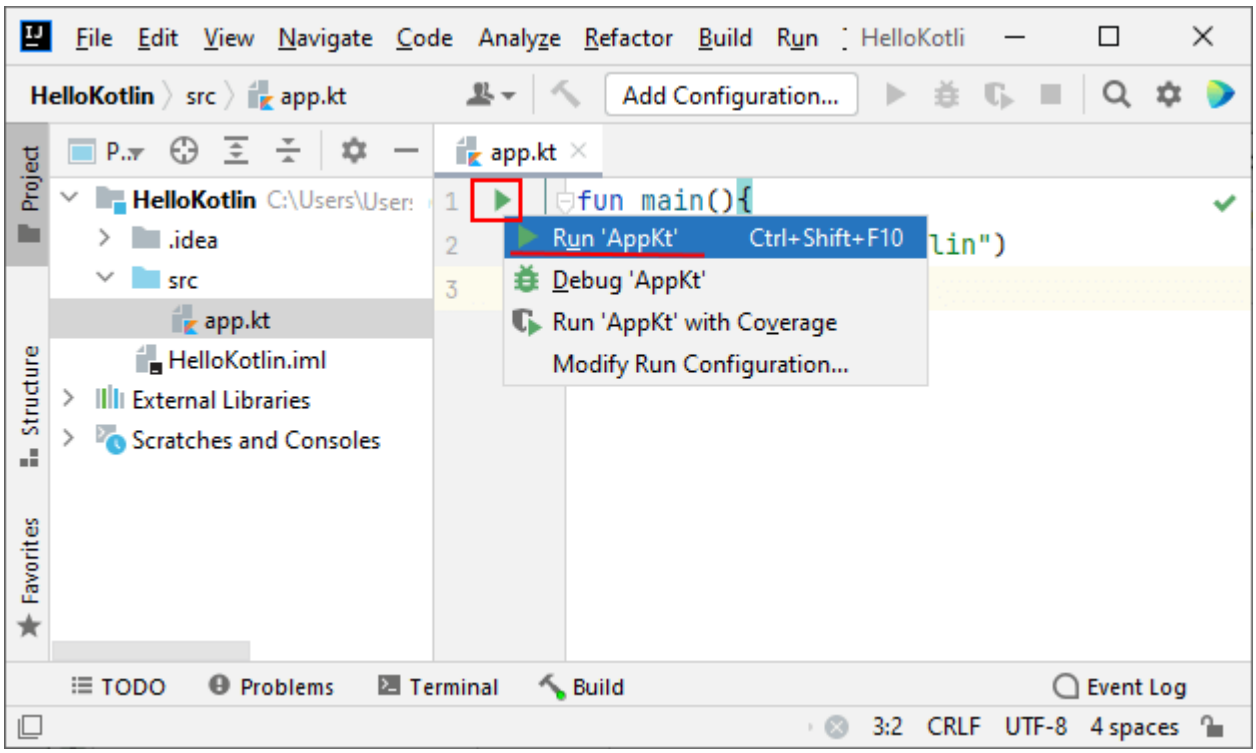

Рисунок 1.19. Вікно запуску програми га виконання

Після цього буде виконано побудову проєкту, а скомпільована програма буде запущена у консолі IntelliJ IDEA, як показано на рисунку 1.20.

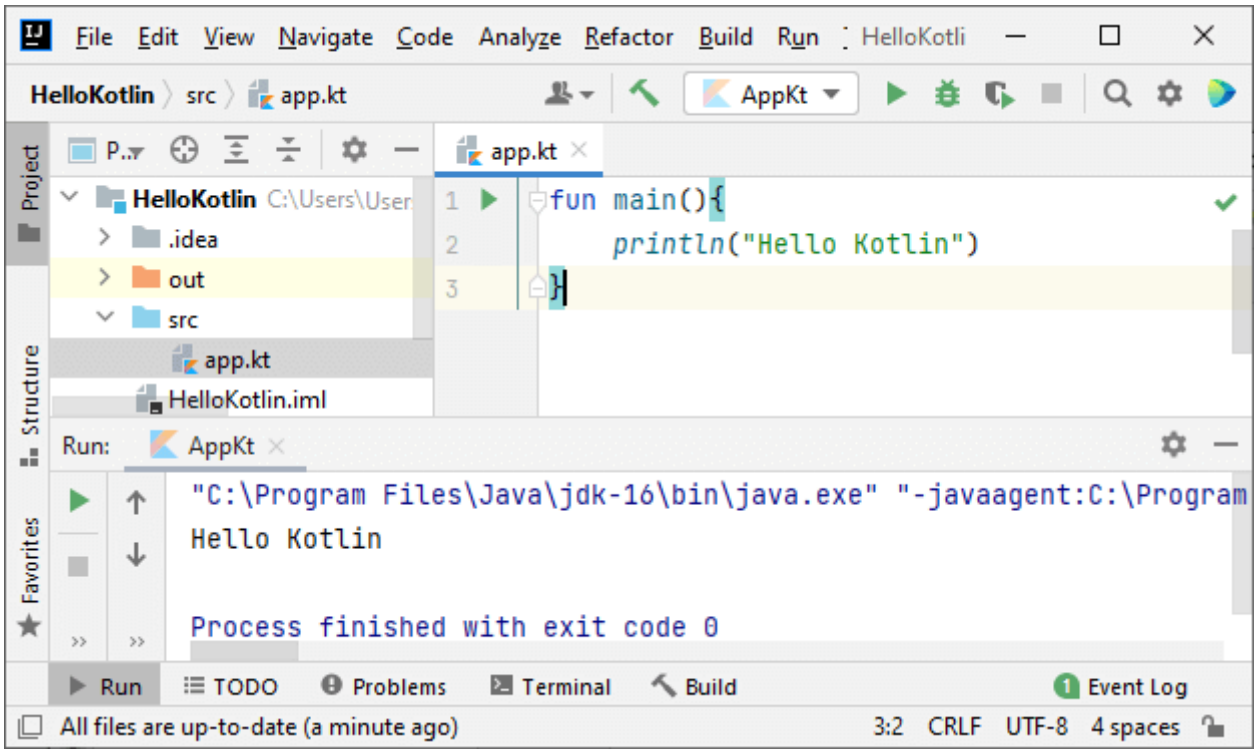

Рисунок 1.20. Результат роботи програми

Резюме

У цьому розділі розглянуто основи мови програмування Kotlin, її особливості та сфери застосування. Kotlin – це сучасна, статично типізована мова, що використовується для розробки мобільних, веб- та десктопних застосунків, а також підтримує кросплатформність. Пояснюється, як створити першу програму мовою Kotlin у текстовому редакторі та скомпілювати її за допомогою командного рядка. Також описується процес встановлення IntelliJ IDEA, популярного середовища розробки для Kotlin, і покрокове створення проєкту в ньому. Наведено приклади коду, що демонструють основні принципи роботи з мовою. Загалом матеріал дає базове уявлення про встановлення, налаштування та запуск простих програм на Kotlin.

Контрольні запитання і завдання

1. Що таке Kotlin і хто його розробив?
2. Які основні платформи підтримує Kotlin?
3. Коли була випущена перша версія Kotlin?
4. Які мови програмування вплинули на Kotlin?
5. Чому Kotlin став офіційною мовою для розробки під Android?
6. Яке ключове слово використовується для оголошення функції в Kotlin?
7. Що є точкою входу в програму Kotlin?
8. Яку функцію використовують для виведення тексту на екран у Kotlin?
9. Яке розширення мають файли, написані мовою Kotlin?
10. Як запустити програму, написану на Kotlin, з командного рядка?
11. Яку утиліту використовують для компіляції програм на Kotlin?
12. Де можна завантажити компілятор Kotlin?
13. Для чого використовується параметр -include-runtime при компіляції?
14. Як запустити скомпільовану програму у форматі **.jar**?
15. Чому потрібно встановити JDK перед початком роботи з Kotlin?
16. Яке середовище розробки рекомендується для Kotlin?
17. Які існують версії IntelliJ IDEA, і яка з них безкоштовна?
18. Які кроки необхідно виконати для створення нового проєкту на Kotlin у IntelliJ IDEA?
19. Як додати новий файл з кодом у проєкт Kotlin в IntelliJ IDEA?
20. Як запустити програму Kotlin у середовищі IntelliJ IDEA?
21. Напишіть програму на Kotlin, яка виводить на екран ваші ім’я та прізвище.

22. Створіть програму, яка виводить два рядки тексту у двох окремих командах **println()**.
23. Напишіть програму, яка виводить число 2025 без лапок, використовуючи **println()**.
24. Збережіть програму у файлі з розширенням **.kt**, скомпілюйте її через командний рядок та запустіть.
25. Встановіть IntelliJ IDEA, створіть у ньому проєкт Kotlin та додайте до нього файл з кодом програми.
26. Модифікуйте програму, щоб вона виводила текст у такому форматі (з новими рядками):
    Привіт!
    Це моя перша програма на Kotlin!
27. Напишіть програму, яка виводить на екран результат обчислення 25 + 17.
28. Запустіть свою програму через IntelliJ IDEA та переконайтеся, що вона працює правильно.
29. Створіть програму, яка виводить на екран три числа, кожне в окремому рядку.
30. Змініть програму так, щоб вона виводила не тільки текст, а й математичний вираз, наприклад:
    10 + 5 = 15

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Що є точкою входу в програму на Kotlin?

a) start()
b) main()
c) begin()
d) run()

Правильна відповідь: b).

Яка команда компілює програму Kotlin в командному рядку?

a) kotlinc app.kt
b) kotlinc app.kt -include-runtime -d app.jar
c) javac app.kt
d) compile app.kt

Правильна відповідь: b).

Яке середовище розробки найкраще підходить для роботи з Kotlin?

a) Eclipse

b) NetBeans

c) IntelliJ IDEA

d) Visual Studio

Правильна відповідь: c).

Що станеться, якщо забути написати main() у програмі Kotlin?

a) Код просто не буде виконаний

b) Програма згенерує помилку

c) Код все одно запуститься

d) Програма покаже "null"

Правильна відповідь: b).

Чому Kotlin став популярним для Android-розробки?

a) Його створила компанія Google

b) Він більш безпечний і сучасний, ніж Java

c) Він працює тільки на Android

d) Він вимагає більше коду, ніж Java

Правильна відповідь: b).

Закриті запитання з кількома правильними відповідями

Які з наведених мов вплинули на Kotlin?

a) Java

b) Python

c) Swift

d) Ruby

Правильна відповідь: a), c).

Завдання на відповідність

Співвіднесіть поняття з їхніми описами:

a) Kotlin → 1. Сучасна мова програмування від JetBrains

b) IntelliJ IDEA → 2. Середовище розробки для Kotlin

c) println() → 3. Функція для виведення тексту в консоль

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

Розставте етапи компіляції та запуску програми в правильному порядку:

a) Написання коду у файлі .kt → 1. Етап 1

b) Виконання команди kotlinc app.kt -include-runtime -d app.jar → 2. Етап 2

c) Запуск програми командою java -jar app.jar → 3. Етап 3

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

Вставити пропущені слова

Що потрібно додати у код, щоб він вивів "Hello, Kotlin"?

```
fun main() {
    _____________ ("Hello, Kotlin")
}
```

Правильна відповідь: println

Завдання з кодом

Що виведе наступний код?

```
fun main() {
    println(10 + 5)
}
```

a) 105

b) 10 + 5

c) 15

d) Помилка компіляції

Правильна відповідь: c).

2. ОСНОВИ МОВИ KOTLIN

2.1. Структура програми

2.1.1. Функція main

Точкою входу в програму мовою Kotlin є функція **main**. Саме з цієї функції починається виконання програми Kotlin, тому ця функція повинна бути в будь-якій програмі мовою Kotlin.

Так, у попередньому розділі було визначено функцію **main** наступним чином:

```
1    fun main(){
2        println("Hello Kotlin")
3    }
```

Визначення функції **main()**, як і інших функцій у Kotlin, починається з ключового слова **fun**. Насправді воно показує, що далі йде визначення деякої функції. Після **fun** вказується ім'я функції. У наведеному прикладі це **main**.

Після імені функції у дужках іде список параметрів функції. В наведеному прикладі функція **main** не приймає жодних параметрів, тому після імені функції йдуть порожні дужки.

Усі команди, які виконує функція, розташовуються у фігурних дужках. У прикладі, наведеному вище, єдине, що робить функція **main** це виведення на консоль повідомлення за допомогою іншої вбудованої функції **println()**.

Варто зазначити, що до версії 1.3 у мові Kotlin функція **main** повинна була приймати параметри в обов’язковому порядку, наприклад:

```
1    fun main(args: Array<String>) {
2        println("Hello Kotlin")
3    }
```

Параметр **args: Array<String>** є масивом рядків, через який у програму можна передати різні дані.

Починаючи з версії 1.3, використовувати таке визначення функції з параметрами необов'язково. Хоча його використання і не заборонено.

2.1.2. Інструкції та блоки коду

Основним будівельним блоком програми мовою Kotlin є **інструкції** (**statement**). Кожна інструкція виконує певну дію, наприклад, виклики функцій, оголошення змінних та присвоєння їм значень. Наприклад:

```
1   println("Hello Kotlin!");
```

Цей рядок викликає вбудовану функцію **println()**, яка виводить на консоль, деяке повідомлення. В наведеному прикладі це рядок "**Hello Kotlin!**".

Варто зазначити, що на відміну від інших схожих мов програмування, наприклад, Java, у Kotlin не обов'язково ставити після інструкції крапку з комою. Кожна інструкція просто розташовується з нового рядка:

```
1   fun main(){
2       println("Kotlin on Sunshine.com")
3       println("Hello Kotlin")
4       println("Kotlin is a fun")
5   }
```

Тим не менш, якщо інструкції розташовуються у одному рядку, то щоб їх відокремити одна від одної, треба ставити після інструкції крапку з комою:

```
1   fun main(){
2       println("Kotlin  on  Sunshine.com");println("Hello
    Kotlin");println("Kotlin is a fun")
3   }
```

2.1.3. Коментарі

Код програми може містити коментарі. Коментарі дозволяють зрозуміти зміст програми, що виконують ті чи інші її частини. При компіляції коментарі ігноруються і не впливають на роботу програми та її розмір.

У Kotlin є два типи коментарів: однорядковий та багаторядковий. Однорядковий коментар розміщується у одному рядку після подвійного слеша //. А багаторядковий коментар розташовується між символами **/* текст коментаря */**. Він може розташовуватись у декількох рядках. Наприклад:

```
1   /*
2       Багаторядковий коментар
3       Функція main -
4       точка входу у програму
5   */
6   fun main(){          // початок функції main
7
8       println("Hello Kotlin") // друк рядка на консолі
9   }                    // кінець функції main
```

2.2. Змінні

Для зберігання даних у програмі Kotlin, як і в інших мовах програмування, застосовуються змінні. **Змінна** є іменованою ділянкою пам'яті, яка зберігає певне значення.

Кожна змінна характеризується певним ім'ям, типом даних та значенням. Ім'я змінної є довільним ідентифікатором, який може містити алфавітно-цифрові символи або символ підкреслення і повинен починатися або з алфавітного символу, або зі знаку підкреслення. Для визначення змінної можна використовувати ключове слово **val**, або ключове слово **var**.

Формальне визначення змінної наступне:

```
val|var ім’я_змінної: тип змінної
```

Спочатку йде слово **val** або **var**, потім ім'я змінної і через двокрапку тип змінної.

Наприклад, визначимо змінну **age**:

```
1   val age: Int
```

Тобто в цьому прикладі оголошено змінну **age**, яка має тип **Int**. Тип **Int** говорить про те, що змінна буде містити цілі числа.

Після визначення змінної їй можна присвоювати значення:

```
1   fun main() {
2       val age: Int
3       age = 23
4       println(age)
5   }
```

Для присвоєння значення змінній використовується знак =. Після цього зі змінною можна виконувати різноманітні операції. Наприклад, у наведеному прикладі, за допомогою функції **println()** значення змінної друкується у консолі. При запуску цієї програми на консолі буде надруковано число 23.

Присвоєння значення змінній має здійснюватися лише після її оголошення. І також можна відразу надати змінній початкове значення при її оголошенні. Такий прийом називається ініціалізацією:

```
1   fun main() {
2       val age: Int = 23
3       println(age)
4   }
```

Однак обов'язково необхідно надавати змінній деяке значення до її використання:

```
1   fun main() {
2
3       val age: Int
4       println(age) // Помилка, змінна не ініціалізована
5   }
```

2.2.1. Змінювані та незмінювані змінні

Вище було сказано, що змінні можуть оголошуватись як за допомогою слова **val**, так і за допомогою слова **var**. З'ясуємо у чому різниця між цими двома способами?

За допомогою ключового слова **val** визначається змінна, що не може змінюватись (**immutable variable**). Тобто можна надати значення такій змінній лише один раз, але змінити його після першого присвоєння вже не буде можливим. Наприклад, у наступному випадку згенерується помилка:

```
1   fun main() {
2       val age: Int
3       age = 23 // тут все добре - перше присвоєння
4       age = 56 // тут помилка - перевизначити значення
    змінної не можна
5       println(age)
6   }
```

А у змінної, яка визначена за допомогою ключового слова **var,** можна багаторазово змінювати значення (**mutable variable**):

```
1   fun main() {
2       var age: Int
3       age = 23
4       println(age)
5       age = 56
6       println(age)
7   }
```

Тому, якщо не планується змінювати значення змінної у програмі, то краще визначати її з ключовим словом **val**.

2.3. Типи даних

У Kotlin всі компоненти програми, зокрема змінні, є об'єктами, які мають певний тип даних. Тип даних визначає, який розмір пам'яті може займати об'єкт цього типу та які операції з ним можна виконувати. У Kotlin є кілька базових типів даних: числа, символи, рядки, логічний тип та масиви.

2.3.1. Типи цілих чисел

Мова Kotlin підтримує наступні типи цілих чисел:

- **Byte**: зберігає ціле число від -128 до 127 і займає 1 байт;
- **Short**: зберігає ціле число від -32768 до 32767 і займає 2 байти;
- **Int**: зберігає ціле число від -2 147 483 648 (-2^{31}) до 2 147 483 647 ($2^{31} - 1$) і займає 4 байти;

- **Long**: зберігає ціле число від –9 223 372 036 854 775 808 (-2^{63}) до 9 223 372 036 854 775 807 ($2^{63}-1$) і займає 8 байт.

В останній версії Kotlin також додано підтримку цілих типів без знаку:

- **UByte**: зберігає ціле число від 0 до 255 і займає 1 байт;
- **UShort**: зберігає ціле число від 0 до 65535 і займає 2 байти;
- **UInt**: зберігає ціле число від 0 до $2^{32}-1$ і займає 4 байти;
- **ULong**: зберігає ціле число від 0 до $2^{64}-1$ і займає 8 байт.

Об'єкти цілих типів зберігають цілі числа, наприклад:

```
1   fun main(){
2
3       val a: Byte = -10
4       val b: Short = 45
5       val c: Int = -250
6       val d: Long = 30000
7       println(a) // -10
8       println(b) // 45
9       println(c) // -250
10      println(d) // 30000
11  }
```

Для присвоєння значень об'єктам, які є беззнаковими цілими типами даних, після числа вказується суфікс **U**, наприклад:

```
1   fun main(){
2
3       val a: UByte = 10U
4       val b: UShort = 45U
5       val c: UInt = 250U
6       val d: ULong = 30000U
7       println(a) // 10
8       println(b) // 45
9       println(c) // 250
10      println(d) // 30000
11  }
```

Орім чисел у десятковій системі можна визначати числа у двійковій та шістнадцятковій системах.

Шістнадцятковий запис числа починається з **0x**, потім іде набір символів від **0** до **F**, які позначають число, наприклад:

```
1   val address: Int = 0x0A1 // 161
2   println(address) // 161
```

Двійковому запису числа передують символами **0b**, після яких іде послідовність з нулів та одиниць, наприклад:

```
1   val a: Int = 0b0101 // 5
2   val b: Int = 0b1011 // 11
```

```
3   println(a) // 5
4   println(b) // 11
```

2.3.2. Числа з рухомою крапкою

Крім цілих типів у Kotlin є два типи для чисел з рухомою крапкою, які дозволяють зберігати дробові числа:

- **Float**: зберігає число з рухомою крапкою від $-3.4 \cdot 10^{38}$ до $3.4 \cdot 10^{38}$ і займає 4 байти;
- **Double**: зберігає число з рухомою крапкою від $\pm 5.0 \cdot 10^{-324}$ до $\pm 1.7 \cdot 10^{308}$ та займає 8 байти.

Як роздільник цілої та дробової частини застосовується символ крапки, наприклад:

```
1   val height: Double = 1.78
2   val pi: Float = 3.14F
3   println(height) // 1.78
4   println(pi) // 3.14
```

Щоб присвоїти число об'єкту типу **Float** після числа, вказується суфікс **f** або **F**.

Також тип **Double** підтримує експонентний запис, наприклад:

```
1   val d: Double = 23e3
2   println(d) // 23 000
3
4   val g: Double = 23e-3
5   println(g) // 0.023
```

2.3.3. Логічний тип Boolean

Тип **Boolean** може зберігати одне із двох значень: **true** (істина) або **false** (хиба), наприклад:

```
1   val a: Boolean = true
2   val b: Boolean = false
```

2.3.4. Символьні дані

Символьні дані представлені типом **Char**. Він позначає окремий символ, який розташовується у одинарних лапках, наприклад:.

```
1   val a: Char = 'A'
2   val b: Char = 'B'
3   val c: Char = 'T'
```

Також тип **Char** може позначати спеціальні послідовності, які інтерпретуються особливим чином, наприклад:

- **\t** : табуляція;

- **\n** : новий рядок;
- **\r** : повернення каретки;
- \' : одинарні лапки;
- \" : подвійні лапки;
- \\ : зворотний слеш.

2.3.5. Рядки

Рядки в мові Kotlin представлені типом **String**. Рядок є послідовністю символів, яка розташовується у подвійних лапках, або потрійних подвійних лапках, наприклад:

```
1   fun main() {
2
3       val name: String = "Petro"
4
5       println(name)
6   }
```

Рядок може містити спеціальні символи або ескейп-послідовності. Наприклад, якщо потрібно вставити в текст перехід на новий рядок, можна використати ескейп-послідовність **\n**, наприклад:

```
1   val text: String = "Мова Kotlin - це сучасна, потужна,
    універсальна, лаконічна, мультиплатформна мова.
    \n За кілька років майже всі у світі андроїд розробники
    (і не тільки) перейшли на неї. У багатьох інших сферах
    розробки мова Kotlin також активно завойовує
    популярність. Це насамперед мультиплатформна розробка
    мобільних застосунків з можливістю писати код одночасно
    під iOS та Android, а також back-end розробка"
```

Для більшої зручності, при створенні багаторядкового тексту можна використовувати потрійні подвійні лапки, наприклад:

```
1   fun main() {
2
3       val text: String = """
4         Мова Kotlin - це сучасна, потужна, універсальна,
    лаконічна, мультиплатформна мова.
5         За кілька років майже всі у світі андроїд
    розробники (і не тільки) перейшли на неї.
6         У багатьох інших сферах розробки мова Kotlin
    також активно завойовує популярність. Це насамперед
    мультиплатформна розробка мобільних застосунків з
    можливістю писати код одночасно під iOS та Android, а
    також back-end розробка
```

```
7                               """
8       println(text)
9   }
```

2.3.5.1. Шаблони рядків

Шаблони рядків (**string templates**) є зручним засобом вставки в рядок різних значень, зокрема, значень змінних. Так, за допомогою знаку долара **$** можна вводити в рядок значення різних змінних, наприклад:

```
1   fun main() {
2
3       val firstName = "Taras"
4       val lastName = "Shevchenko"
5       val welcome = "Привіт, $firstName $lastName"
6       println(welcome) // Hello, Taras Shevchenko
7   }
```

У цьому прикладі замість **$firstName** і **$lastName** вставлятимуться значення відповідних змінних. При цьому змінні необов'язково мають бути рядкового типу, наприклад:

```
1   val name = "Taras"
2   val age = 22
3   val userInfo = "Ваше ім'я: $name Ваш вік: $age"
```

2.3.6. Автоматичне визначення типу

Мова Kotlin дозволяє автоматично визначати тип змінної на основі даних, якими змінну ініціалізовано. Тому при ініціалізації змінної тип можна не вказувати, наприклад:

```
1   val age = 5
```

У цьому прикладі компілятор побачить, що змінній присвоюється значення типу **Int**, тому змінна **age** матиме тип **Int**.

Відповідно якщо присвоїти змінній рядок, то така змінна матиме тип **String**, наприклад.

```
1   val name = "Taras"
```

Будь-які цілі числа сприймаються як дані типу **Int**.

Якщо необхідно явно вказати, що число має значення типу **Long**, то слід використовувати суфікс **L**, наприклад:

```
1   val sum = 45L
```

Якщо треба вказати, що об'єкт є беззнаковим типом, то застосовується суфікс **u** або **U**, наприклад:

```
1   val sum = 45U
```

Аналогічно, всі числа з рухомою крапкою, які містять точку як роздільник цілої та дробової частини, розглядаються як числа типу **Double**, наприклад:

```
1   val height = 1.78
```

Якщо необхідно вказати, що дані будуть мати тип **Float**, то необхідно використовувати суфікс **f** або **F**, наприклад:

```
1  val height = 1.78F
```

Проте, не можна спочатку оголосити змінну без зазначення типу, а потім десь у програмі надати їй якесь значення, наприклад:

```
1   val age // Помилка, змінну не ініціалізовано
2   age = 5
```

2.3.7. Статична типізація та тип Any

Оголошення типу даних обмежує набір значень, які можна присвоїти змінній. Наприклад, не можна присвоїти змінній типу **Double** рядок **String**:

```
1   val height: Double = "1.78"
```

І після того, як тип змінної встановлено, його не можна буде змінити, наприклад:

```
1   fun main() {
2
3       var height: String = "1.78"
4       height = 1.81 // !Помилка - змінна height зберігає
    лише рядки
5       println(height)
6   }
```

Однак у мові Kotlin також є тип **Any**, який дозволяє присвоїти змінній цього типу будь-яке значення, наприклад:

```
1   fun main() {
2
3       var name: Any = "Taras"
4       println(name) // Taras
5       name = 6758
6       println(name) // 6758
7   }
```

2.4. Введення та виведення на консоль

2.4.1. Виведення на консоль

Для виведення інформації на консоль у мові Kotlin є дві вбудовані функції, а саме:

```
1   print()
2   println()
```

Обидві ці функції приймають певний об'єкт, який необхідно вивести на консоль, зазвичай це рядок. Відмінність між ними полягає в тому, що функція **println()** при виведенні на консоль виконує перехід на новий рядок, наприклад:

```
1   fun main() {
2
3       print( "Привіт" )
4       print( "Kotlin" )
5       print( "on Sunshine.com" )
6       println()
7       println( "Kotlin це весело" )
8   }
```

Причому функція **println()** необов'язково має набувати певного значення. Так, у прикладі вище, застосовується порожній виклик функції, який просто переводить консоль на новий рядок:

```
1   println()
```

Результат роботи програми наведено на рисунку 2.1.

```
Привіт Kotlin на Sunshine.com
Kotlin це весело
```

Рисунок 2.1. Результат роботи програми

2.4.2. Введення з консолі

Для введення даних з консолі використовується вбудована функція **readLine()**. Вона повертає введений рядок. Варто зазначити, що результат цієї функції завжди є об'єктом типу **String**. Відповідно введений рядок можна передати до змінної типу **String**, наприклад:

```
1   fun main() {
2
3       print( "Введіть ім'я: " )
4       val name = readLine()
5
6       println( "Ваше ім'я: $name" )
7   }
```

В цьому прикладі спочатку виводиться запрошення на введення даних. Далі, введене значення передається змінній **name**. Результат роботи програми наведено на рисунку 2.2.

```
Введіть ім'я: Тарас
Ваше ім'я: Тарас
```

Рисунок 2.2. Результат роботи програми

Подібним чином можна вводити різні дані, наприклад:

```
1   fun main() {
2
```

```
3       print( "Введіть ім'я: " )
4       val name = readLine()
5       print( "Введіть email: " )
6       val email = readLine()
7       print( "Введіть адресу: " )
8       val address = readLine()
9
10      println( "Ваше ім'я: $name" )
11      println("Ваш email: $email")
12      println("Ваша адреса: $address")
13  }
```

Приклад роботи програми наведено на рисунку 2.3.

```
Введіть ім'я: Тарас
Введіть email: taras@au.mf
Введіть адресу: вул. Будівельників, буд.333, кв. 777
Ваше ім'я: Тарас
Ваш email: taras@au.mf
Ваша адреса: вул. Будівельників, буд.333, кв. 777
```

Рисунок 2.3. Результат роботи програми

2.5. Операції з числами

2.5.1. Арифметичні операції

Мова Kotlin підтримує базові арифметичні операції, а саме:

Оператор додавання (+), який повертає суму двох чисел, наприклад:

```
1   val x = 15
2   val y = 16
3   val z = x + y
4   println(z) // z = 31
```

Оператор віднімання (-), який повертає різницю двох чисел, наприклад:

```
1   val x = 15
2   val y = 16
3   val z = x - y // z = -1
```

Оператор множення (*), який повертає добуток двох чисел, наприклад:

```
1   val x = 4
2   val y = 5
3   val z = x * y // z = 20
```

Оператор ділення (/), який повертає результат ділення двох чисел, наприклад:

```
1   val x = 80
```

```
2   val y = 20
3   val z = x/y // z = 4
```

При цьому якщо в операції ділення обидва операнди представляють цілі числа, то результатом теж буде ціле число, а якщо в процесі ділення утворилася дробова частина, то вона відкидається, наприклад:

```
1   fun main() {
2
3       val x = 11
4       val y = 5
5       val z = x / y // z = 2
6       println(z) // 2
7   }
```

Так у цьому прикладі, якщо згідно зі стандартною математикою поділити **11** на **5**, то отримаємо **2.2**. Але оскільки обидва операнда представляють цілий тип, а саме тип **Int**, то дробова частина - **0.2** відкидається, тому результатом буде число **2**, а змінна **z** представлятиме тип **Int**.

Щоб результатом було дробове число, один з операндів повинен представляти число з рухомою крапкою, наприклад:

```
1   fun main() {
2
3       val x = 11
4       val y = 5.0
5       val z = x/y // z =2.2
6       println(z) // 2.2
7   }
```

У цьому прикладі змінна **y** представляє тип **Double**, тому результатом ділення буде число **2.2**, і змінна **z** також представлятиме тип **Double.**

Оператор (**%**), який повертає залишок від цілочислового ділення двох чисел, наприклад:

```
1   val x = 65
2   val y = 10
3   val z = x % y // z = 5
```

Оператор інкремента (++), який виконує збільшення на одиницю значення і буває двох видів.

Префіксний інкремент, який повертає збільшене значення, наприклад:

```
1   var x = 6
2   val y = ++x
3   println(x) // x = 7
4   println(y) // y = 7
```

Постфіксний інкремент повертає значення, яке було до збільшення на одиницю, наприклад:

```
1   var x = 6
```

```
2   val y = x++
3   println(x) // x = 7
4   println(y) // y = 6
```

Оператор декремента (--), який виконує зменшення значення на одиницю і буває двох видів.

Префіксний декремент, який повертає зменшене значення, наприклад:

```
1   var x = 6
2   val y = --x
3   println(x) // x = 5
4   println(y) // y = 5
```

Постфіксний декремент повертає значення, яке було до зменшення на одиницю, наприклад:

```
1   var x = 6
2   val y = x--
3   println(x) // x = 5
4   println(y) // y = 6
```

Також є низка операторів присвоєння, які поєднують арифметичні операції та операції присвоєння, а саме:

- **+=** Присвоєння після додавання. Присвоює лівому операнду суму лівого та правого операндів: **A += B**, що еквівалентно **A = A + B**;
- **-=** Присвоєння після віднімання. Присвоює лівому операнду різницю лівого і правого операндів: **A -= B**, що еквівалентно **A = A – B**;
- ***=** Присвоєння після множення. Присвоює лівому операнду добуток лівого та правого операндів: **A *= B**, що еквівалентно **A = A * B**;
- **/=** Присвоєння після ділення. Присвоює лівому операнду результат ділення лівого та правого операндів: **A /= B**, що еквівалентно **A = A / B**;
- **%=** Присвоєння після ділення за модулем. Присвоює лівому операнду залишок від цілочислового ділення лівого операнда на правий: **A %= B**, що еквівалентно **A = A % B**

2.5.2. Порозрядні операції

Мова Kotlin має низку операторів для роботи з двійковими розрядами числа. Важливо розуміти, як виглядає двійкове подання тих чи інших чисел. Зокрема, число **4** у двійковому вигляді матиме вигляд – **100**, а число **15** – **1111**.

У мові Kotlin передбачені наступні порозрядні оператори. При цьому варто зазначити, що вони застосовуються тільки до даних типів **Int** та **Long**.

Оператор зсуву бітів числа зі знаком ліворуч (**shl**), наприклад:

```
1   val z = 3 shl 2 // z = 11 << 2 = 1100
```

```
2   println(z) // z = 12
3   val d = 0b11 shl 2
4   println(d) // d = 12
```

У цьому прикладі число зсувається на два розряди ліворуч, тому праворуч число у двійковому поданні доповнюється двома нулями. В двійковому поданні число **3** позначається як **11**. Зсув на два розряди ліворуч і доповнення праворуч двома нулями формує кінцевий результат у вигляді **1100**, тобто в десятковій системі це буде число **12**.

Оператор зсуву бітів числа зі знаком праворуч (**shr**), наприклад:

```
1   val z = 12 shr 2 // z = 1100 >> 2 = 11
2   println(z) // z = 3
3   val d = 0b1100 shr 2
4   println(d) // d = 3
```

У цьому прикладі число **12** зсувається на два розряди праворуч, тобто фактично два розряди праворуч відкидаються і отримується число **11**, тобто **3** в десятковій системі.

Оператор зсуву бітів беззнакового числа праворуч (**ushr**), наприклад:

```
1   val z = 12 ushr 2 // z = 1100 >> 2 = 11
2   println(z) // z = 3
```

Оператор логічного множення **І** або кон'юнкція (**and**). Ця операція порівнює відповідні розряди двох чисел і повертає одиницю, якщо ці розряди у обох числах дорівнюють **1**, інакше повертає **0**, наприклад:

```
1   val x = 5 // 101
2   val y = 6 // 110
3   val z = x і y // z = 101 & 110 = 100
4   println(z) // z = 4
5
6   val d = 0b101 і 0b110
7   println(d) // d = 4
```

Оператор логічного додавання **АБО** або диз'юнкція (**or**). Ця операція порівнює два відповідні розряди обох чисел і повертає **1**, якщо хоча б один розряд дорівнює **1**. Якщо обидва розряди дорівнюють **0**, то повертається **0**, наприклад.

```
1   val x = 5 // 101
2   val y = 6 // 110
3   val z = x або y // z = 101 | 110 = 111
4   println(z) // z = 7
5
6   val d = 0b101 або 0b110
7   println(d) // d = 7
```

Оператор **виключного АБО** (**xor**) порівнює два розряди і повертає **1**, якщо один із розрядів дорівнює **1**, а інший дорівнює **0**. Якщо обидва розряди рівні, то повертається **0**, наприклад:

```
1   val x = 5 // 101
2   val y = 6 // 110
3   val z = x xor y // z = 101^110 = 011
4   println(z) // z = 3
5
6   val d = 0b101 xor 0b110
7   println(d) // d = 3
```

Оператор логічного заперечення **НЕ** (**inv**) або інверсія інвертує біти числа, наприклад:

```
1   val b = 11 // 1011
2   val c = b.inv()
3   println(c) // -12
```

2.6. Умовні вирази

Умовні вирази позначають деяку умову, яка повертає значення типу **Boolean**: або **true** (якщо умова істинна), або **false** (якщо умова хибна).

2.6.1. Операції порівняння

Розглянемо детально оператори порівняння.

Оператор «більше ніж» (>) повертає **true**, якщо перший операнд більший за другий. Інакше повертає **false**, наприклад:

```
1   val a = 11
2   val b = 12
3   val c : Boolean = a > b
4   println(c) // false - a менше ніж b
5
6   val d = 35 > 12
7   println(d) // true - 35 більше ніж 12
```

Оператор «менше ніж» (<) повертає **true**, якщо перший операнд менший за другий. Інакше повертає **false**, наприклад:

```
1   val a = 11
2   val b = 12
3   val c = a < b // true
4
5   val d = 35 < 12 // false
```

Оператор «більше ніж або дорівнює» (>=) повертає **true**, якщо перший операнд більший або дорівнює другому, наприклад:

```
1   val a = 11
```

```
2   val b = 12
3   val c = a >= b // false
4   val d = 11 >= a // true
```

Оператор «менше ніж або дорівнює» (<=) повертає **true**, якщо перший операнд менший або дорівнює другому, наприклад:

```
1   val a = 11
2   val b = 12
3   val c = a <= b // true
4   val d = 11 <= a // false
```

Оператор «дорівнює» (==) повертає **true**, якщо обидва операнди дорівнюють один одному. Інакше повертає **false**, наприклад:

```
1   val a = 11
2   val b = 12
3   val c = a == b // false
4   val d = b == 12 // true
```

Оператор «НЕ дорівнює» (**!=**) повертає **true**, якщо обидва операнди не дорівнюють один одному, наприклад:

```
1   val a = 11
2   val b = 12
3   val c = a! = b // true
4   val d = b! = 12 // false
```

2.6.2. Логічні операції

Операндами у логічних операціях є два значення типу **Boolean**. Нерідко логічні операції поєднують кілька операцій порівняння. Розглянемо їх більш детально.

Оператор **and** повертає **true**, якщо обидва операнди дорівнюють **true**, наприклад:

```
1   val a = true
2   val b = false
3   val c = a and b // false
4   val d = (11> = 5) and (9 <10) // true
5   println(c)
6   println(d)
```

Оператор **or** повертає **true**, якщо хоча б один із операндів дорівнює **true**, наприклад:

```
1   val a = true
2   val b = false
3   val c = a or b // true
4   val d = (11 < 5) or (9 > 10) // false
```

Оператор **xor** повертає **true**, якщо один з операндів дорівнює **true**. Якщо ж операнди дорівнюють один одному, то повертається **false**, наприклад:

```
1   val a = true
2   val b = false
3   val c = a xor b // true
4   val d = a xor (90> 10) // false
```

Оператор **!** повертає **true**, якщо операнд дорівнює **false**. І навпаки, якщо операнд дорівнює **true**, то повертається **false**, наприклад:

```
1   val a = true
2   val b = !a // false
3   val c = !b // true
```

Як альтернатива оператору **!** можна використовувати метод **not()**:

```
1   val a = true
2   val b = a.not() // false
3   val c = b.not() // true
```

Оператор **in** повертає **true**, якщо операнд належить деякій послідовності, наприклад:

```
1   val a = 5
2   val b = a in 1..6 // true - число 5 належить до
    послідовності від 1 до 6
3
4   val c = 4
    val d = c in 11..15 // false - число 4 не належить до
5   послідовності від 11 до 15
```

В наведеному прикладі, вираз **1..6** створює послідовність чисел від **1** до **6**. І в цьому випадку оператор **in** перевіряє, чи належить значення змінної **a** до цієї послідовності. Оскільки значення змінної **a** належить до цієї послідовності, то повертається **true**.

Вираз **11..15** створює послідовність чисел від **11** до **15**. І оскільки значення змінної **c** не належить цій послідовності, тому повертається **false**.

Якщо, навпаки, потрібно повертати **true**, якщо число не належить до зазначеної послідовності, можна застосувати комбінацію операторів **!in**, наприклад:

```
1   val a = 8
2   val b = a !in 1..6 // true - число 8 не належить до
    послідовності від 1 до 6
```

2.7. Умовні конструкції

Умовні конструкції дозволяють направити виконання програми по одному із шляхів залежно від умови.

2.7.1. Конструкція if...else

Конструкція **if** приймає умову, і якщо ця умова є вірною, то виконується наступний блок інструкцій, наприклад:

```
1   val a = 10
2   if (a == 10) {
3
4       println( "a дорівнює 10" )
5   }
```

В наведеному прикладі в конструкції **if** перевіряється вірність виразу **a==10** якщо він вірний, то виконується наступний блок коду у фігурних дужках, і на консоль виводиться повідомлення **"a дорівнює 10"**. Якщо твердження помилкове, тоді блок коду не виконується.

Якщо необхідно задати альтернативний варіант, можна додати блок **else**, наприклад:

```
1   val a = 10
2   if (a == 10) {
3       println( "a дорівнює 10" )
4   }
5   else {
6       println( "a НЕ дорівнює 10" )
7   }
```

Таким чином, якщо умовний вираз після оператора **if** вірний, то виконується блок після **if**, якщо хибний, то виконується блок після **else**.

Якщо блок коду складається з одного виразу, то фігурні дужки можна опустити, наприклад:

```
1   val a = 10
2   if (a == 10)
3       println( "a дорівнює 10" )
4   else
5       println( "a НЕ дорівнює 10" )
```

Якщо необхідно перевірити кілька альтернативних варіантів, можна додати вирази **else if**, наприклад:

```
1   val a = 10
2   if (a == 10) {
3       println( "a дорівнює 10" )
4   }
5   else if (a == 9) {
6       println( "a дорівнює 9" )
7   }
8   else if (a == 8) {
9       println( "a дорівнює 8" )
10  }
```

```
11 else {
12     println( "a має невідоме значення" )
13 }
```

2.7.1.1. Повернення значення з if

Варто зазначити, що конструкція **if** може повертати значення. Наприклад, знайдемо максимальне із двох чисел:

```
1   val a = 10
2   val b = 20
3   val c = if (a > b) a else b
4
5   println(c) // 20
```

Якщо при визначенні значення, що повертається, необхідно виконати ще які-небудь дії, то можна розташувати ці дії в середину блоку коду, наприклад:

```
1   val a = 10
2   val b = 20
3   val c = if (a > b) {
4       println( "a = $a" )
5       a
6   } else {
7       println( "b = $b" )
8       b
9   }
```

В наведеному прикладі, в кінці кожного блоку вказується значення, що повинно повертатися.

2.7.2. Конструкція when

Конструкція **when** перевіряє значення деякого об'єкта і залежно від його значення виконує той чи інший код. Конструкція **when** аналогічна конструкції **switch** в інших мовах. Формальне визначення наступне:

```
1   when (об'єкт) {
2
3       значення1 -> дії1
4       значення2 -> дії2
5
6       значення N -> дії N
7   }
```

Якщо значення об'єкта дорівнює одному із значень у блоці коду **when**, то виконуються відповідні дії, що записані після оператора -> після відповідного значення, наприклад:

```
1   fun main() {
2
```

```
3       val isEnabled = true
4       when (isEnabled){
5           false -> println( "isEnabled off" )
6           true -> println( "isEnabled on" )
7       }
8   }
```

В цьому прикладі змінна **isEnabled** передається як об’єкт до конструкції **when**. Далі її значення по черзі порівнюється зі значеннями **false** і **true**. В наведеному прикладі змінна **isEnabled** дорівнює **true**, тому буде виконуватись наступний код:

```
println( "isEnabled on" )
```

2.7.2.1. Вираз else

У прикладі вище змінна **isEnabled** мала тільки два можливі варіанти: **true** і **false**. Однак частіше трапляються випадки, коли значення в блоці **when** не покривають всі можливі значення об'єкта. Додатковий вираз **else** дозволяє задати дії, які виконуються, якщо об'єкт не відповідає жодному зі значень, наприклад:

```
val a = 30
when (a){
    10 -> println( "a = 10" )
    20 -> println( "a = 20" )
    else -> println( "невідоме значення" )
}
```

В наведеному прикладі, через те, що змінна **a** дорівнює **30,** то вона не відповідає жодному зі значень в блоці **when**. І відповідно виконуватимуться інструкції з виразу **else**.

Якщо необхідно, аби при збігу значень виконувалося кілька інструкцій, то для кожного значення можна визначити окремий блок коду, наприклад:

```
var a = 10
when (a){
    10 -> {
        println( "a = 10" )
        a *= 2
    }
    20 -> {
        println( "a = 20" )
        a *= 5
    }
    else -> { println( "невідоме значення" )}
}
println(a)
```

2.7.2.2. Порівняння з набором значень

У мові Kotlin передбачена можливість визначити одні й самі дії відразу для кількох значень. У цьому випадку значення перераховуються через кому, наприклад:

```
1   val a = 10
2   when (a){
3       10, 20 -> println( "a = 10 або a = 20" )
4       else -> println( "невідоме значення" )
5   }
```

Також можна здійснювати порівняння з цілим діапазоном значень за допомогою оператора **in**, наприклад:

```
1   val a = 10
2   when (a){
3       in 10..19 -> println( "a в діапазоні від 10 до 19" )
4       in 20..29 -> println( "a в діапазоні від 20 до 29" )
5       !in 10..20 -> println( "a поза діапазоном від 10 до
    20" )
6       else -> println( "невідоме значення" )
7   }
```

Якщо оператор **in** дозволяє дізнатися, чи належить значення до певного діапазону, то зв'язка операторів **!in** дозволяє перевірити відсутність належності значення до певної послідовності.

2.7.2.3. When і значення, що динамічно обчислюються

Вираз у **when** також може здійснювати порівняння зі значеннями, що динамічно обчислюються, наприклад:

```
1   fun main() {
2
3       val a = 10
4       val b = 5
5       val c = 3
6       when (a){
7           b - c -> println( "a = b - c" )
8           b + 5 -> println(" a = b + 5" )
9           else -> println( "невідоме значення" )
10      }
11  }
```

Так, у наведеному прикладі значення змінної **a** порівнюється з результатом операцій **b – c** та **b + 5**.

Крім того, **when** також може приймати об'єкт, що динамічно обчислюється, наприклад:

```
1   fun main() {
```

```
2
3       val a = 10
4       val b = 20
5       when (a + b) {
6           10 -> println( "a + b = 10" )
7           20 -> println( "a + b = 20" )
8           30 -> println( "a + b = 30" )
9           else -> println( "невідоме значення" )
10      }
11  }
```

Можна навіть визначати змінні, які будуть доступні всередині блоку **when**, наприклад:

```
1   fun main() {
2
3       val a = 10
4       val b = 26
5       when (val c = a + b) {
6           10 -> println( "a + b = 10" )
7           20 -> println( "a + b = 20" )
8           else -> println( "c = $c" )
9       }
10  }
```

2.7.2.4. When як альтернатива для if..else

Варто зазначити, що взагалі то необов'язково порівнювати значення будь-якого об'єкта. Конструкція **when** аналогічно до конструкції **if..else** просто може повіряти набір умов і якщо одна з умов повертає **true**, то здійснити виконання відповідного переліку дій, наприклад:

```
1   fun main() {
2
3       val a = 15
4       val b = 6
5       when {
6           (b > 10) -> println( "b більше 10" )
7           (a > 10) -> println( "a більше 10" )
8           else -> println( "і a, і b менше або дорівнюють
    10" )
9       }
10  }
```

2.7.2.5. Повернення значення

Як і **if** конструкція **when** може повертати значення. Значення, що повертається, розташовується після оператора ->, наприклад:

```
1   val sum = 1000
```

```
2
3   val rate = when (sum) {
4       in 100..999 -> 10
5       in 1000..9999 -> 15
6       else -> 20
7   }
8   println(rate) // 15
```

Таким чином, якщо значення змінної **sum** належить певному діапазону, то повертається значення, яке вказано після символу стрілки ->.

2.8. Цикли

Цикли є видом керуючих конструкцій, які дозволяють залежно від певних умов виконувати деяку дію багато разів.

2.8.1. Цикл For

Цикл **for** пробігає всіма елементами колекції. У цьому плані цикл **for** у мові Kotlin еквівалентний циклу **for-each** в низці інших мов програмування. Його формальна форма запису виглядає наступним чином:

```
1   for (змінна in послідовність) {
2       виконувані інструкції
3   }
```

Наприклад, надрукуємо у консолі всі квадрати чисел від **1** до **9**, використовуючи цикл **for**:

```
1   for (n in 1..9) {
2       print( "${n * n} \t" )
3   }
```

В наведеному прикладі перебирається послідовність чисел від **1** до **9**. При кожному проході циклу (ітерації циклу) з цієї послідовності обиратиметься елемент і поміщатиметься в змінну **n**. І через змінну **n** можна маніпулювати значення елемента. Результат роботи цієї програми наведено на рисунку 2.4.

```
1 4 9 16 25 36 49 64 81
```

Рисунок 2.4. Результат роботи програми

Цикли можуть бути вкладеними один до одного. Наприклад, роздрукуємо у консолі таблицю множення:

```
1   for (i in 1..9) {
2       for (j in 1..9) {
3           print( "${i * j} \t" )
4       }
5       println()
```

```
6   }
```

У результаті роботи програми на консоль буде виведена таблиця множення, як показано на рисунку 2.5.

```
1  2   3   4   5   6   7   8   9
2  4   6   8  10  12  14  16  18
3  6   9  12  15  18  21  24  27
4  8  12  16  20  24  28  32  36
5  10 15  20  25  30  35  40  45
6  12 18  24  30  36  42  48  54
7  14 21  28  35  42  49  56  63
8  16 24  32  40  48  56  64  72
9  18 27  36  45  54  63  72  81
```

Рисунок 2.5. Результат роботи програми

2.8.2. Цикл while

Цикл **while** повторює певні дії допоки вірна деяка умова, наприклад:

```
1   var i = 10
2   while (i > 0) {
3       println(i*i)
4       i--
5   }
```

В цьому прикладі, допоки змінна **i** більше за **0**, виконуватиметься цикл, у якому на консоль виводитиметься квадрат значення **i**.

Отже, спочатку перевіряється умова (**i > 0**) і якщо вона вірна (тобто повертає **true**), то виконується цикл. При цьому, цілком можлива ситуація, коли до початку виконання циклу умова не буде виконуватися. Наприклад, змінна **i** спочатку менше за **0**, тоді цикл взагалі не виконуватиметься жодного разу.

У мові Kotlin є ще одна форма циклу **while**, а саме: **do..while**, наприклад:

```
1   var i = -1
2   do {
3       println(i*i)
4       i--
5   }
6   while (i > 0)
```

В цьому прикладі спочатку виконується блок коду після ключового слова **do**, а вже потім перевіряється умова, котра зазначена після **while**. Якщо умова є вірною, то повторюється виконання блоку коду зазначеного після **do**. Тобто

незважаючи на те, що в наведеному прикладі змінна **i** менша за **0** і вона не відповідає умові, проте блок **do** виконається хоча б один раз.

2.8.3. Оператори continue та break

Іноді при використанні циклу виникає необхідність за певних умов не чекати на виконання всіх інструкцій у циклі, а перейти до нової ітерації. Для цього можна використовувати оператор **continue**, наприклад:

```
1   for (n in 1..8) {
2       if (n == 5) continue ;
3       println(n * n)
4   }
```

В цьому прикладі, в разі коли **n** дорівнюватиме **5**, спрацює оператор **continue**. І наступна інструкція, яка виводить на консоль квадрат числа, не виконуватиметься. Цикл перейде до обробки наступного елемента в масиві.

Трапляється, що за деяких умов необхідно вийти з циклу, тобто припинити його виконання. У цьому випадку застосовується оператор **break**, наприклад:

```
1   for (n in 1..5) {
2       if (n == 5) break ;
3       println(n * n)
4   }
```

Коли **n** стане дорівнювати **5**, то за допомогою оператора **break** буде виконано вихід із циклу. І цикл повністю завершиться.

2.9. Послідовності

Послідовністю є набір значень чи діапазон. Для створення послідовності застосовується оператор **..**, наприклад:

```
1   val range = 1..5 // послідовність [1, 2, 3, 4, 5 ]
```

Цей оператор приймає два значення – межі послідовності, і всі елементи між цими значеннями (включаючи їх самі) складають цю послідовність.

Послідовність не обов'язково має містити числові дані. Наприклад, це можуть бути рядки:

```
1   val range = "a" .. "d"
```

Оператор (**..**) дозволяє створити впорядковану послідовність по наростаючій, де кожен наступний елемент буде більшим за попередній. А за допомогою спеціальної функції **downTo** можна побудувати послідовність у зворотному порядку, наприклад:

```
1   val range1 = 1..5 // 1 2 3 4 5
2   val range2 = 5 downTo 1 // 5 4 3 2 1
```

Ще одна спеціальна функція **step** дозволяє встановити крок, на який будуть змінюватися наступні елементи, наприклад:

```
1   val range1 = 1..10 step 2 // 1 3 5 7 9
2   val range2 = 10 downTo 1 step 3 // 10 7 4 1
```

Функція **until** дозволяє не включати верхню межу в саму послідовність, наприклад:

```
1    val range1 = 1 until 9 // 1 2 3 4 5 6 7 8
2    val range2 = 1 until 9 step 2 // 1 3 5 7
```

За допомогою спеціальних операторів можна перевірити наявність або відсутність елементів у послідовності, наприклад:

in // повертає **true**, якщо об'єкт є у послідовності;

!in // повертає **true**, якщо об'єкт відсутній у послідовності

Приклад використання операторів **in** та **!in** наведено нижче.

```
1   fun main() {
2
3       val range = 1..5
4
5       var isInRange = 5 in range
6       println(isInRange) // true
7
8       isInRange = 86 in range
9       println(isInRange) // false
10
11      var isNotInRange = 6! in range
12      println(isNotInRange) // true
13
14      isNotInRange = 3! in range
15      println(isNotInRange) // false
16  }
```

2.9.1. Перебір елементів послідовності

За допомогою циклу **for** можна перебирати елементи послідовності, наприклад:

```
1   val range1 = 5 downTo 1
2   for (c in range1) print(c) // 54321
3   println()
4
5   val range2 = 'a'..'d'
6   for (c in range2) print(c) // abcd
7   println()
8
9   for (c in 1..9) print(c) // 123456789
```

```
10 println()
11
12 for (c in 1 until 9) print(c) // 12345678
13 println()
14
15 for (c in 1..9 step 2) print(c) // 13579
```

2.10. Масиви

Для зберігання набору значень у мові Kotlin, як і в інших мовах програмування, можна використовувати **масиви**. При цьому масив може зберігати дані тільки одного типу. У мові Kotlin масиви представлені типом **Array**.

При визначенні масиву після типу **Array** у кутових дужках необхідно вказати, об'єкти якого типу можуть зберігатись у масиві. Наприклад, визначимо масив цілих чисел:

```
1   val numbers: Array<Int>
```

За допомогою вбудованої функції **arrayOf()** можна передати набір значень, які складатимуть масив, наприклад:

```
1    val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
```

Тобто, в наведеному прикладі, створюється масив з п’ятьма числами від 1 до 5.

За допомогою індексів можна звертатися до певного елемента масиву. Індексація елементів у масивах починається з нуля, тобто перший елемент матиме індекс 0. Індекс вказується у квадратних дужках, наприклад:

```
1   val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
2   val n = numbers[1] // отримуємо другий елемент n=2
3   numbers[2] = 8 // записуємо у третій елемент число 8
4   println( "numbers[2] = ${numbers[2]}" ) // numbers[2] =
    8
```

Також можна ініціалізувати масив значеннями у наступний спосіб:

```
1   val numbers = Array(4, {1}) // [1, 1, 1, 1]
```

В цьому прикладі використовується конструктор класу **Array**. У цей конструктор передаються два параметри. Перший параметр вказує, скільки елементів буде у масиві. Отже у наведеному прикладі буде 4 елементи. Другим параметром є вираз, який генерує елементи масиву. Він розташовується у фігурних дужках. В наведеному прикладі у фігурних дужках стоїть число 1, тобто всі елементи масиву міститимуть число 1. Таким чином, масив складатиметься з чотирьох одиниць.

Але вираз, який створює елементи масиву, може бути складнішим, наприклад:

```
1   var i = 1;
2   val numbers = Array(3, { i++ * 2}) // [2, 4, 6]
```

В наведеному прикладі елементом масиву є результат множення змінної **i** на 2. При цьому, при кожному зверненні до змінної **i** значення її збільшується на одиницю.

2.10.1. Перебір елементів масивів

У мові Kotlin для перебору елементів масивів можна використовувати цикл **for**, наприклад:

```
1   fun main() {
2
3       val numbers = arrayOf(1, 2, 3, 4, 5)
4       for (number in numbers){
5           print( "$number \t" )
6       }
7   }
```

В цьому прикладі змінна **numbers** є масивом чисел. При переборі цього масиву в циклі кожен його елемент перебуває у змінній **number**, значення якої, наприклад, можна вивести на консоль. Результат роботи програми наведено на рисунку 2.6.

```
1 2 3 4 5
```

Рисунок 2.6. Результат роботи програми

Подібним чином можна перебирати елементи масивів й інших типів, наприклад:

```
1   fun main() {
2
3       val people =
    arrayOf( "Taras" , "Stepan" , "Volodymyr" )
4       for (person in people){
5           print( "$person \t" )
6       }
7   }
```

Результат роботи програми наведено на рисунку 2.7.

```
Taras Stepan Volodymyr
```

Рисунок 2.7. Результат роботи програми

Можна застосовувати інші типи циклів для перебору елементів масиву. Як приклад розглянемо можливість використання циклу **while**:

```
1   fun main() {
2
3       val people =
    arrayOf( "Taras" , "Stepan" , "Volodymyr" )
4
5       var i = 0
6       while ( i in people.indices) {
7           println(people[i])
8           i++;
9       }
10  }
```

В цьому прикладі визначено додаткову змінну **i**, яка позначає індекс елемента масиву. Також тут використано, притаманну масивам, спеціальну властивість **indices**, яка містить набір всіх індексів. А вираз **i in people.indices** повертає **true**, якщо значення змінної **i** входить до набору індексів масиву.

У самому циклі за індексом звертаємося до елемента масиву за допомогою конструкції **println (people [i])**, і потім переходимо до наступного індексу, збільшуючи лічильник **i ++**.

Те саме можна написати за допомогою циклу **for**, наприклад:

```
1   for (i in people.indices) {
2       println(people[i])
3   }
```

2.10.2. Перевірка наявності елемента у масиві

Як і у випадку з послідовностями, можна перевірити наявність або відсутність елементів масиві за допомогою операторів **in** та **!in**, наприклад:

```
1   val numbers: Array<Int> = arrayOf(1, 2, 3, 4, 5)
2
3   println(4 in numbers) // true
4   println(2 ! in numbers) // false
```

2.10.3. Масиви для базових типів

Для спрощення створення масивів у мові Kotlin визначено додаткові типи масивів: **BooleanArray**, **ByteArray**, **ShortArray**, **IntArray**, **LongArray**, **CharArray**, **FloatArray** та **DoubleArray.** Вони дозволяють створювати масиви для певних типів. Наприклад, тип масивів **IntArray** дозволяє визначити масив, що складається з об'єктів типу **Int**, а **DoubleArray** – масив, що складається з об'єктів типу **Double**:

```
1   val numbers: IntArray = intArrayOf(1, 2, 3, 4, 5)
2   val doubles: DoubleArray = doubleArrayOf(2.4, 4.5, 1.2)
```

Для визначення даних для цих масивів можна застосовувати функції, які починаються на назву типу в нижньому регістрі, наприклад, **int** і потім йде **ArrayOf**.

Аналогічно для ініціалізації подібних масивів також можна застосовувати конструктор відповідного класу, наприклад:

```
1   val numbers = IntArray(3, {5})
2   val doubles = DoubleArray(3, {1.5})
```

2.10.4. Двовимірні масиви

Вище розглядалися одномірні масиви, які можна уявити як рядок значень. Але, крім того, у мові Kotlin можна використовувати багатовимірні масиви. Наприклад, візьмемо двовимірний масив, тобто такий масив, кожен елемент якого у свою чергу є масивом. Двовимірний масив ще можна уявити у вигляді таблиці, де кожен рядок – це окремий масив, а окремий елемент рядка – це елементи вкладеного масиву.

Визначення двомірних масивів менш інтуїтивно зрозуміле і може спричинити складнощі. Розглянемо наприклад, двомірний масив чисел:

```
1   val table: Array<Array<Int>> = Array(3, { Array(5, {0})
    })
```

В наведеному прикладі двомірний масив матиме три елементи - три рядки. Кожен рядок матиме по п'ять елементів, кожен із яких дорівнюватиме **0**.

Використовуючи індекси, можна звертатися до підмасивів у подібному масиві, у тому числі встановлювати їх значення, наприклад:

```
1    val table = Array(3, { Array(3, {0}) })
2    table[0] = arrayOf(1, 2, 3) // перший рядок таблиці
3    table[1] = arrayOf(4, 5, 6) // другий рядок таблиці
4    table[2] = arrayOf(7, 8, 9) // третій рядок таблиці
```

Для звернення до елементів підмасивів двовимірного масиву потрібні два індекси. За першим індексом здійснюється отримання рядка, а за другим індексом отримується стовпчик в межах цього рядка, наприклад:

```
1   val table = Array(3, { Array(3, {0}) })
2   table[0][1] = 6 // другий елемент першого рядка
3   val n = table[0][1] // n = 6
```

Використовуючи два цикли, можна перебирати елементи двовимірних масивів, наприклад:

```
1   fun main() {
2
3       val table: Array<Array<Int>> = Array(3, { Array(3,
    {0}) })
4       table[0] = arrayOf(1, 2, 3)
```

```
5       table[1] = arrayOf(4, 5, 6)
6       table[2] = arrayOf(7, 8, 9)
7       for (row in table) {
8
9           for (cell in row) {
10              print( "$cell \t" )
11          }
12          println()
13      }
14  }
```

За допомогою зовнішнього циклу **for(row in table)** пробігаємось по всіх елементах двовимірного масиву, тобто рядками таблиці. Кожен із елементів двовимірного масиву сам є масивом, тому можна пробігтися по цьому масиву і отримати з нього ті значення, які в ньому зберігаються. Результат роботи програми наведено на рисунку 2.8.

```
1    2    3
4    5    6
7    8    9
```

Рисунок 2.8. Результат роботи програми

Резюме

В наведеному розділі розглянуто основи мови програмування Kotlin. Він пояснює, що вхідною точкою програми є функція **main()**, а код складається з інструкцій і блоків. Описано, як працюють змінні (**val** – незмінні, **var** – змінні) та їхні типи даних (цілі числа, числа з рухомою крапкою, рядки, логічні значення тощо). Розглянуто операції з числами, включаючи арифметичні, порозрядні та логічні. Пояснено, як використовувати умовні конструкції (**if**, **when**) та цикли (**for**, **while**, **do..while**). Також подано інформацію про послідовності та масиви, їх створення, перебір елементів і перевірку наявності значень, без чого не можливо створення мобільних, десктопних та веб-застосунків мовою Kotlin.

Контрольні запитання і завдання

1. Яка функція є точкою входу в програму на Kotlin?
2. Чим відрізняється **fun main()** від **fun main(args: Array<String>)**?
3. Які типи коментарів підтримує мова Kotlin?
4. У чому різниця між **val** і **var** при оголошенні змінних?
5. Які існують типи цілих чисел у Kotlin?
6. Як присвоїти змінній типу **Float** значення?

7. Яке значення може приймати змінна типу **Boolean**?
8. Як визначити рядкову змінну в Kotlin?
9. Які основні арифметичні операції підтримує Kotlin?
10. Що робить оператор **%** при діленні чисел?
11. У чому різниця між **++x** і **x++?**
12. Як працюють оператори **shl** і **shr**?
13. Чим конструкція **when** відрізняється від **if...else**?
14. Як можна використовувати оператор **in** у **when**?
15. Що станеться, якщо в **when** не передбачено **else**, а жодна умова не виконується?
16. Як працює цикл **for** у Kotlin?
17. Чим відрізняється **while** від **do..while**?
18. Для чого використовуються оператори **break** та continue у циклах?
19. Як створити масив цілих чисел у Kotlin?
20. Як перевірити, чи міститься значення в масиві або послідовності?
21. Напишіть програму, яка виведе на екран «Привіт, Kotlin!».
22. Оголосіть змінну name (тип **String**) зі значенням «Андрій» і змінну **age** (тип **Int**) зі значенням 25. Виведіть на екран рядок «Мене звати Андрій, мені 25 років», використовуючи шаблони рядків.
23. Напишіть програму, яка запитує у користувача два числа, зчитує їх і виводить їхню суму, різницю, добуток і частку.
24. Напишіть програму, яка зчитує число від користувача та виводить, чи є воно додатним, від’ємним чи нулем.
25. Напишіть програму, яка приймає число і визначає, чи воно парне чи непарне.
26. Напишіть простий калькулятор, який приймає два числа і знак операції (+, -, *, /), а потім виводить результат, використовуючи when.
27. Використовуючи цикл **for**, виведіть таблицю множення для числа 5.
28. Напишіть програму, яка виводить всі парні числа від 1 до 20, використовуючи цикл **for** або **while**.
29. Створіть масив із 5 улюблених міст. Використовуючи **for**, виведіть кожне місто у консоль.
30. Напишіть програму, яка приймає три числа і визначає найбільше з них, використовуючи **if...else** або **when**.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Яка функція є точкою входу в програму на Kotlin?
a) start()
b) run()
c) main()
Правильна відповідь: c).

Закриті запитання з кількома правильними відповідями

Які змінні є незмінними в Kotlin?
a) val age = 25
b) var name = "Andrii"
c) val pi = 3.14
Правильна відповідь: a), c).

Завдання на відповідність

Співвіднесіть типи змінних із їх описами:
a) Int → 1. Цілі числа
b) String → 2. Текстові дані
c) Boolean → 3. Значення true або false
d) Double → 4. Числа з рухомою крапкою
Правильна відповідь:
a) → 1.
b) → 2.
c) → 3.
d) → 4.

Вставити пропущені слова

Оператор для перевірки, чи число належить певному діапазону:
`val result = 10 _____ 1..20`
Правильна відповідь: in

Завдання з кодом

Який код правильно визначає змінну, що містить рядок "Kotlin"?
a) val text = 'Kotlin'

b) val text = "Kotlin"

c) val text: Char = "Kotlin"

d) var text: Boolean = "Kotlin"

Правильна відповідь: b).

Що виконається при наступному коді?

```
val a = 10
if (a > 5) println("Більше 5") else println("Менше або дорівнює 5")
```

a) "Більше 5"

b) "Менше або дорівнює 5"

Правильна відповідь: a).

Скільки разів виконається цикл?

```
for (i in 1..5) {
    println(i)
}
```

a) 5

b) 4

c) 6

Правильна відповідь: 5

Відкриті запитання

Що виведе наступний код? І чому?

```
val x = 5
val y = 2
println(x / y)
```

Правильна відповідь: Буде виведено 2. Оскільки обидва числа цілі, дробова частина відкидається.

Яка помилка у цьому коді?

```
val name
name = "Kotlin"
```

Правильна відповідь: Змінну val необхідно одразу ініціалізувати.

Як отримати останній елемент масиву val arr = arrayOf(1, 2, 3, 4, 5)?

Правильна відповідь: arr.last() або arr[arr.size - 1]

3. ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ

3.1.Функції та їх параметри

Одним із будівельних блоків програми є функції. Функція визначає певну дію. У мові Kotlin функція оголошується за допомогою ключового слова **fun**, після якого вказується назва функції. Після назви в дужках вказується список параметрів. Якщо функція повертає яке-небуть значення, то після списку параметрів через кому можна вказати тип значення, що повертається. І далі у фігурних дужках розташовується тіло функції, наприклад.

```
fun ім'я_функції (параметри): тип_що_повертається {
    виконувані інструкції
}
```

При цьому, параметри є не обов'язковими.

Наприклад, визначимо та викличемо функцію, яка просто виводить деякий рядок на консоль:

```
fun main() {

    hello() // виклик функції hello
    hello() // виклик функції hello
    hello() // виклик функції hello
}
// Визначення функції hello
fun hello(){
    println( "Hello" )
}
```

Функції можна визначати у файлі поза іншими функціями чи класами, тобто окремо і самостійно, так наприклад, визначається функція **main**. Такі функції називають функціями верхнього рівня (**top-level functions**).

В прикладі, наведеному вище, крім головної функції **main**, також визначено функцію **hello**, яка не приймає жодних параметрів і нічого не повертає. Вона просто виводить рядок на консоль.

Функція **hello**, як і будь-яка інша конкретна функція, крім **main,** сама по собі не починає виконуватись. Щоб її запустити на виконання, її необхідно викликати. Для виклику функції вказується її ім'я, в наведеному прикладі **"hello"**, після якого ставлять порожні дужки.

Отже, якщо необхідно у різних частинах програми виконати одні й самі дії, можна ці дії винести у функцію, і потім викликати цю функцію за необхідністю.

3.1.1. Передача параметрів

Через параметри функція може отримувати деякі значення ззовні. Параметри вказуються після назви функції у дужках через кому у форматі **і'мя_параметра: тип_параметра**. Наприклад, визначимо функцію, яка обчислює факторіал числа:

```
1   fun main() {
2
3       showMessage( "Hello Kotlin" )
4       showMessage( "Привіт Kotlin" )
5       showMessage( "Salut Kotlin" )
6   }
7
8   fun showMessage(message: String){
9       println(message)
10  }
```

Функція **showMessage()** приймає один параметр типу **String**. Тому під час виклику функції у дужках необхідно передати значення для цього параметра, а саме **showMessage("Hello Kotlin")**. Причому значення параметру має бути типом **String**, тобто рядком. Значення, що передаються параметрам функції, ще називають аргументами. Результат роботи програми наведено на рисунку 3.1.

```
Hello Kotlin
Привіт Kotlin
Salut Kotlin
```

Рисунок 3.1. Результат роботи програми

Розглянемо інший приклад, а саме функцію, яка виводить інформацію про користувача на консоль:

```
1   fun main() {
2
3       displayUser( "Taras" , 23)
4       displayUser( "Olena" , 19)
5       displayUser( "Kateryna" , 25)
6   }
7   fun displayUser(name: String, age: Int) {
8       println( "Name: $name Age: $age" )
9   }
```

Функція **displayUser()** приймає два параметри: **name** та **age**. При виклику функції у дужках їй передаються значення цих параметрів. При цьому значення передаються параметрам за позицією і повинні відповідати параметрам за типом.

Оскільки спочатку записано параметр типу **String**, а потім параметр типу **Int**, то для виклику функції у дужках спочатку передається рядок, а потім число.

3.1.2. Аргументи за замовчуванням

У прикладі, наведеному вище, при виклику функцій **showMessage** та **displayUser** обов'язково потрібно надати для кожного параметра якесь певне значення, яке відповідає типу параметра. Не можна, наприклад, викликати функцію **displayUser**, не передавши їй аргументів для параметрів, це буде помилкою, наприклад:

```
1   displayUser() //Помилка
```

Проте можна визначити якісь параметри функції як необов'язкові та встановити для них значення за замовчуванням:

```
1   fun displayUser(name: String, age: Int = 18, position:
    String= "unemployed" ){
2       println( "Name:      $name Age:      $age Position:
    $position" )
3   }
4
5   fun main() {
6
7       displayUser( "Taras" , 23, "Manager" )
8       displayUser( "Olena" , 21)
9       displayUser( "Kateryna" )
10  }
```

У цьому прикладі функція **displayUser** має три параметри для передачі імені, віку та посади. Для першого параметра **name** значення за замовчуванням не встановлено, тому для нього, як і раніше, обов'язково передавати конкретне значення. Два наступні параметри **age** та **position** є необов'язковими, і для них встановлено значення за замовчуванням. Якщо для цих параметрів не передаються певні значення, то параметри використовують значення за замовчуванням. Тож для цих параметрів необов'язково передавати аргументи. Але якщо для якогось параметра визначено значення за замовчуванням, то для всіх наступних параметрів теж має бути встановлено значення за замовчуванням. Результат роботи програми наведено на рисунку 3.2.

```
Name: Taras Age: 23 Position: Manager
Name: Olena Age: 21 Position: unemployed
Name: Kateryna Age: 18 Position: unemployed
```

Рисунок 3.2. Результат роботи програми

3.1.3. Іменовані аргументи

За замовчуванням значення передаються параметрам за позицією. Тобто перше значення передається першому параметру, а друге значення передається другому параметру і так далі. Проте, використовуючи іменовані аргументи, можна перевизначити порядок передачі цих аргументів параметрам функцій, наприклад:

```
1   fun main() {
2
3       displayUser( "Taras" ,      position= "Manager" ,
    age=28)
4       displayUser(age=21, name= "Olena" )
5       displayUser( "Kateryna" ,       position= "Middle
    Developer" )
6   }
```

При виклику функції в дужках можна вказати назву параметра і за допомогою знаку рівності (=) передати йому потрібне значення.

При цьому, як видно з останнього прикладу, необов'язково всі аргументи передавати за іменем. Частина аргументів може передаватися параметрам за позицією. Але якщо якийсь аргумент передано за іменем, то і всі інші аргументи після нього також повинні передаватися за іменем відповідних параметрів.

Також, якщо до обов'язкового параметра функції вказані необов'язкові параметри, то для обов'язкового параметра значення передається за іменем, наприклад:

```
1   fun displayUser(age: Int = 18, name: String){
2       println( "Name: $name Age: $age" )
3   }
4   fun main() {
5
6       displayUser(name="Taras", age=28)
7       displayUser(name="Kateryna")
8   }
```

3.1.4. Зміна параметрів

За замовчуванням всі параметри функції рівносильні **val-змінним**, тому їх значення не можна змінити. Наприклад, у наступній функції під час компіляції згенерується помилка:

```
1   fun double(n: Int){
2       n = n * 2 // !Помилка - значення параметра не
    можна змінити
3       println( "Значення у функції double: $n" )
```

```
4   }
```

Однак якщо параметр є якимось складним об'єктом, то можна змінювати окремі значення в цьому об'єкті. Наприклад, розглянемо функцію, яка як параметр приймає масив:

```
1   fun double(numbers: IntArray){
2       numbers[0] = numbers[0] * 2
3       println( "Значення у функції double:
    ${numbers[0]}" )
4   }
5
6   fun main() {
7
8       var nums = intArrayOf(4, 5, 6)
9       double(nums)
10      println( "Значення у функції main: ${nums[0]}" )
11  }
```

В цьому прикладі функція **double** приймає числовий масив і збільшує значення першого елемента в два рази. При цьому зміна елемента масиву всередині функції призведе до того, що також буде змінено значення елемента в тому масиві, який передається як аргумент у функцію, оскільки це один і той самий масив. Результат роботи програми наведено на рисунку 3.3.

```
Значення у функції double: 8
Значення у функції main: 8
```

Рисунок 3.3. Результат роботи програми

3.2. Змінна кількість параметрів. Vararg

Функція може приймати різну кількість параметрів однакового типу. Для визначення таких параметрів використовується ключове слово **vararg**. Наприклад, необхідно передати у функцію кілька рядків, але скільки саме рядків зазделегіть точно не відомо. Тож їх може бути п'ять, шість, сім тощо:

```
1   fun printStrings(vararg strings: String){
2       for (str in strings)
3           println(str)
4   }
5   fun main() {
6
7       printStrings( "Taras" , "Volodymyr" , "Stepan" )
8       printStrings( "Kotlin" , "JavaScript" , "Java" ,
     "C#" , "C++" )
```

```
9   }
```

Функція **printStrings** приймає невизначену кількість рядків. У самій функції можна виконувати операції з параметром як із послідовністю рядків, наприклад, перебирати елементи послідовності в циклі та виконувати з ними деякі операції.

При виклику функції можна передати їй будь-яку кількість рядків.

Інший приклад це підрахунок суми невизначеної наперед кількості чисел:

```
1   fun sum(vararg numbers: Int){
2       var result=0
3       for (n in numbers)
4           result += n
5       println( "Сума чисел дорівнює $result" )
6   }
7   fun main() {
8
9       sum(1, 2, 3, 4, 5)
10      sum(1, 2, 3, 4, 5, 6, 7, 8, 9)
11  }
```

Якщо функція приймає кілька параметрів, то, зазвичай, **vararg-параметр** вказується останнім:

```
1  fun printUserGroup(count:Int, vararg users: String){
2      println( "Count: $count" )
3      for (user in users)
4          println(user)
5  }
6
7  fun main() {
8
9      printUserGroup(3, "Taras" , "Volodymyr" , "Olena" )
10 }
```

Однак це необов'язково, але якщо після **vararg-параметра** йдуть ще якісь параметри, то при виклику функції значення цим параметрам потрібно передавати за допомогою іменованих аргументів, наприклад:

```
1   fun printUserGroup(group:   String,   vararg   users:
    String, count:Int){
2       println( "Group: $group" )
3       println( "Count: $count" )
4       for (user in users)
5           println(user)
6   }
7   fun main() {
8
9       printUserGroup( "AEC-
    012" , "Taras" , "Volodymyr" , "Olena" , count=3)
```

```
10  }
```

В цьому прикладі функція **printUserGroup** приймає три параметри. Значення параметрів до параметра **vararg** передаються за позиціями. Тобто в цьому випадку **"AEC-012"** представлятиме значення для параметра **group**. Наступні значення інтерпретуються як значення для **vararg-параметра** аж до іменованих аргументів.

3.2.1. Оператор *

Оператор *** (spread operator**), який не варто плутати зі знаком множення, дозволяє передати параметр як значення елементів з масиву, наприклад:

```
1   fun changeNumbers(vararg numbers: Int, koef: Int){
2       for (number in numbers)
3           println(number * koef)
4   }
5   fun main() {
6
7       val nums = intArrayOf(1, 2, 3, 4)
8       changeNumbers(*nums, koef=2)
9   }
```

Варто звернути увагу на зірочку перед **nums** під час виклику функції: **changeNumbers(*nums , koef=2)**. Без застосування цього оператора згенерувалася б помилка, оскільки параметри функції не є масивом, а є невизначеною кількістю значень типу **Int**.

3.3. Повернення результату. Оператор return

Функція може повертати певний результат. У цьому випадку після списку параметрів через двокрапку вказується тип, що повертається. А в тілі функції застосовується оператор **return**, після якого вказується значення, що повертається.

Наприклад, визначимо функцію, яка повертає суму двох чисел:

```
1   fun sum(x:Int, y:Int): Int{
2
3       return x + y
4   }
5   fun main() {
6
7       val a = sum (4, 3)
8       val b = sum (5, 6)
9       val c = sum (6, 9)
10      println( "a=$a b=$b c=$c" )
11  }
```

Під час оголошення функції **sum** після списку параметрів через двокрапку вказується тип **Int**, який представлятиме тип значення, що повертається:

```
fun sum(x:Int, y:Int): Int
```

У самій функції за допомогою оператора **return** повертаємо отримане значення результат операції додавання:

```
return x + y
```

Оскільки функція повертає значення, то під час її виклику це значення можна присвоїти іншій змінній:

```
val a = sum (4, 3)
```

3.3.1. Тип Unit

Якщо функція не повертає ніякого результату, то фактично вона неявно повертає значення типу **Unit**. Цей тип аналогічний типу **void** у низці мов програмування, що повідомляє, що функція ні чого не повертає. Наприклад, наступна функція:

```
fun hello(){
    println( "Hello" )
}
```

буде аналогічна наведеній нище:

```
fun hello() : Unit{
    println( "Hello" )
}
```

Формально, навіть можна присвоїти результат такої функції іншій змінній, наприклад:

```
val d = hello()
val e = hello()
```

Однак практичного сенсу це не має, тому що значення, що повертається, представляє об'єкт **Unit**, який більше ніяк не застосовується.

Якщо функція повертає значення **Unit**, то можна також використовувати оператор **return** для виходу з функції, наприклад:

```
fun checkAge(age: Int){
    if (age < 0 | | age > 110) {
        println( "Invalid age" )
        return
    }
    println( "Age is valid" )
}
fun main() {

    checkAge(-10)
    checkAge(10)
```

```
12  }
```

У разі якщо значення параметра **age** виходить межі діапазону від 0 до 110, за допомогою оператора **return** здійснюється вихід із функції, і наступні інструкції не виконуються. Якщо функція повертає значення **Unit**, то після оператора **return** можна не вказувати ніякого значення.

3.4. Однорядкові та локальні функції

3.4.1. Однорядкові функції

Однорядкові функції (**single expression function**) використовують скорочений синтаксис визначення функції у вигляді одного виразу. Ця форма дозволяє не вказувати як тип, що повертається, так і оператор **return**, наприклад:

```
1   fun ім'я_функції (параметри_функції) = тіло_функції
```

Такі функції також визначаються за допомогою ключового слова **fun**, після якого зазначається ім'я функції та список параметрів. Але після списку параметрів не вказується тип, що повертається. Тип, що повертається, буде виводиться компілятором самостійно. Далі через оператор присвоєння визначається тіло функції у вигляді одного виразу.

Наприклад, функція піднесення числа у квадрат виглядатиме наступним чином:

```
1   fun square(x: Int) = x * x
2
3   fun main() {
4
5       val a = square (5) // 25
6       val b = square (6) // 36
7       println( "a=$a b=$b" )
8   }
```

У цьому прикладі функція **square** зводить число у квадрат. Вона складається з одного виразу **x * x**. Значення цього виразу і повертатиметься з функції. Також тут не використовується оператор **return**.

Такі функції більш лаконічні, більш наочні, але також опціонально можна і вказувати тип, що повертається явно, наприклад:

```
1   fun square(x: Int) : Int = x * x
```

3.4.2. Локальні функції

У мові Kotlin одні функції можна визначити всередині інших функцій. Внутрішні або вкладені функції називають локальними.

Локальні функції можуть визначати дії, які використовуються лише в межах якоїсь конкретної функції та ніде більше не застосовуються.

Наприклад, є функція, яка порівнює два значення віку між собою:

```
1   fun compareAge(age1: Int, age2: Int){
2
3       fun ageIsValid(age: Int): Boolean {
4           return age > 0 && age < 111
5       }
6       if ( !ageIsValid(age1) || !ageIsValid(age2)) {
7           println( "Invalid age" )
8           return
9       }
10
11      when {
12          age1 == age2 -> println( "age1 == age2" )
13          age1 > age2 -> println( "age1 > age2" )
14          age1 < age2 -> println( "age1 < age2" )
15      }
16  }
17  fun main() {
18
19      compareAge(20, 23)
20      compareAge(-3, 20)
21      compareAge(34, 134)
22      compareAge(15, 8)
23  }
```

Однак ззовні у функцію можуть бути передані некоректні дані. Чи є сенс порівнювати вік менше за нуль з іншим? Очевидно, ні. Для цієї мети у функції визначено локальну функцію **ageIsValid()**, яка повертає **true**, якщо вік є допустимим. Більше у програмі ця функція ніде не використовується, тому її можна зробити локальною.

При цьому локальна функція може використовуватися тільки в тій функції, де вона визначена.

До того ж, в наведеному прикладі, зручніше зробити локальну функцію однорядковою, наприклад:

```
1   fun compareAge(age1: Int, age2: Int){
2
3       fun ageIsValid(age: Int)= age > 0 && age < 111
4
5       if( !ageIsValid(age1) || !ageIsValid(age2)) {
6           println( "Invalid age" )
7           return
8       }
```

```
9
10      when {
11          age1 == age2 -> println( "age1 == age2" )
12          age1 > age2 -> println( "age1 > age2" )
13          age1 < age2 -> println( "age1 < age2" )
14      }
15  }
```

3.5. Перевантаження функцій

Перевантаження функцій (**function overloading**) є визначенням кількох функцій з одним і тим самим ім'ям, але з різними параметрами. Параметри перевантажених функцій можуть відрізнятися за кількістю, типом або порядком слідування у списку параметрів, наприклад:

```
1   fun sum(a: Int, b: Int) : Int{
2       return a + b
3   }
4   fun sum(a: Double, b: Double) : Double {
5       return a + b
6   }
7   fun sum(a: Int, b: Int, c: Int) : Int{
8       return a + b + c
9   }
10  fun sum(a: Int, b: Double) : Double {
11      return a + b
12  }
13  fun sum(a: Double, b: Int) : Double{
14      return a + b
15  }
```

В наведеному прикладі для однієї функції **sum()** визначено п'ять перевантажених версій. Кожна з версій відрізняється або за типом, або за кількістю, або за порядком слідування параметрів. При виклику функції **sum** компілятор в залежності від типу та кількості параметрів зможе вибрати для виконання потрібну версію функції самостійно, наприклад:

```
1   fun main() {
2
3       val a = sum(1, 2)
4       val b = sum (1.5, 2.5)
5       val c = sum(1, 2, 3)
6       val d = sum (2, 1.5)
7       val e = sum (1.5, 2)
8   }
```

Важливо пам’ятати, що при перевантаженні функції компілятор не враховує тип результату функції, що повертається. Наприклад, нехай визначено дві наступні версії функції **sum**:

```
fun sum(a: Double, b: Int) : Double{
    return a + b
}
fun sum(a: Double, b: Int) : String{
    return "$a + $b"
}
```

Вони збігаються у всьому за винятком типу, що повертається. Однак у цьому випадку буде згенеровано помилку, оскільки перевантажені версії повинні відрізнятися саме за типом, порядком або кількістю параметрів. Відмінність у типі, що повертається, не мають значення.

3.6. Тип функції

У мові Kotlin все є об'єктом, у тому числі і функції. Тож функції, як і інші об'єкти, мають певний тип. Тип функції визначається наступним чином:

```
(типи_параметрів) -> тип_що_повертається
```

Розглянемо функцію, яка не приймає жодних параметрів і нічого не повертає:

```
fun hello(){

    println( "Hello Kotlin" )
}
```

Вона має наступний тип:

```
() -> Unit
```

Якщо функція не приймає параметрів, то при визначенні типу вказуються порожні дужки, а як тип значення, що повертається вказується тип **Unit**.

Розглянемо іншу функцію:

```
fun sum(a: Int, b: Int): Int{
    return a + b
}
```

Ця функція приймає два параметри типу **Int** та повертає значення типу **Int**, тому вона має тип:

```
(Int, Int) -> Int
```

Отже, що дає знання типу функції? Використовуючи тип функції, можна визначати змінні та параметри інших функцій, які представлятимуть функції.

3.6.1. Змінні функції

У мові Kotlin змінна може бути функцією. За допомогою типу функції можна визначити, які саме функції змінна може представляти, наприклад:

```
fun main() {

    val message: () -> Unit
    message = ::hello
    message()
}

fun hello(){
    println( "Hello Kotlin" )
}
```

В цьому прикладі змінна **message** є функцією із типом **() -> Unit**, тобто функцію без параметрів, яка нічого не повертає. Далі визначена саме така функція **hello()**, відповідно можна передати функцію **hello** змінній **message**.

Щоб передати функцію, перед назвою функції необхідно розмістити оператор (**::**), наприклад:

```
message = ::hello
```

Потім можна звертатися до змінної **message()** як до звичайної функції, наприклад:

```
message()
```

Оскільки змінна **message** посилається на функцію **hello**, то під час виклику функції **message()** фактично буде викликатися функція **hello()**.

При цьому тип функції також може виводитися компілятором автоматично виходячи з присвоєного змінній значення, наприклад:

```
val message = ::hello // message має тип () -> Unit
```

Розглянемо інший приклад, коли змінна посилається на функцію з параметрами:

```
fun main() {

    val operation: (Int, Int) -> Int = :: sum
    val result = operation(3, 5)
    println(result) // 8
}
fun sum(a: Int, b: Int): Int{
    return a + b
}
```

Змінна **operation** представляє функцію з типом **(Int, Int) -> Int**, тобто функцію з двома параметрами типу **Int** і значенням типу **Int**, що повертається.

Відповідно, такій змінній можна присвоїти функцію **sum**, яка відповідає цьому типу.

Потім через ім'я змінної, фактично, можна звертатися до функції **sum()**, передаючи значення для параметрів і отримуючи її результат, наприклад:

```
1   val result = operation(3, 5)
```

При цьому можна динамічно змінювати значення, головне щоб воно відповідало типу змінної, наприклад:

```
1   fun main() {
2
3       // operation вказує на функцію sum
4       var operation: (Int, Int) -> Int = :: sum
5       val result1 = operation(14, 5)
6       println(result1) // 19
7
8       // operation вказує на функцію subtract
9       operation = ::subtract
10      val result2 = operation(14, 5)
11      println(result2) // 9
12
13  }
14  fun sum(a: Int, b: Int): Int{
15      return a + b
16  }
17  fun subtract(a: Int, b: Int): Int{
18      return a - b
19  }
```

3.7. Функції вищого порядку

Функції вищого порядку (**high order function**) - це функції, які або приймають функцію як параметр, або повертають функцію, або те, й інше.

3.7.1. Функція як параметр функції

Щоб функція могла приймати іншу функцію через параметр, цей параметр повинен представляти тип функції, наприклад:

```
1   fun main() {
2
3       displayMessage(::morning)
4       displayMessage(::evening)
5   }
6   fun displayMessage(mes: () -> Unit){
7       mes()
8   }
```

```
9   fun morning(){
10      println( "Good Morning" )
11  }
12  fun evening(){
13      println( "Good Evening" )
14  }
```

В наведеному прикладі, функція **displayMessage()** через параметр **mes** приймає функцію типу **() -> Unit**, тобто таку функцію, яка не має параметрів і нічого не повертає:, а саме

```
1   fun displayMessage(mes: () -> Unit){
```

Під час виклику цієї функції можна передати цьому параметру функцію, яка відповідає цьому типу, наприклад:

```
1   displayMessage(::morning)
```

Розглянемо наступний приклад параметра-функції, яка приймає параметри:

```
1   fun main() {
2
3       action(5, 3, :: sum) // 8
4       action(5, 3, ::multiply) // 15
5       action(5, 3, :: subtract) // 2
6   }
7
8   fun action (n1: Int, n2: Int, op: (Int, Int)-> Int) {
9       val result = op(n1, n2)
10      println(result)
11  }
12  fun sum(a: Int, b: Int): Int{
13      return a + b
14  }
15  fun subtract(a: Int, b: Int): Int{
16      return a - b
17  }
18  fun multiply(a: Int, b: Int): Int{
19      return a * b
20  }
```

В цьому прикладі функція **action** приймає три параметри. Перші два параметри мають значення типу **Int**. А третій параметр є функцією, яка має тип **(Int, Int)-> Int**, тобто приймає два числа і повертає деяке одне число.

У самій функції **action** викликається ця параметр-функція, передаючи їй два числа, а результат виводиться на консоль.

Під час виклику функції **action** можна передати для третього параметра конкретну функцію, яка відповідає цьому параметру за типом, наприклад:

```
1   action(5, 3, :: sum) // 8
2   action(5, 3, ::multiply) // 15
```

```
3   action(5, 3, :: subtract) // 2
```

3.7.2. Повернення функції із функції

У окремих випадках може знадобитися повернути функцію з іншої функції. В цьому разі для функції як тип, що повертається, встановлюється тип іншої функції. А в тілі функції повертається лямбда-вираз, наприклад:

```
1   fun main() {
2       val action1 = selectAction(1)
3       println(action1(8,5)) // 13
4
5       val action2 = selectAction(2)
6       println(action2(8,5)) // 3
7   }
8   fun selectAction(key: Int): (Int, Int) -> Int{
9       // Визначення результату, що повертається
10      when (key) {
11          1 -> return :: sum
12          2 -> return :: subtract
13          3 -> return ::multiply
14          else -> return ::empty
15      }
16  }
17  fun empty (a: Int, b: Int): Int{
18      return 0
19  }
20  fun sum(a: Int, b: Int): Int{
21      return a + b
22  }
23  fun subtract(a: Int, b: Int): Int{
24      return a - b
25  }
26  fun multiply(a: Int, b: Int): Int{
27      return a * b
28  }
```

В цьому прикладі, функція **selectAction** приймає один параметр **key**, який представляє тип **Int**. В якості типу функції, що повертається, вказаний тип **(Int, Int) -> Int**. Тобто **selectAction** повертатиме певну функцію, яка приймає два параметри типу **Int** і повертає об'єкт типу Int.

У тілі функції **selectAction** залежно від значення параметра **key** повертається певна функція, що відповідає типу **(Int, Int) -> Int**.

Далі у функції **main** визначається змінна **action1**, яка зберігає результат функції **selectAction**. Оскільки **selectAction()** повертає функцію, то і змінна

action1 буде зберігати цю функцію. Потім через змінну **action1** можна викликати цю функцію.

Оскільки функція, що повертається, відповідає типу **(Int, Int) -> Int**, то під час виклику в **action1** необхідно передати два числа і відповідно можна отримати результат та вивести його на консоль.

3.8. Анонімні функції

Анонімні функції виглядають як звичайні за тим винятком, що вони не мають імені. Анонімна функція може бути записана одним виразом, наприклад:

```
fun (x: Int, y: Int): Int = x + y
```

Або може містити блок коду, наприклад:

```
fun (x: Int, y: Int): Int {
    return x + y
}
```

Анонімну функцію можна передавати як значення змінної, наприклад:

```
fun main() {

    val message = fun ()=println( "Hello" )
    message()
}
```

В цьому прикладі змінній **message** передається анонімна функція **fun()=println("Hello")**. Ця анонімна функція не приймає параметрів, а тільки виводить на консоль рядок "**Hello**". Таким чином, змінна **message** буде представляти тип **() -> Unit**.

Далі можна викликати цю функцію через ім'я змінної як звичайну функцію, тобто **message()**.

Розглянемо інший приклад анонімної функції з параметрами, а саме:

```
fun main() {

    val sum = fun (x: Int, y: Int): Int = x + y
    val result = sum(5, 4)
    println(result) // 9
}
```

В цьому прикладі змінній **sum** присвоюється анонімна функція, яка приймає два параметри: два цілих числа типу **Int**, та повертає їхню суму.

Також через ім'я змінної можна викликати цю анонімну функцію, передавши деякі значення для параметрів, та отримати її результат, використавши конструкцію типу **val result = sum(5, 4)**.

3.8.1. Анонімна функція як аргумент функції

Анонімну функцію можна передавати до функції, якщо параметр відповідає типу цієї функції, наприклад:

```
1   fun main() {
2
3       doOperation(9,5, fun (x: Int, y: Int): Int = x +
    y) // 14
4       doOperation(9,5, fun (x: Int, y: Int): Int = x -
    y) // 4
5
6       val op = fun (x: Int, y: Int): Int = x * y
7       doOperation(9, 5, op) // 45
8   }
9   fun doOperation(x: Int, y: Int, op: (Int, Int) -
    >Int){
10
11      val result = op(x, y)
12      println(result)
13  }
```

3.8.2. Повернення анонімної функції із функції

У мові Kotlin функція також може повертати анонімну функцію як результат, наприклад:

```
1   fun main() {
2
3       val action1 = selectAction(1)
4       val result1 = action1(4, 5)
5       println(result1) // 9
6
7       val action2 = selectAction(3)
8       val result2 = action2(4, 5)
9       println(result2) // 20
10
11      val action3 = selectAction(9)
12      val result3 = action3(4, 5)
13      println(result3) // 0
14  }
15
16  fun selectAction(key: Int): (Int, Int) -> Int{
17      // Визначення результату, що повертається
18      when (key) {
19          1 -> return fun (x: Int, y: Int): Int = x + y
20          2 -> return fun (x: Int, y: Int): Int = x - y
```

```
21          3 -> return fun (x: Int, y: Int): Int = x * y
22          else -> return fun(x: Int, y: Int): Int = 0
23      }
24  }
```

В цьому прикладі функція **selectAction()** залежно від переданого значення, повертає одну з чотирьох анонімних функцій. Остання анонімна функція **fun(x: Int, y: Int): Int = 0** повертає число 0.

При зверненні до функції **selectAction()** змінна отримає певну анонімну функцію, а саме:

```
1   val action1 = selectAction(1)
```

Тобто в цьому разі змінна **action1** зберігає посилання на функцію **fun(x: Int, y: Int): Int = x + y**.

3.9. Лямбда-вирази

Лямбда-вирази є невеликими шматочками коду, які виконують деякі дії. Фактично лямбда-вирази представляють скорочений запис функцій. При цьому лямбда-вирази, як і звичайні та анонімні функції, можуть передаватися як значення змінних і параметрів функції.

Лямбда-вирази обертаються у фігурні дужки, а саме:

```
1   {println( "Hello Kotlin " )}
```

У цьому прикладі лямбда-вираз виводить на консоль рядок "**Hello Kotlin**".

Лямбда-вираз можна зберегти у звичайній змінній і потім викликати через ім'я цієї змінної як звичайну функцію, наприклад:

```
1   fun main() {
2
3       val hello = {println( "Hello Kotlin" )}
4       hello()
5       hello()
6   }
```

У цьому прикладі лямбда-вираз збережено у змінній **hello** і через цю змінну викликається двічі. Оскільки лямбда-вираз є скороченою формою функції, то змінна **hello** має тип функції, тобто **() -> Unit**, а саме:

```
1   val hello: ()->Unit = {println( "Hello Kotlin" )}
```

Також лямбда-вираз можна запускати як звичайну функцію, використовуючи круглі дужки, наприклад:

```
1   fun main() {
2
3       {println( "Hello Kotlin" )}()
4   }
```

Слід враховувати, що якщо до подібного запису зазначені будь-які інструкції, Kotlin автоматично може не визначити, що записаний лямбда-вираз становить нову інструкцію. У цьому випадку попередню інструкцію можна завершити крапкою з комою (**;**), наприклад:

```
1   fun main() {
2
3       {println( "Hello Kotlin" )}();
4       {println( "Kotlin on Sunshine.com" )}()
5   }
```

3.9.1. Передача параметрів

Лямбда-вирази як функції можуть приймати параметри. Для передачі параметрів використається символ стрілки (->). Параметри вказуються ліворуч від стрілки, а тіло лямбда-виразу, тобто самі операції, що виконуються, праворуч від стрілки, наприклад:

```
1   fun main() {
2
3       val printer = {message: String ->
    println(message)}
4       printer( "Hello" )
5       printer( "Good Bye" )
6   }
```

В цьому прикладі лямбда-вираз приймає один параметр типу **String**, значення якого виводиться на консоль. Змінна **printer** у цьому прикладі має тип **(String) -> Unit**.

Під час виклику лямбда-виразу відразу при його визначенні в дужках вказуються значення його параметрів, наприклад:

```
1   fun main() {
2
3       {message: String -> println(message)}( "Welcome
    to Kotlin" )
4   }
```

Якщо параметрів кілька, то вони вказуються зліва від стрілки через кому, наприклад:

```
1   fun main() {
2
3       val sum = {x:Int, y:Int -> println(x + y)}
4       sum(2, 3) // 5
5       sum(4, 5) // 9
6   }
```

Якщо у лямбда-виразі необхідно виконати не одну, а кілька операцій, то ці операції можна розташовувати в окремих рядках після символу стрілки(->), наприклад:

```
1   val sum = {x:Int, y:Int ->
2       val result = x + y
3       println( "$x + $y = $result" )
4   }
```

3.9.2. Повернення результату

Вираз, який зазначається після символу стрілки (->), визначає результат лямбда-виразу. Цей результат можна потім, наприклад, присвоїти іншій змінній.

Якщо лямбда-вираз формально не повертає жодного результату, то фактично, як і у функціях, повертається значення типу **Unit**, наприклад:

```
1   val hello = { println( "Hello" )}
2   val h = hello() // h має тип Unit
3
4   val printer = {message: String -> println(message)}
5   val p = printer( "Welcome" ) // p має тип Unit
```

В обох випадках використовується функція **println**, яка формально не повертає жодного значення, точніше повертає об'єкт типу **Unit**.

Але також може повертатися конкретне значення, наприклад:

```
1   fun main() {
2
3       val sum = {x:Int, y:Int -> x + y}
4
5       val a = sum (2, 3) // 5
6       val b = sum (4, 5) // 9
7       println( "a=$a b=$b" )
8   }
```

В цьому прикладі, вираз праворуч від символу стрілки **x+y** продукує нове значення – суму чисел, і при виклику лямбда-виразу це значення можна передати іншій змінній. У цьому прикладі лямбда-вираз має тип **(Int, Int) -> Int**.

Якщо лямбда-вираз багаторядковий, тобто складається з декількох інструкцій, то повертається значення, яке генерується останньою інструкцією, наприклад:

```
1   val sum = {x:Int, y:Int ->
2       val result = x + y
3       println( "$x + $y = $result" )
4       result
5   }
```

Останній вираз по суті представляє число, а саме суму чисел **x** і **y**, і воно буде повертатися як результат лямбда-виразу.

3.9.3. Лямбда-вирази як аргументи функцій

Лямбда-вирази можна передавати параметрам функції, якщо вони представляють один і той самий тип функції, наприклад:

```
1   fun main() {
2
3       val sum = {x:Int, y:Int -> x + y}
4       doOperation(3, 4, sum) // 7
5       doOperation(3, 4, {a:Int, b: Int -> a * b}) // 12
6   }
7   fun doOperation(x:  Int,  y:  Int,  op:  (Int,  Int)  -
8   >Int){
9
10      val result = op(x, y)
11      println(result)
12  }
```

3.9.4. Типізація параметрів лямбда-виразів

При передачі лямбда-виразу параметру або змінній, для якої явно зазначений тип, можна в лямбда-виразі не вказувати типи параметрів, наприклад:

```
1   fun main() {
2       val sum: (Int, Int) -> Int = {x, y -> x + y}
3       doOperation(3, 4, {a, b -> a * b})
4   }
5   fun doOperation(x: Int, y: Int, op: (Int, Int) ->Int){
6
7       val result = op(x, y)
8       println(result)
9   }
```

В цьому випадку Kotlin бачить, що тип змінної **sum** є **(Int, Int) -> Int**, тобто і перший, і другий параметр представляють тип **Int**. Тому при присвоєнні змінній лямбда-виразу **{x, y -> x + y}** Kotlin автоматично зрозуміє, що параметри **x** та **y** мають саме тип **Int**.

Те ж саме стосується й виклику функції **doOperation()** при передачі в неї лямбда-виразу Kotlin автоматично зрозуміє, який параметр є яким типом.

3.9.5. Trailing lambda

Якщо параметр, який приймає функцію, є останнім у списку, то при передачі лямбда-виразу, сам лямбда-вираз можна прописати після списку параметрів. Наприклад, візьмемо вище використану функцію **doOperation()**:

```
fun doOperation(x: Int, y: Int, op: (Int, Int)
->Int){

    val result = op(x, y)
    println(result)
}
```

В цьому прикладі параметр, який представляє функцію **op** є останнім у списку параметрів. Тому замість того, щоб написати так:

```
doOperation(3, 4, {a, b -> a * b}) // 12
```

також можна написати так:

```
doOperation(3, 4) {a, b -> a * b} // 12
```

Тобто можна винести лямбда-вираз за список параметрів. Це так звана кінцева лямбда або **trailing lambda**.

3.9.6. Повернення лямбда-виразу з функції

Також у мові Kotlin функція може повертати лямбда-вираз, який відповідає типу її результату, що повертається, наприклад:

```
fun main() {
    val action1 = selectAction(1)
    val result1 = action1(4, 5)
    println(result1) // 9

    val action2 = selectAction(3)
    val result2 = action2(4, 5)
    println(result2) // 20

    val action3 = selectAction(9)
    val result3 = action3(4, 5)
    println(result3) // 0
}
fun selectAction(key: Int): (Int, Int) -> Int{
    // Визначення результату, що повертається
    when (key) {
        1 -> return {x, y -> x + y}
        2 -> return {x, y -> x - y}
        3 -> return {x, y -> x * y}
        else -> return {x, y -> 0}
    }
```

```
22  }
```

3.9.7. Параметри, що не використовуються

Звернемо увагу у попередньому приклади на останній лямбда-вираз, а саме:

```
1   else -> return {x, y -> 0}
```

Якщо у функцію **selectAction()** передається число, відмінне від 1, 2 або 3, то повертається лямбда-вираз, який просто повертає число 0. З одного боку цей лямбда-вираз має відповідати типу результату, що повертається, функцією **selectAction() - (Int, Int) -> Int**, а з іншого боку, він не використовує параметри, ці параметри не потрібні. У цьому випадку замість параметрів, що не використовуються, можна вказати символ підкреслення (**_**):

```
1   else -> return {_, _ -> 0 }
```

Резюме

В наведеному розділі розглядається функціональне програмування в Kotlin, зокрема, робота з функціями. Показано, як оголошувати функції, передавати їм параметри та використовувати значення за замовчуванням. Описано іменовані аргументи, змінні параметри (**vararg**) та оператор `*`, що допомагає передавати масиви у функції. Розглянуто повернення значень за допомогою **return**, а також використання однорядкових і локальних функцій. Крім того, пояснюється перевантаження функцій, робота з анонімними функціями та лямбда-виразами. Також важливим аспектом є функції високого порядку, які можуть приймати інші функції або повертати їх.

Контрольні запитання і завдання

1. Як оголошується функція в Kotlin?
2. Чи обов’язково функція повинна мати параметри?
3. Як викликати функцію у програмі?
4. Що таке функції верхнього рівня (top-level functions)?
5. Як передавати параметри у функцію?
6. Що таке аргументи за замовчуванням і як їх використовувати?
7. Як працюють іменовані аргументи у Kotlin?
8. Чому параметри функції є val-змінними?
9. Що таке **vararg** і для чого він використовується?
10. Як працює оператор ***** у функціях із vararg?
11. Як повернути значення з функції у Kotlin?
12. Що таке тип **Unit** і коли він використовується?

13. Що таке однорядкові функції? Наведіть приклад.
14. Яка різниця між локальними та звичайними функціями?
15. Як працює перевантаження функцій у Kotlin?
16. Як змінна може містити функцію?
17. Що таке функції високого порядку?
18. Як функція може повертати іншу функцію?
19. У чому різниця між анонімними функціями та лямбда-виразами?
20. Що таке **trailing lambda**, і коли вона використовується?
21. Напишіть функцію **greet(name: String)**, яка приймає ім'я користувача та виводить привітання у форматі «Привіт, [ім'я]!».
22. Створіть функцію **sum(a: Int, b: Int): Int**, яка приймає два числа та повертає їхню суму.
23. Реалізуйте функцію **printUserInfo(name: String, age: Int = 18)**, яка виводить інформацію про користувача. Перевірте, як вона працює з різною кількістю аргументів.
24. Напишіть функцію **multiply(vararg numbers: Int)**, яка приймає довільну кількість чисел та виводить їхній добуток.
25. Реалізуйте функцію **getMax(a: Int, b: Int): Int** у скороченій (однорядковій) формі, яка повертає більше з двох чисел.
26. Створіть локальну функцію всередині іншої функції, яка перевіряє, чи число є парним.
27. Перевантажте функцію **printMessage**, щоб вона могла приймати або рядок, або число.
28. Напишіть функцію **applyOperation(a: Int, b: Int, operation: (Int, Int) -> Int)**, яка приймає два числа та функцію-операцію (наприклад, додавання чи множення) і повертає результат.
29. Створіть анонімну функцію, яка приймає два числа та повертає їхній добуток. Присвойте її змінній та викличте.
30. Реалізуйте функцію **selectOperation(type: String): (Int, Int) -> Int**, яка повертає одну з трьох функцій (додавання, віднімання або множення) залежно від переданого рядка.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Яке ключове слово використовується для оголошення функції в Kotlin?
а) function

b) def
c) fun
d) fn
Правильна відповідь: c).

Чому варто використовувати named arguments (іменовані аргументи)?
a) Це змушує всі параметри бути обов’язковими
b) Це дозволяє передавати параметри у будь-якому порядку
c) Це не рекомендовано у Kotlin
d) Це потрібно лише для функцій із vararg
Правильна відповідь: b).

Який із варіантів є правильним лямбда-виразом у Kotlin?
a) { x, y -> x + y }
b) fun (x, y) => x + y
c) (x, y) -> x + y
d) lambda(x, y) { return x + y }
Правильна відповідь: a).

Закриті запитання з кількома правильними відповідями

Які твердження про функції в Kotlin правильні? (Виберіть кілька варіантів)
a) Функція в Kotlin може мати значення за замовчуванням
b) Параметри функції можна змінювати всередині функції
c) Функція може бути передана як аргумент іншій функції
d) Функція обов’язково має повертати значення
Правильна відповідь: a), c).

Завдання на відповідність

Співвіднесіть поняття з їхніми описами:
a) vararg → 1. Використовується для повернення значень із функції
b) return → 2. Дозволяє передавати змінну кількість аргументів у функцію
c) Unit → 3. Тип, що вказує на відсутність поверненого значення
Правильна відповідь:
a) → 1.
b) → 2.
c) → 3.

Вставити пропущені слова

Яке ключове слово використовується для виходу з функції та повернення значення?

_____

Правильна відповідь: return

Завдання з кодом

Що не так у цьому коді?

```
fun greet(name: String) {
    println("Hello, " + name)
    return name
}
```

a) Проблема з типами, функція повинна повертати String
b) Не можна використовувати return у функціях
c) Код працює правильно
d) println не підтримує конкатенацію рядків
Правильна відповідь: a).

Що виведе цей код?

```
fun double(x: Int): Int = x * 2

fun main() {
    println(double(4))
}
```

a) 4
b) 8
c) 44
d) Помилка компіляції
Правильна відповідь: b).

Що потрібно змінити в коді, щоб він працював правильно?

```
fun multiply(a: Int, b: Int): Int {
    println(a * b)
}
```

a) Додати return перед a * b
b) Видалити println
c) Змінити Int на Unit
d) Код працює правильно

Правильна відповідь: а).

Що виведе цей код?

```
fun printMessage(message: String = "Hello!") {
    println(message)
}

fun main() {
    printMessage()
}
```

a) "Hello!"
b) ""
c) null
d) Виникне помилка компіляції

Правильна відповідь: а).

4. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

4.1. Класи та об'єкти

Мова Kotlin підтримує об'єктно-орієнтовану парадигму програмування, а це означає, що програму цією мовою можна представити у вигляді об'єктів, що взаємодіють між собою.

Уособленням об'єкта є **клас**. Клас фактично надає визначення об'єкта. А об'єкт є безпосереднім втіленням класу. Наприклад, у всіх є деяке уявлення про автомобіль, наприклад, кузов, чотири колеса, кермо тощо – тобто, деякий загальний набір характеристик, властивих кожному автомобілю. Це уявлення власне і є класом. При цьому є різні автомобілі, які мають різну форму кузова, якісь інші деталі, тобто конкретні втілення цього класу, конкретні об'єкти або екземпляри класу.

Для визначення класу у мові Kotlin застосовується ключове слово **class**, після якого вказується ім'я класу. Після імені класу у фігурних дужках визначається тіло класу. Якщо клас не має тіла, то фігурні дужки можна не писати. Наприклад, визначимо клас, який описує людину:

```
1   class Person
2
3   // або можна так
4   class Person { }
```

Клас фактично є новим типом даних, тому можна визначати змінні цього типу, наприклад:

```
1   fun main() {
2
3       val taras: Person
4       val volodymyr: Person
5       val olena: Person
6   }
7
8   class Person
```

У функції **main** визначено три змінні типу **Person**. Варто також відзначити, що на відміну від інших об'єктно-орієнтованих мов, таких як C# або Java, функція **main** у мові Kotlin не включається в окремих клас, а завжди визначається поза будь-яким класом.

Для створення об'єкта класу необхідно викликати конструктор цього класу. Конструктор фактично є функцією з ім’ям, яке збігається з ім'ям класу і яка виконує ініціалізацію об'єкта. За замовчуванням компілятор у мові Kotlin генерує порожній конструктор, який відразу можна використовувати, наприклад:

```
1   val taras: Person = Person()
```

Частина коду після знаку рівності **Person()** якраз і є викликом конструктора, який створює об'єкт класу **Person**. До виклику конструктора змінна класу не вказує на жодний об'єкт.

Приклад створення трьох об’єктів класу Person наведено нижче:

```
1   fun main() {
2
3       val taras: Person = Person()
4       val volodymyr: Person = Person()
5       val olena: Person = Person()
6   }
7
8   class Person
```

4.1.1. Властивості

Кожен клас може зберігати деякі дані або стан у вигляді властивостей. Властивостями є змінні, визначені на рівні класу з ключовими словами **val** чи **var**. Якщо властивість визначено за допомогою ключового слова **val**, то значення такої властивості можна встановити лише один раз, тобто вона є незмінюваною **(immutable)**. Якщо властивість визначено з допомогою ключового слова **var**, то значення цієї властивості можна змінювати багаторазово.

Властивість має бути ініціалізована, тобто обов'язково повинна мати початкове значення. Наприклад, визначимо декілька властивостей:

```
1   class Person{
2       var name: String = "Undefined"
3       var age: Int = 18
4   }
```

В наведеному прикладі в класі **Person**, який описує людину, визначено властивості **name** (ім'я людини) та **age** (вік людини). І ці властивості ініціалізовані початковими значеннями.

Оскільки ці властивості визначені за допомогою ключового слова **var**, то зміну їх початкових значень дозволено, наприклад:

```
1   fun main() {
2
3       val volodymyr: Person = Person () // створюємо
    об'єкт
4       println(volodymyr.name) // Undefined
5       println(volodymyr.age) // 18
6
7       volodymyr.name = "Volodymyr"
8       volodymyr.age = 25
```

```
9
10      println(volodymyr.name) // Volodymyr
11      println(volodymyr.age) // 25
12  }
13
14  class Person{
15      var name: String = "Undefined"
16      var age: Int = 18
17  }
```

Для звернення до властивостей використовується ім'я змінної, яка є об'єктом, і після крапки (**.**) вказується ім'я властивості. Приклад отримання значення властивості наведено нижче:

```
1   val personName : String = volodymyr.name
```

Приклад встановлення значення властивості:

```
1   volodymyr.name = "Volodymyr"
```

4.1.2. Функції класу

Клас може містити функції. Функції визначають поведінку об'єктів цього класу. Такі функції ще називають **member functions**, або функції-члени класу. Наприклад, визначимо клас із функціями:

```
1   class Person{
2       var name: String = "Undefined"
3       var age: Int = 18
4
5       fun sayHello(){
6           println( "Hello, my name is $name" )
7       }
8
9       fun go(location: String){
10          println( "$name goes to $location" )
11      }
12
13      fun personToString() : String{
14          return "Name: $name Age: $age"
15      }
16  }
```

Функції класу визначають як і звичайні функції. Зокрема в цьому прикладі, у класі **Person** визначено функцію **sayHello()**, яка виводить на консоль рядок **"Hello"** та емулює вітання об'єкта **Person**. Друга функція - **go()** емулює рух об'єкта **Person** до певного місця призначення. Місце призначення передається через параметр **location**. І третя функція - **personToString()** повертає інформацію про поточний об'єкт у вигляді рядка.

У функціях, визначених усередині класу, доступні властивості цього класу. Так, у цьому прикладі, у функціях можна звернутися до властивостей **name** та **age**, які визначені у класі **Person**.

Для звернення до функцій класу необхідно використовувати ім'я об'єкта, після якого йде назва функції та у дужках значення для параметрів цієї функції, наприклад:

```
1   fun main() {
2
3       val taras = Person()
4       taras.name = "Taras"
5       taras.age = 37
6
7       taras.sayHello()
8       taras.go( "the shop" )
9       println(taras.personToString())
10
11  }
12
13  class Person{
14      var name: String = "Undefined"
15      var age: Int = 18
16
17      fun sayHello(){
18          println( "Hello, my name is $name" )
19      }
20
21      fun go(location: String){
22          println( "$name goes to $location" )
23      }
24
25      fun personToString() : String{
26          return "Name: $name Age: $age"
27      }
28  }
```

Результат роботи програми наведено на рисунку 4.1.

```
Hello, my name is Taras
Taras goes to the shop
Name: Taras Age: 37
```

Рисунок 4.1. Результат роботи програми

4.2. Конструктори

Для створення об'єкта потрібно викликати конструктор класу. За замовчуванням компілятор створює конструктор, який не приймає параметрів і який відразу можна використовувати. Але також існує можливість визначати власні конструктори. Для визначення власних конструкторів застосовується ключове слово **constructor**.

Класи в мові Kotlin можуть мати один **первинний конструктор (primary constructor)** та один або кілька **вторинних конструкторів (secondary constructor)**.

4.2.1. Первинний конструктор

Первинний конструктор є частиною заголовка класу та визначається відразу після імені класу, наприклад:

```
1   class Person constructor(_name: String){
2
3   }
```

Конструктори, як і стандартні функції, можуть містити параметри. Так, у наведеному прикладі, конструктор має параметр **_name**, який має тип **String**. Через параметри конструктора можна передати дані ззовні і використовувати їх для ініціалізації об'єкта. При цьому, первинний конструктор на відміну від функцій не визначає та не виконує жодних дій, він може тільки приймати дані ззовні через параметри.

Якщо первинний конструктор не має жодних анотацій чи модифікаторів доступу, як у наведеному прикладі, то ключове слово **constructor** можна не писати, наприклад:

```
1   class Person(_name: String){
2
3   }
```

4.2.2. Ініціалізатор

Розглянемо що можна робити із отриманими через конструктор даними. У мові Kotlin можна їх використовувати для ініціалізації властивостей класу. Для цього застосовуються блоки ініціалізаторів, наприклад:

```
1   class Person(_name: String){
2       val name: String
3       init{
4           name = _name
5       }
```

```
6   }
```

У класі **Person** визначено властивість **name**, що зберігає ім'я. Щоб передати цій властивості значення **_name** з первинного конструктора, застосовується блок ініціалізатора. Блок ініціалізатора визначається після ключового слова **init**.

Мета ініціалізації полягає у встановленні первинних значень властивостей об'єкта під час його первинного створення. Варто відзначити, що тут властивості **name** не задається початкове значення, тому ця властивість у будь-якому випадку буде ініціалізована в блоці ініціалізатора, і при створенні об'єкта вона в будь-якому разі набуде значення.

Отже, тепер можемо використовувати первинний конструктор класу для створення об'єкта, наприклад:

```
1   fun main() {
2       val taras = Person( "Taras" )
3       val volodymyr = Person( "Volodymyr" )
4       val olena = Person( "Olena" )
5
6       println(taras.name) // Taras
7       println(volodymyr.name) // Volodymyr
8       println(olena.name) // Olena
9   }
10
11  class Person(_name: String){
12      val name: String
13      init{
14          name = _name
15      }
16  }
```

Важливо враховувати, що в разі визначення первинного конструктора, використання конструктору за замовчуванням, який генерується компілятором автоматично, вже не можна. Тобто, для створення об'єкта обов'язково потрібно використовувати первинний конструктор, якщо він визначений у класі.

Варто зазначити, що у класі може бути визначено кілька блоків ініціалізатора.

Також варто відзначити, що в наведеному прикладі, в ініціалізаторі немає особливого сенсу, оскільки параметри первинного конструктора можна передавати властивостям безпосередньо, наприклад:

```
1   class Person(_name: String){
2
3       val name: String = _name
4   }
```

4.2.3. Первинний конструктор та властивості

Первинний конструктор також може використовуватися для визначення властивостей, наприклад:

```
1   fun main() {
2
3       val volodymyr: Person = Person( "Volodymyr" , 23)
4
5       println( "Name: ${volodymyr.name} Age: ${
    volodymyr.age}" )
6   }
7
8   class Person( val name: String, var age: Int) {
9
10  }
```

Властивості визначаються так само як і параметри, причому їх визначення починається з ключового слова **val**, якщо їх не планується змінювати, і **var**, якщо властивості повинні бути змінюваними. В цьому випадку вже необов'язково явно визначати ці властивості в тілі класу, так як їх вже визначає конструктор. І під час виклику конструктора цим властивостям автоматично передаються значення: **Person("Volodymyr", 23)**.

4.2.4. Вторинні конструктори

Клас може визначати вторинні конструктори. Вони застосовуються в основному, щоб визначити додаткові параметри, за допомогою яких можна передавати дані для ініціалізації об'єкта.

Вторинні конструктори визначаються у тілі класу. Якщо у класі визначено первинний конструктор, то вторинний конструктор повинен викликати первинний за допомогою ключового слова **this**, наприклад:

```
1   class Person(_name: String){
2       val name: String = _name
3       var age: Int = 0
4
5       constructor(_name:      String,      _age:      Int)
    : this (_name){
6           age = _age
7       }
8   }
```

В цьому прикладі у класі **Person** визначено первинний конструктор, який отримує значення для встановлення властивості **name**:

```
1   class Person(_name: String)
```

Також у цьому прикладі доданий вторинний конструктор. Він приймає два параметри: **_name** та **_age**. За допомогою ключового слова **this** викликається первинний конструктор, тому через цей виклик необхідно передати значення для параметрів первинного конструктора. Зокрема, у первинний конструктор передається значення параметра **_name**. А у самому вторинному конструкторі встановлюється значення властивості **age**:

```
constructor(_name: String, _age: Int) : this (_name){
    age = _age
}
```

Таким чином, під час виклику вторинного конструктора спочатку викликається первинний конструктор, в якому спрацьовує блок ініціалізатора, який встановлює властивість **name**. А потім виконуються інструкції власне вторинного конструктора, який встановлює властивість **age**.

Розглянемо приклад використання наведеної вище модифікації класу **Person**:

```
fun main() {

    val taras: Person = Person( "Taras" )
    val volodymyr: Person = Person( "Volodymyr" , 45)

    println( "Name: ${taras.name} Age: ${taras.age}" )
    println( "Name: ${volodymyr.name} Age: ${volodymyr.age}" )
}

class Person(_name: String){
    val name: String = _name
    var age: Int = 0

    constructor(_name: String, _age: Int) : this (_name){
        age = _age
    }
}
```

В наведеній програмі, у функції **main** створюються два об'єкти **Person**. Для створення об'єкта **taras** застосовується первинний конструктор, що приймає один параметр. Для створення об'єкта **volodymyr** застосовується вторинний конструктор із двома параметрами.

Результат роботи програми наведений на рисунку 4.2.

```
Name: Taras Age: 0
Name: Volodymyr Age: 45
```

Рисунок 4.2. Результат роботи програми

За необхідності можна визначати і більше вторинних конструкторів, наприклад:

```
fun main() {

    val taras = Person( "Taras" )
    val volodymyr = Person( "Volodymyr" , 41)
    val stepan = Person( "Stepan" , 32, "JetBtains" )

    println( "Name:                 ${taras.name} Age: ${taras.age} Company: ${taras.company}" )
    println( "Name:              ${volodymyr.name} Age: ${volodymyr.age} Company: ${volodymyr.company}" )
    println( "Name:                ${stepan.name} Age: ${stepan.age} Company: ${stepan.company}" )
}

class Person(_name: String){
    val name = _name
    var age: Int = 0
    var company: String = "Undefined"

    constructor(_name:     String,     _age:     Int) : this (_name){
        age = _age
    }

    constructor(_name:  String,  _age:  Int,  _comp: String) : this (_name, _age){
         company = _comp
    }
}
```

В наведеному коді у клас **Person** додано нову властивість - **company**, яка описує компанії, в якій працює людина. А також доданий ще один конструктор, який приймає три параметри:

```
constructor(_name: String, _age: Int, _comp: String) : this (_name, _age){
    company = _comp
}
```

Щоб не дублювати код ініціалізації значень властивостей **name** і **age**, цей вторинний конструктор передає ініціалізацію цих властивостей іншому вторинному конструктору, який приймає два параметри через виклик **this(_name, _age)**. Тобто цей виклик по суті викликатиме перший вторинний конструктор із двома параметрами.

Результат роботи програми наведений на рисунку 4.3.

```
Name: Taras Age: 0 Company: Undefined
Name: Volodymyr Age: 41 Company: Undefined
Name: Stepan Age: 32 Company: JetBtains
```

Рисунок 4.3.Результат роботи програми

4.3. Пакети та імпорт

Пакети в мові Kotlin уособлюють логічний блок, який поєднує деякий функціонал, наприклад, класи та функції, що використовуються для вирішення близьких за характером завдань. Так, класи і функції, які призначені для вирішення одного завдання, можна розташувати в один пакет, а класи і функції для інших завдань можна розташувати в інші пакети.

Для оголошення пакета застосовується ключове слово **package**, після якого вказується ім'я пакета, наприклад:

```
1   package email
```

Визначення пакета розташовується на початку файлу. Весь вміст цього файлу розглядатиметься як вміст цього пакета.

Наприклад, додамо до проєкту новий файл **email.kt**, як показано на рисунку 4.4.

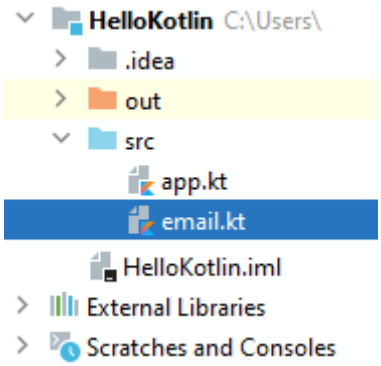

Рисунок 4.4. Додавання нового пакета до проєкту

Створимо в ньому наступний код:

```
1   package email
2
3   class Message( val text: String)
```

```
4
5   fun send(message: Message, address: String){
6       println( "Message `${message.text}` has been sent
    to $address" )
7   }
```

В цьому прикладі створений новий пакет називається "**email**". Він містить клас **Message**, який містить одну властивість **text**. Умовно кажучи, цей клас позначає email-повідомлення, а властивість **text** –текст email-повідомлення.

Також у цьому пакеті визначено функцію **send()**, яка умовно надсилає повідомлення на певну адресу.

Припустимо, необхідно використовувати функціонал цього пакета в іншому файлі. Для підключення сутностей із пакета необхідно застосувати директиву **import**.

У мові Kotlin існують різні способи підключення функціоналу пакета. Можна підключити весь пакет, наприклад:

```
1   import email.*
```

В цьому прикладі після назви пакета записано символ крапки (**.**) і зірочка (*****), що позначає виконання імпорту всіх типів цього пакета. Наприклад, візьмемо інший файл проєкту - ***app.kt***, який визначає функцію **main**, і використаємо у ньому функціонал пакета **email**, як наведено нище:

```
1   import email.*
2
3   fun main() {
4
5       val myMessage = Message( "Hello Kotlin" )
6       send(myMessage, "taras@gmail.com" )
7   }
```

Оскільки на початку файлу імпортовані всі типи з пакета **email**, то з'явилась можливість використання класу **Message** та функції **send** у функції **main**.

Результат роботи програми наведено на рисунку 4.5.

```
Message `Hello Kotlin` has been sent to taras@gmail.com
```

Рисунок 4.5. Результат роботи програми

Також можна імпортувати окремі типи, які визначені у пакеті:

```
1   import email.send
2   import email.Message
```

4.3.1. Псевдоніми

У мові Kotlin за допомогою оператора **as** можна визначати псевдонім для типу, що підключається, а потім звертатися до цього типу через його псевдонім, наприклад:

```
1   import email.send as sendEmail
2   import email.Message as EmailMessage
3
4   fun main() {
5
6       val myMessage = EmailMessage( "Hello Kotlin" )
7       sendEmail(myMessage, "taras@gmail.com" )
8   }
```

В цьому прикладі для функції **send()** визначено псевдонім **sendEmail**. Тож далі для звернення до цієї функції необхідно використовувати її псевдонім, наприклад:

```
1   sendEmail(myMessage, "taras@gmail.com" )
```

Також для класу **Message** визначено псевдонім **EmailMessage**. Відповідно, при використанні цього класу необхідно застосовувати його псевдонім, а не оригінальне ім'я, наприклад:

```
1   val myMessage = EmailMessage( "Hello Kotlin" )
```

Псевдоніми можуть знадобитися, якщо імпортуються з різних пакетів типи з однаковими іменами. Наприклад, нехай у проекті є файл **sms.kt**, а саме:

```
1   package sms
2
3   class Message( val text: String)
4
5   fun send(message: Message, phoneNumber: String){
6       println( "Message `${message.text}` has been sent
    to $phoneNumber" )
7   }
```

В цьому прикладі визначено пакет **sms**, який також містить клас **Message** та функцією **send** для надсилання повідомлення за допомогою SMS.

Припустимо, у файлі **app.kt** необхідно одночасно використовувати клас **Message** та функцію **send** і з файлу **email.kt**, і з файлу **sms.kt**, наприклад:

```
1   import email.send as sendEmail
2   import email.Message as EmailMessage
3   import sms.send as sendSms
4   import sms.Message as SmsMessage
5
6   fun main() {
7
```

```
8       val myEmailMessage      =      EmailMessage( "Hello
    Kotlin" )
9       sendEmail(myEmailMessage, "taras@gmail.com" )
10
11      val mySmsMessage = SmsMessage( "Hello Kotlin" )
12      sendSms(mySmsMessage, "+1234567890" )
13  }
```

4.3.2. Вбудовані пакети

Kotlin має низку вбудованих пакетів [12], які підключаються за замовчуванням у будь-який файл на мові Kotlin, а саме:

- **kotlin.***
- **kotlin.annotation.***
- **kotlin.collections.***
- **kotlin.comparisons.***
- **kotlin.io.***
- **kotlin.ranges.***
- **kotlin.sequences.***
- **kotlin.text.***

Тому якщо виникне потреба використовувати якісь типи, визначені в цих пакетах, то явно ці пакети не потрібно імпортувати.

4.4. Успадковування

Успадковування дозволяє створювати класи, які розширюють функціональність або змінюють поведінку вже існуючих класів. Щодо успадковування виділяються два ключові компоненти. Насамперед це **базовий клас** (клас-батько, батьківський клас, суперклас), який визначає базову функціональність. І **похідний клас** (клас-спадкоємець, підклас), який успадковує функціональність базового класу і може розширювати чи модифікувати її.

Щоб функціональність класу можна було успадкувати, необхідно визначити для цього класу інструкцію **open**. За замовчуванням без цієї інструкції клас не може бути успадкований, наприклад:

```
1   open class базовий_клас
2   class похідний_клас: базовий_клас
```

Для налаштування успадковування, після назви похідного класу вказується символ двокрапки (**:**) і потім вказується клас, від якого здійснюватиметься успадковування, наприклад:

```
1   open class Person{
2       var name: String = "Undefined"
```

```
3       fun printName(){
4           println(name)
5       }
6   }
7   class Employee: Person()
```

В цьому прикладі, клас **Person** описує людину, яка має властивість **name** (ім'я людини) та метод **printName()** для виведення інформації про людину. Клас **Employee** описує умовного працівника. Оскільки працівник є людиною, то клас працівника буде розділяти спільний функціонал із класом людини. Тому замість того, щоб ще раз визначати в класі **Employee** властивість **name**, краще успадкувати весь функціонал класу **Person**. Тобто, в цьому разі, клас **Person** є базовим або суперкласом, а клас **Employee** є похідним класом або класом-спадкоємцем.

Варто враховувати, що при успадковуванні похідний клас повинен викликати первинний конструктор (а якщо такого немає, то конструктор за замовчуванням) базового класу.

В наведеному вище прикладі клас **Person** явно не визначає первинних конструкторів, тому в класі **Employee** необхідно викликати конструктор за замовчуванням для класу **Person**.

Викликати конструктор базового класу у похідному класі можна двома способами. Перший спосіб – це після символу двокрапки (**:**) відразу здійснити виклик конструктора базового класу, наприклад:

```
1   class Employee: Person()
```

В цьому рядку запис **Person()** позначає виклик конструктора за замовчуванням класу **Person**.

Другий спосіб виклику конструктора базового класу – це визначити допоміжний конструктор у похідному класі та викликати в ньому конструктор базового класу за допомогою ключового слова **super**, наприклад:

```
1   open class Person{
2       var name: String = "Undefined"
3       fun printName(){
4           println(name)
5       }
6   }
7   class Employee: Person{
8
9       constructor() : super (){
10
11      }
12  }
```

В наведеному коді за допомогою ключового слова **constructor** у класі **Employee** визначається вторинний конструктор. А після списку його параметрів після символу двокрапки (**:**) записано звернення до конструктора базового класу: **constructor(): super()**. Тобто в цьому випадку виклик **super()** – це виклик конструктора базового класу.

Незалежно від того, який спосіб буде обраний, стає можливим створення об'єктів класу **Employee** і використання для них успадкованого від класу **Person** функціоналу, а саме:

```
1   fun main() {
2
3       val volodymyr: Employee = Employee()
4       volodymyr.name = "Volodymyr"
5       volodymyr.printName()
6   }
7   open class Person{
8       var name: String = "Undefined"
9       fun printName(){
10          println(name)
11      }
12  }
13  class Employee: Person()
```

4.4.1. Успадковування класу з первинним конструктором

Якщо базовий клас явно визначає конструктор (первинний чи вторинний), то похідний клас повинен викликати цей конструктор. Для виклику конструктора базового класу у похідному застосовуються ті ж самі способи, що наведені вище. Розглянемо їх докладніше.

Перший спосіб – це виклик конструктора після назви класу через символ двокрапки (**:**), наприклад:

```
1   open class Person( val name: String){
2       fun printName(){
3           println(name)
4       }
5   }
6   class Employee(empName: String): Person(empName)
```

В наведеному прикладі клас **Person** через конструктор встановлює властивість **name**. Тому в класі **Employee** теж визначений конструктор, який приймає значення типу **String** і передає його в конструктор **Person**.

Якщо похідний клас не має явного первинного конструктора, тоді під час виклику вторинного конструктора повинен викликатися конструктор базового класу через ключове слово **super**, наприклад:

```
1   open class Person( val name: String){
2       fun printName(){
3           println(name)
4       }
5   }
6   class Employee: Person{
7
8       constructor(empName: String) : super (empName){}
9   }
```

Знову ж таки, оскільки конструктор **Person** приймає один параметр, то у **super()** необхідно передати значення для цього параметра.

Наведемо ще один приклад застосування класів:

```
1   fun main() {
2
3       val volodymyr = Employee( "Volodymyr" )
4       volodymyr.printName()
5   }
6
7   open class Person( val name: String){
8       fun printName(){
9           println(name)
10      }
11  }
12  class Employee(empName: String): Person(empName)
```

Вище розглядався випадок, коли в базовому класі визначено первинний конструктор. Та все те діє і в тому випадку, якщо в базовому класі є тільки вторинні конструктори, наприклад:

```
1   fun main() {
2
3       val volodymyr = Employee( "Volodymyr" )
4       volodymyr.printName()
5   }
6
7   open class Person{
8
9       val name: String
10      constructor(userName: String){
11          name=userName
12      }
13      fun printName(){
14          println(name)
15      }
16  }
17  class Employee(empName: String): Person(empName)
```

4.4.2. Розширення базового класу

Похідний клас успадковує функціонал від базового класу, але також може визначати свій власний функціонал, наприклад:

```
1   fun main() {
2
3       val volodymyr =
    Employee( "Volodymyr" , "JetBrains" )
4       volodymyr.printName()
5       volodymyr.printCompany()
6   }
7
8   open class Person( val name: String){
9       fun printName(){
10          println(name)
11      }
12  }
13  class Employee(empName:          String, val company:
    String): Person(empName){
14
15      fun printCompany(){
16          println(company)
17      }
18  }
```

В наведеному прикладі клас **Employee** додає до успадкованого функціонала властивість **company**, яка зберігає назву компанії працівника, та функцію **printCompany()**.

Варто відзначити, що у мові Kotlin можна успадкувати клас тільки від одного класу, **множинне успадковування не підтримується**.

Також, варто зазначити, що всі класи за замовчуванням успадковуються від класу **Any**, навіть якщо клас **Any** явно не вказаний як базовий. Тому будь-який клас вже за замовчуванням матиме всі властивості та функції, які визначені у класі **Any**. Тож всі класи за замовчуванням вже будуть мати такі функції, як **equals**, **toString**, **hashcode**.

4.5. Модифікатори видимості

Всі типи, що використовуються, а також компоненти типів (класи, об'єкти, інтерфейси, конструктори, функції, властивості) мають певний рівень видимості, що визначається модифікатором видимості (модифікатором доступу). Модифікатор видимості визначає, де ті чи інші типи та їх компоненти доступні та де їх можна використовувати. У мові Kotlin є наступні модифікатори видимості:

- **private**: класи, об'єкти, інтерфейси, і навіть функції та властивості, визначені поза класом, з цим модифікатором доступні лише у тому файлі, де вони визначені. Члени класу з цим модифікатором доступні лише у межах свого класу;
- **protected**: члени класу з цим модифікатором доступні у класі, в якому вони визначені, та у класах-спадкоємцях;
- **internal**: класи, об'єкти, інтерфейси, функції, властивості, конструктори з цим модифікатором доступні у будь-якій частині модуля, де вони визначені. При цьому у мові Kotlin **модуль** - це набір файлів, скомпільованих разом в одну структурну одиницю. Це може бути модуль **IntelliJ IDEA** або проєкт **Maven**;
- **public**: класи, функції, властивості, об'єкти, інтерфейси з цим модифікатором доступні у будь-якій частині програми. При цьому якщо функції або класи з цим модифікатором визначені в іншому пакеті, їх все одно потрібно імпортувати.

Для встановлення рівня видимості модифікатор ставиться перед ключовими словами **var/val/fun** на самому початку визначення властивості або функції.

Якщо модифікатор видимості явно не вказаний, то за замовчуванням застосовується модифікатор **public**. Тобто наступний клас:

```
1   class Person(){
2
3       var name = "Undefined"
4       var age = 18
5
6       fun printPerson(){
7           println( "Name: $name Age: $age" )
8       }
9   }
```

буде еквівалентний визначенню класу, наведеному нижче, а саме:

```
1   class Person(){
2
3       public var name = "Undefined"
4       public var age = 18
5
6       public fun printPerson(){
7           println( "Name: $name Age: $age" )
8       }
9   }
```

Якщо властивості оголошуються через первинний конструктор і для них явно не вказано модифікатор видимості, наприклад:

```
1   class Person( val name: String, val age: Int) {
```

```
2       public fun printPerson(){
3           println( "Name: $name Age: $age" )
4       }
5   }
```

то, до таких властивостей також автоматично застосовується модифікатор видимості **public**, а саме:

```
1   class Person(public val name: String, public val age:
    Int){
2       public fun printPerson(){
3           println( "Name: $name Age: $age" )
4       }
5   }
```

Відповідно, можна звертатися до подібних компонентів класу в будь-якому місці програми, наприклад:

```
1   fun main() {
2
3       val taras = Person( "Taras" , 37)
4       taras.printPerson() // Ім'я: Taras Age: 37
5
6       println(taras.name)
7       println(taras.age)
8   }
```

4.5.1. Модифікатор private

Якщо до властивостей та методів застосовується модифікатор **private**, то до них не можна буде звернутися ззовні, тобто поза цим класом, наприклад:

```
1   class Person( private val name:String, _age: Int){
2
3       private val age = _age
4
5       fun printPerson(){
6           printName()
7           printAge()
8       }
9       private fun printName(){
10          println( "Name: $name" )
11      }
12      private fun printAge(){
13          println( "Age: $age" )
14      }
15  }
16
17  fun main() {
```

```
18
19      val taras = Person( "Taras" , 37)
20      taras.printPerson()
21
22      // println(taras.name) // Помилка! - Властивість
    name - private
23      //   taras.printAge() //   Помилка!   -   функція
    printAge - private
24  }
```

У наведеному прикладі клас **Person** визначає дві властивості, **name** (ім’я людини) і **age** (вік людини). Для більшої ілюстрації одна властивість визначається за допомогою конструктора, а інша визначається як змінна класу. І оскільки ці властивості визначаються за допомогою модифікатора **private**, існує можливість отримати до них доступ лише в цьому класі. Можливість звертатися до них за межами класу заборонена, а саме:

```
1   println(taras.name) //   Помилка! -   Властивість
    name - private
```

Також, в прикладі наведеному вище, клас **Person** визначає три функції **printPerson ()**, **printAge ()** та **printName ()**. Останні дві функції виводять на консоль значення властивостей **name** (ім’я людини) та **age** (вік людини). А функція **printPerson** виводить інформацію про об'єкт, викликаючи попередні дві функції.

Однак функції **printAge()** та **printName()** визначені як приватні, тому їх можна використовувати лише всередині класу, наприклад:

```
1   taras.printAge() // Помилка! - функція printAge
    - private
```

4.5.2. Модифікатор protected

Модифікатор **protected** визначає властивості і функції, які поза класом видно тільки в класах-спадкоємцях, наприклад:

```
1   fun main() {
2
3       val taras = Employee( "Taras" , 37)
4       taras.printEmployee() // Ім’я: Taras Age: 37
5
6       // println(taras.name) // Помилка! - Властивість
    name - protected
7       //  taras.printPerson  () //  Помилка!  -  функція
    printPerson - protected
8   }
9   open class Person( protected val name:String, private
```

```
    val age: Int){
10
11      protected fun printPerson(){
12          printName()
13          printAge()
14      }
15      private fun printName(){
16          println( "Name: $name" )
17      }
18      private fun printAge(){
19          println( "Age: $age" )
20      }
21  }
22  class Employee(name:String, age: Int) : Person(name,
    age){
23
24      fun printEmployee(){
25          println( "Employee $name. Full information:" )
26          printPerson()
27          // printName() // та не можна - printName -
    private
28          // println("Age: $age") // так не можна age -
    private
29      }
30  }
```

У цьому прикладі у класі **Person** властивість **name** визначено як **protected**, тому вона доступна в класі-спадкоємці **Employee**, але поза базовим і похідним класом, наприклад у функції **main,** вона не доступна. А ось властивість **age** визначена за допомогою модифікатора **private**, тому вона доступна лише всередині класу **Person**.

Також у класі **Employee** буде доступна функція **printPerson()**, оскільки вона визначена за допомогою модифікатора **protected**, а функції **printAge()** та **printName()** з модифікатором **private** будуть недоступні.

4.5.3. Модифікатори конструкторів

Конструктори, як первинні, так і вторинні, також можуть мати модифікатори. Модифікатор вказується перед ключовим словом **constructor**. За замовчуванням конструктори загальнодоступні. Якщо ж потрібно явно додати модифікатор доступу для основного конструктора, то конструктор визначається за допомогою ключового слова **constructor**, наприклад:

```
1   fun main() {
2
```

```
3       // val volodymyr = Person("Volodymyr") // Так не
    можна - конструктор private
4   }
5   open class Person private constructor( val name:Strin
    g){
6
7        fun printPerson(){
8           println( "Name: $name" )
9       }
10  }
11  // class Employee(name:String) : Person(name) // так
    не можна - конструктор у Person private
```

Варто відзначити, що в наведеному прикладі, через те, що конструктор визначено з модифікатором доступу **private**, то не можна використовувати його поза класом, або створити об'єкт класу у функції **main**, або при успадковуванні. Проте можна використовувати такий конструктор в інших конструкторах всередині класу, наприклад:

```
1   fun main() {
2
3       val taras = Employee( "Taras" , 37)
4       taras.printPerson()
5   }
6   open class Person private constructor( val name:Strin
    g){
7
8       var age: Int = 0
9       protected constructor(_name:String,          _age:
    Int): this (_name){     //     викликаємо     приватний
    конструктор
10          age = _age
11      }
12       fun printPerson(){
13          println( "Name: $name Age: $age" )
14      }
15  }
16  class Employee(name:String, age: Int) : Person(name,
    age)
```

В цьому коді вторинний конструктор класу **Person**, який має модифікатор **protected** (тобто доступний у поточному класі та класах-спадкоємцях) викликає первинний конструктор класу **Person**, який має модифікатор **private**.

4.5.4. Модифікатори об'єктів та типів верхнього рівня

Класи, а також змінні і функції, які визначені поза іншими класами, також можуть мати модифікатори **public**, **private** і **internal**.

Нехай є файл **base.kt**, який визначає однойменний пакет з наступним змістом:

```
1   package base
2
3   private val privateVal = 3
4   val publicVal = 5
5
6   private class PrivateClass( val name: String)
7   class PublicClass( val name:String)
8
9   private fun privateFun(){
10      println( "privateFun" )
11      println(privateVal)
12      val privateClass= PrivateClass( "Taras" )
13  }
14
15  fun publicFun(){
16      println( "publicFun" )
17      println(privateVal)
18      val privateClass= PrivateClass( "Taras" )
19  }
```

Усередині цього файлу можна використовувати його приватні змінні, функції, класи. Проте, при підключенні цього пакета до інших файлів, приватні змінні, функції і класи будуть недоступні, наприклад:

```
1   import base.*
2
3   fun main() {
4
5       publicFun()
6       val publicClass = PublicClass( "Taras" )
7       println(publicVal)
8
9
10      // privateFun() // функція недоступна
11      // val privateClass= PrivateClass("Taras") // клас
    недоступний
12      // println(privateVal) // змінна недоступна
13  }
```

Навіть усередині одного файлу є обмеження на використання приватних класів, а саме:

```
1   package email
2
3   private class Message( val text: String)
4
5   fun send(message: Message, address : String){
        println( "Message `${message.text}` has been sent
6   to $address" )
7   }
```

В цьому прикладі згенерується помилка, тому що публічна (**public)** функція не може приймати параметр приватного (**private)** класу. І в цьому випадку необхідно або зробити клас **Message** публічним **(public)**, або функцію **send** приватною (**private)**.

4.6. Гетери та сетери

Гетери (**getter**) та сетери (**setter**), які ще називають методами доступу, дозволяють керувати доступом до змінної. Їх формальний синтаксис наступний:

```
1   var ім'я_властивості[:       тип_властивості]       [=
    ініціалізатор_властивості]
2       [getter]
3       [setter]
```

Ініціалізатор, гетер та сетер є необов'язковими властивостями. Вказувати тип властивості також необов'язково, якщо він може бути виведений автоматично зі значення ініціалізатора або зі значення гетера, що повертається.

Гетери та сетери необов'язково визначати саме для властивостей усередині класу, вони можуть також застосовуватися до змінних верхнього рівня.

4.6.1. Сетер

Сетер визначає логіку для присвоєння значення змінній. Вона визначається за допомогою слова **set**. Наприклад, є змінна **age**, яка зберігає вік користувача і має числове значення:

```
1   var age: Int = 18
```

Але теоретично можна встановити будь-який вік: 2, 6, -200, 100500. І не всі ці значення будуть коректними. Наприклад, у людини не може бути негативного віку. І для перевірки вхідних значень можна використовувати сетер, наприклад:

```
1   var age: Int = 18
2       set(value){
3           if ((value>0) and (value<110))
4               field = value
5       }
6
```

```
7   fun main() {
8
9       println(age) // 18
10      age = 45
11      println(age) // 45
12      age = -345
13      println(age) // 45
14  }
```

Блок **set** визначається відразу після властивості, до якої він відноситься. В наведеному прикладі блок **set** визначається після властивості **age**. При цьому блок **set** фактично є функцією, яка приймає один параметр – **value**. Через параметр **value** передається встановлюване значення. Наприклад, у виразі **age = 45** число 45 і буде тим об'єктом, який зберігатиметься у властивості **value**.

У блоці **set** перевіряється, чи входить значення в діапазон допустимих значень. Якщо входить, тобто якщо значення коректне, то воно передається об'єкту **field**. Якщо значення некоректне, то властивість просто зберігає своє попереднє значення.

Ідентифікатор **field** позначає автоматично згенероване поле, яке безпосередньо зберігає значення властивості. Тобто властивості фактично є надбудовою над полями, але безпосередньо в класі не можна визначати поля, тут можна працювати тільки з властивостями. Варто зазначити, що до поля через ідентифікатор **field** можна звернутися тільки за допомогою гетера або сетера, і в кожній конкретній властивості можна звертатися тільки до свого поля.

У функції **main** при другому зверненні до сетеру (**age = -345**) можна побачити, що значення властивості **age** не змінилося. Оскільки нове значення 345 не входить у діапазон від 0 до 110.

4.6.2. Гетер

Гетер керує отриманням значення властивості та визначається за допомогою ключового слова **get**, наприклад:

```
1   var age: Int = 18
2       set(value){
3           if ((value>0) and (value<110))
4               field = value
5       }
6       get() = field
```

Праворуч від виразу **get()** через знак дорівнює (=) вказується значення, що повертається. В наведеному прикладі повертається значення поля **field**, яке зберігає значення властивості **name**. Хоча в такому гетері великого сенсу немає, оскільки отримати подібне значення можна і без гетера.

Якщо гетер повинен містити більше інструкцій, то гетер можна оформити в блок з кодом усередині фігурних дужок:

```
1   var age: Int = 18
2       set(value){
3           println( "Call setter" )
4           if ((value>0) and (value<110))
5               field = value
6       }
7       get(){
8           println( "Call getter" )
9           return field
10      }
```

Якщо гетер оформлений в блок коду, то для повернення значення необхідно використовувати оператор **return**. І, таким чином, щоразу, коли буде отримуватись значення змінної **age** (наприклад, у разі виклику **println(age)**), буде спрацьовувати гетер, коли повертатиметься значення. Наприклад:

```
1   fun main() {
2
3       println(age) // спрацьовує get
4       age = 45 // спрацьовує set
5       println(age) // спрацьовує get
6   }
```

Результат роботи програми наведено на рисунку 4.6.

```
Call getter
18
Call setter
Call getter
45
```

Рисунок 4.6. Результат роботи програми

4.6.3. Використання гетерів та сетерів у класах

Хоча гетери та сетери можуть застосовуватись до глобальних змінних, як правило, вони використовуються для опосередкованого доступу до властивостей класу.

Наведемо приклад використання сетеру:

```
1   fun main() {
2
3       val volodymyr: Person = Person( "Volodymyr" )
4       volodymyr.age = 25 // Викликаємо сетер
5
```

```
6       println(volodymyr.age) // 25
7       volodymyr.age = -8 // Викликаємо сетер
8       println(volodymyr.age) // 25
9   }
10  class Person( val name: String) {
11
12      var age: Int = 1
13          set(value){
14              if ((value>0) and (value<110))
15                  field = value
16          }
17  }
```

При другому зверненні до сетеру (**volodymyr.age = -8**) можна побачити, що значення властивості **age** не змінилося. Оскільки нове значення **-8** не входить у діапазон від **0** до **110**.

4.6.4. Обчислюваний гетер

Гетер може повертати обчислювані значення, які можуть задіяти кілька властивостей, наприклад:

```
1   fun main() {
2       val taras = Person( "Taras" , "Shevchenko" )
3       println(taras.fullname) // Taras Shevchenko
4       taras.lastname = "Spivakovskyi"
5       println(taras.fullname) // Taras
    Spivakovskyi
6   }
7   class Person( var firstname:
    String, var lastname: String){
8
9       val fullname: String
10          get() = "$firstname $lastname"
11  }
```

В цьому прикладі властивість **fullname** визначає блок **get**, який повертає повне ім'я користувача, створене на основі його властивостей **firstname** та **lastname**. При цьому значення самої властивості **fullname** безпосередньо змінити не можна – воно визначено як доступне тільки для читання. Однак якщо зміняться значення складових його властивостей – **firstname** і **lastname**, то також зміниться значення, що повертатиметься з **fullname**.

4.6.5. Використання полів для зберігання значень

Вище вже розглядалося, що за допомогою спеціального поля **field** у сетері та гетері можна звертатися до безпосереднього значення властивості, яке зберігається у спеціальному полі. Проте можна самостійно визначити подібне поле. Нерідко це приватне поле.

Можна використовувати і гетер і сетер одночасно, наприклад:

```
1   fun main() {
2
3       val taras = Person( "Taras" )
4       println(taras.age) // 1
5       taras.age = 37
6       println(taras.age) // 37
7       taras.age = 156
8       println(taras.age) // 37
9   }
10  class Person( val name: String) {
11
12      private var _age = 1
13      var age: Int
14          set(value){
15              if ((value > 0) and (value < 110))
16                  _age = value
17          }
18          get()= _age
19  }
```

В цьому прикладі для властивості **age** додані гетер і сетер, які є надбудовою над полем **_age**, яке власне і зберігає значення.

4.7. Перевизначення методів та властивостей

Мова Kotlin дозволяє перевизначати у похідному класі функції та властивості, які визначені у базовому класі. Щоб функції та властивості базового класу можна було перевизначити, до них застосовується інструкція **open**. При перевизначені у похідному класі до цих функцій застосовується інструкція **override**.

4.7.1. Перевизначення властивостей

Щоб вказати, що властивість можна перевизначити у похідному класі, перед визначенням вказується ключове слово **open**, наприклад:

```
1   open class Person( val name: String){
2       open var age: Int = 1
```

```
3   }
```

В наведеному прикладі властивість **age** доступна для перевизначення.

Якщо властивість визначається через первинний конструктор, то також перед визначенням ставиться інструкція **open**, наприклад:

```
1   open class Person( val name:                String,
    open var age: Int = 1) {
2   }
```

У похідному класі для перевизначення властивості перед нею вказується інструкція **override**, наприклад:

```
1   open class Person( val name:                String,
    open var age: Int = 1) {
2   }
3
4   open class Employee(name: String): Person(name){
5
6       override var age: Int = 18
7   }
```

В цьому прикладі перевизначення полягає у зміні початкового значення властивості **age**.

Також перевизначити властивість можна відразу в первинному конструкторі, наприклад:

```
1   open class Person( val name:   String,   open var age:
    Int = 1) {
2   }
3
4   open class Employee(name:    String, override var age:
    Int = 18): Person(name, age){}
```

Розглянемо наступний приклад:

```
1   fun main() {
2
3       val taras = Person( "Taras" )
4       println( "Name:                ${taras.name} Age:
    ${taras.age}" )
5
6       val volodymyr = Employee( "Volodymyr" )
        println( "Name:              ${volodymyr.name} Age:
7   ${volodymyr.age}" )
8   }
9   open class Person( val name:  String,  open var age:
    Int = 1)
10
11  open class Employee(name:  String, override var age:
    Int = 18): Person(name, age)
```

Результат роботи програми наведено на рисунку 4.7.

```
Name: Taras Age: 1
Name: Volodymyr Age: 18
```

Рисунок 4.7. Результат роботи програми

4.7.2. Перевизначення гетерів та сетерів

Також можна перевизначати гетери та сетери властивостей, наприклад:

```
1   open class Person( val name: String){
2
3       open val fullInfo:String
4           get() = "Person $name - $age"
5
6       open var age: Int = 1
7           set(value){
8               if (value > 0 && value < 110)
9                   field = value
10          }
11  }
12  open class Employee(name: String): Person(name){
13
14      override val fullInfo: String
15          get() = "Employee $name - $age"
16
17      override var age: Int = 18
18          set(value){
19              if (value > 17 && value < 110)
20                  field = value
21          }
22  }
23
24  fun main() {
25
26      val taras = Person( "Taras" )
27      taras.age = 14
28      println(taras.fullInfo)
29
30      val volodymyr = Employee( "Volodymyr" )
31      volodymyr.age = 14
32      println(volodymyr.fullInfo)
33  }
```

В цьому прикладі клас **Employee** перевизначає гетер властивості **fullInfo** та сетер властивості **age**.

4.7.3. Перевизначення методів

Щоб функції базового класу можна було перевизначити, до них застосовується інструкція **open**. При перевизначені у похідному класі до цих функцій застосовується інструкція **override**, наприклад:

```
1   open class Person( val name: String){
2       open fun display(){
3           println( "Name: $name" )
4       }
5   }
6   class Employee(name:   String, val company:   String):
    Person(name){
7
8       override fun display() {
9           println( "Name: $name Company: $company" )
        }
10  }
11
12  fun main() {
13
14      val taras = Person( "Taras" )
15      taras.display() // Name: Taras
16
17      val volodymyr                                    =
    Employee( "Volodymyr" , "JetBrains" )
18      volodymyr.display() //  Name:  Volodymyr Company:
19  JetBrains
20  }
```

В цьому прикладі функція **display** визначена у класі **Person** з інструкцією **open**, тож у похідних класах її можна перевизначити. У класі **Employee** ця функція перевизначена із застосуванням інструкції **override**.

4.7.4. Перевизначення в ієрархії успадковування класів

Варто враховувати, що перевизначити функції можна по всій ієрархії успадковування. Наприклад розглянемо клас **Manager**, успадкований від **Employee**:

```
1   open class Person( val name: String){
2       open fun display(){
3           println( "Name: $name" )
4       }
5   }
6   open class Employee(name:         String, val company:
    String): Person(name){
```

```
7
8       override fun display() {
9           println( "Name: $name Company: $company" )
10      }
11  }
12  class Manager(name:          String,          company:
    String):Employee(name, company){
13      override fun display() {
14          println( "Name:        $name        Company:
    $company Position: Manager" )
15      }
16  }
```

У наведеному прикладі клас **Manager** перевизначає функцію **display**, оскільки серед базових класів є клас **Person**, який визначає цю функцію з ключовим словом **open**.

4.7.5. Заборона перевизначення

Іноді буває необхідно заборонити подальше перевизначення функції у класах-спадкоємцях. Для цього застосовується ключове слово **final**, наприклад:

```
1   open class Person( val name: String){
2       open fun display(){
3           println( "Name: $name" )
4       }
5   }
6   open class Employee(name:       String, val company:
    String): Person(name){
7
8       final override fun display() {
9           println( "Name: $name Company: $company" )
10      }
11  }
12  class Manager(name:          String,          company:
    String):Employee(name, company){
13      // тепер функцію не можна перевизначити
14      /*override fun display() {
15          println("Name:        $name        Company:
    $company Position: Manager")
16      }*/
17  }
```

4.7.6. Звернення до реалізації властивостей та функцій базового класу

За допомогою ключового слова **super** у похідному класі можна звертатися до реалізації властивостей та функцій базового класу, наприклад:

```
1   open class Person( val name: String){
2
3       open val fullInfo:String
4           get() = "Name: $name"
5
6       open fun display(){
7           println( "Name: $name" )
8       }
9   }
10  open class Employee(name:    String, val company:
    String): Person(name){
11
12      override val fullInfo: String
13          get()    = "${super.fullInfo}    Company:
    $company"
14
15      final override fun display() {
16          super .display()
17          println( "Company: $company" )
18      }
19  }
```

У наведеному прикладі похідний клас **Employee** при перевизначені властивості та функції застосовує реалізацію з базового класу **Person**. Наприклад, через **super.fullInfo** повертається значення властивості базового класу (тобто значення властивості **name**), а за допомогою виклику **super.display()** викликається реалізація функції **display** з класу **Person**.

4.8. Абстрактні класи та методи

Абстрактні класи – це класи, що визначені за допомогою модифікатора **abstract**. Відмінною рисою абстрактних класів є те, що не можна створити об'єкт подібного класу. Розглянемо приклад визначення абстрактного класу **Human**:

```
1   abstract class Human( val name: String)
```

Абстрактний клас, як і звичайний, може мати властивості, функції, конструктори, але створити об'єкт безпосередньо викликавши його конструктор не можливо:

```
1   val kateryna:  Human //  норм,  просто  визначення
    змінної
2   val olena: Human = Human ( "Olena" ) //! помилка,
    створити об'єкт не можна
```

Такий клас можна лише успадкувати, наприклад:

```
1   abstract class Human( val name: String){
2
```

```
3       fun hello(){
4           println( "My name is $name" )
5       }
6   }
7   class Person(name: String): Human(name)
```

Варто зазначити, що в наведеному прикладі перед абстрактним класом не потрібно вказувати інструкцію **open**, як при успадковуванні неабстрактних класів:

```
1   fun main(args: Array<String>) {
2
3       val kateryna: Person = Person( "Kateryna" )
4       val sofia: Human = Person( "Sofia Sunshine" )
5       kateryna.hello() // My name is Kateryna
6       sofia.hello() // My name is Sofia Sunshine
7   }
```

Абстрактні класи можуть мати абстрактні методи та властивості. Це функції та властивості, які визначаються за допомогою ключового слова **abstract**. Абстрактні методи не містять реалізації, тобто вони не мають тіла. А для абстрактних властивостей значення не вказано. Більше того, абстрактні методи та властивості можна визначити лише в абстрактних класах, наприклад:

```
1   abstract class Human( val name: String){
2
3       abstract var age: Int
4       abstract fun hello()
5   }
6   class Person(name: String): Human(name){
7
8       override var age: Int = 1
9       override fun hello(){
10          println( "My name is $name" )
11      }
12  }
```

Якщо клас успадковується від абстрактного класу, він повинен або реалізувати всі його абстрактні методи та властивості, або також бути абстрактним.

Так, у наведеному прикладі, клас **Person** повинен обов'язково визначити реалізацію для функції **hello()** і властивості **age**. При цьому, як і при перевизначені звичайних методів та властивостей, застосовується інструкція **override**.

Абстрактні властивості також можна реалізувати у первинному конструкторі, наприклад:

```
1   abstract class Human( val name: String){
2
```

```
3       abstract var age: Int
4       abstract fun hello()
5   }
6   class Person(name:      String, override var age:
    Int): Human(name){
7       override fun hello(){
8           println( "My name is $name" )
9       }
10  }
```

Розглянемо призначення та приклади використання абстрактних класів. Класи зазвичай відбивають якісь сутності реального світу. Але деякі з цих сутностей є абстракцією, яка безпосереднього втілення не має. Наприклад, візьмемо систему геометричних фігур. Насправді немає геометричної фігури як такої. Є коло, прямокутник, квадрат, але фігури немає. Однак і коло, і прямокутник мають щось спільне і є фігурами. У цьому випадку можна визначити абстрактний клас фігури і потім від нього успадковувати всі інші класи фігур, наприклад:

```
1   // абстрактний клас фігури
2   abstract class Figure {
3       // абстрактний метод отримання периметра
4       abstract fun perimeter(): Float
5
6       // абстрактний метод отримання площі
7       abstract fun area(): Float
8   }
9   // Похідний клас прямокутника
10  class Rectangle( val width:        Float, val height:
    Float) : Figure()
11  {
12      // Перевизначення отримання периметра
13      override fun perimeter(): Float{
14          return width * 2 + height * 2;
15      }
16      // Перевизначення отримання площі
17      override fun area(): Float{
18          return width * height;
19      }
20  }
```

4.9. Інтерфейси

Інтерфейси позначають угоду, яку має реалізовувати клас. Інтерфейси можуть містити оголошення властивостей та функцій, а також їхню реалізацію за замовчуванням.

Для визначення інтерфейсу використовується ключове слово **interface**, наприклад:

```
1   interface Movable{
2       var speed: Int // оголошення властивості
3       fun move() //    визначення    функції    без
    реалізації
4       fun stop(){ //    визначення    функції    з
    реалізацією за замовчуванням
5           println( "Зупинився" )
6       }
7   }
```

Наприклад, у цьому прикладі інтерфейс **Movable** описує функціонал транспортного засобу. Він містить дві функції та одну властивість. Функція **move()** є абстрактним методом, який не має реалізації. Друга функція **stop()** має реалізацію за замовчуванням.

При визначенні властивостей в інтерфейсі їм не присвоюються значення.

Об'єкт інтерфейсу можна створити безпосередньо, оскільки інтерфейс не підтримує конструктори і просто представляє шаблон, якому клас повинен відповідати.

Визначимо два класи, які застосовують інтерфейс, наприклад:

```
1   class Car : Movable{
2
3       override var speed = 60
4       override fun move(){
5           println( "Машина їде зі швидкістю $speed
    км/год" )
6       }
7   }
8   class Aircraft : Movable{
9
10      override var speed = 600
11      override fun move(){
12          println( "Літак   летить   зі   швидкістю
    $speed км/год" )
13      }
14      override fun stop(){
15          println( "Приземлився" )
16      }
```

```
17  }
```

Для застосування інтерфейсу після імені класу ставиться символ двокрапки(**:**) за яким вказується назва інтерфейсу. При застосуванні інтерфейсу клас повинен реалізувати всі його абстрактні методи та властивості, а також може створити свою власну реалізацію для тих властивостей та методів, які вже мають реалізацію за замовчуванням. При реалізації функцій та властивостей перед ними ставиться ключове слово **override**.

Так, клас **Car** описує автомобіль та застосовує інтерфейс **Movable**. Оскільки інтерфейс містить абстрактний метод **move()**, то клас **Car** обов'язково повинен його реалізувати.

Те саме стосується і властивості **speed**, тобто клас **Car** повинен її визначити. В наведеному прикладі, реалізація властивості полягає у встановленні їй певного початкового значення.

А ось функцію **stop()** клас **Car** може не реалізовувати, тому що вона вже має реалізацію за замовчуванням.

Клас **Aircraft** описує літак і також застосовує інтерфейс **Movable**. При цьому клас **Aircraft** реалізує обидві функції інтерфейсу.

Далі у програмі можна розглядати об'єкти класів **Car** та **Aircraft** як об'єкти **Movable**, наприклад:

```
1   fun main() {
2
3       val m1: Movable = Car()
4       val m2: Movable = Aircraft()
5       //  val  m3:  Movable  =  Movable()  безпосередньо
    об'єкт інтерфейсу створити не можна
6
7       m1.move()
8       m1.stop()
9       m2.move()
10      m2.stop()
11  }
```

Результат роботи програми показано на рисунку 4.8.

```
Машина їде зі швидкістю 60 км/год.
Зупинився
Літак летить зі швидкістю 600 км/год
Літак приземлився
```

Рисунок 4.8. Результат роботи програми

4.9.1. Реалізація властивостей

Розглянемо ще один приклад. Визначимо інтерфейс **Info**, який оголошує кілька властивостей, наприклад:

```
interface Info{
    val model: String
        get() = "Undefined"
    val number: String
}
```

Перша властивість має гетер, а це означає, що вона має реалізацію за замовчуванням. При застосуванні інтерфейсу таку властивість необов'язково реалізувати. Друга властивість **number** є абстрактною, вона не має ні гетера, ні сетера, тобто немає реалізації за замовчуванням, тому класи повинні його реалізувати.

Для реалізації інтерфейсу візьмемо вище визначений клас **Car**, наприклад:

```
class Car( override val model:
String, override var number: String) : Movable, Info{

    override var speed = 60
    override fun move(){
        println( "Машина їде зі швидкістю $speed
км/год" )
    }
}
```

Тепер клас **Car** застосовує два інтерфейси. Клас може застосовувати кілька інтерфейсів, в такому разі вони вказуються через символ коми (**,**), і всі ці інтерфейси клас повинен реалізувати. Клас **Car** реалізує обидві властивості. При цьому, при реалізації властивостей класу необов'язково вказувати гетер або сетер. Можна також реалізувати властивості у первинному конструкторі, як це зроблено у випадку з властивостями **model** та **number**.

Приклад застосування такого класу наведено нижче:

```
fun main() {

    val tesla: Car = Car( "Tesla" , "7777777" )
    println(tesla.model)
    println(tesla.number)

    tesla.move()
    tesla.stop()
}
```

4.9.2. Правила перевизначення

У мові Kotlin можна успадковувати клас і використовувати інтерфейси. При цьому можна одночасно успадковуватися від класу, і застосовувати один або кілька інтерфейсів. Однак розглянемо, що відбудеться, якщо перевизначена функція з базового класу має те ж соме ім'я, що і функція із застосовуваного інтерфейсу, наприклад:

```
1   open class Video {
2       open fun play() { println( "Play video" ) }
3   }
4
5   interface AudioPlayable {
6       fun play() { println( "Play audio" ) }
7   }
8
9   class MediaPlayer() : Video(), AudioPlayable {
10      // Функцію play обов'язково треба перевизначити
11      override fun play() {
12          super <Video>.play() //              Викликаємо
    Video.play()
13          super <AudioPlayable>.play() //      Викликаємо
    AudioPlayable.play()
14      }
15  }
```

В цьому прикладі, клас **Video** та інтерфейс **AudioPlayable** визначають функцію **play**. У цьому випадку клас **MediaPlayer**, який успадковується від **Video** та застосовує інтерфейс **AudioPlayable**, обов'язково має визначити функцію з тим самим ім'ям, тобто **play**. За допомогою конструкції **super<ім'я_типу>.ім'я_функції** можна звернутися до певної реалізації або у базовому класі, або у інтерфейсі.

4.10. Вкладені класи та інтерфейси

У мові Kotlin класи та інтерфейси можуть бути визначені в інших класах та інтерфейсах. Такі класи (вкладені класи або **nested classes**) зазвичай виконують якусь допоміжну роль, а визначення їх усередині класу або інтерфейсу дозволяє розмістити їх якомога ближче до місця, де вони безпосередньо використовуються.

Розглянемо приклад визначення вкладеного класу:

```
1   class Person{
2       class Account( val username:
    String, val password: String){
3
```

```
4           fun showDetails(){
5               println( "UserName:    $username Password:
    $password" )
6           }
7       }
8   }
```

В цьому прикладі клас **Account** є вкладеним, а клас **Person** зовнішнім.

За замовчуванням вкладені класи мають модифікатор видимості **public**, тобто доступні в будь-якій частині програми. Але для звернення до вкладеного класу слід використовувати ім'я зовнішнього класу. Наведемо приклад створення об'єкта вкладеного класу:

```
1  fun main() {
2
3      val userAcc                                          =
   Person.Account( "qwerty" , "123456" );
4      userAcc.showDetails()
5  }
```

Якщо необхідно обмежити область застосування вкладеного класу лише зовнішнім класом, то вкладений клас слід визначити з модифікатором **private**, наприклад:

```
1   class Person(username: String, password: String){
2
3       private val account: Account = Account(username,
    password)
4
5       private class Account( val username:
    String, val password: String)
6
7       fun showAccountDetails(){
8           println( "UserName:
    ${account.username} Password: $account.password" )
9       }
10  }
11  fun main() {
12
13      val taras = Person( "qwerty" , "123456" );
14      taras.showAccountDetails()
15  }
```

У мові Kotlin класи можуть містити вкладені інтерфейси. Крім того, інтерфейси теж можуть містити вкладені класи та інтерфейси, наприклад:

```
1   interface SomeInterface {
2       class NestedClass
3       interface NestedInterface
```

```
4   }
5
6   class SomeClass {
7       class NestedClass
8       interface NestedInterface
9   }
```

4.10.1. Внутрішні (inner) класи

Варто враховувати, що вкладений (**nested**) клас за замовчуванням не має доступу до властивостей та функцій зовнішнього класу. Наприклад, у наступному програмному коді при спробі звернутися до властивості зовнішнього класу згенерується помилка:

```
1   class BankAccount( private var sum: Int){
2
3       fun display(){
4           println( "sum = $sum" )
5       }
6
7       class Transaction{
8           fun pay(s: Int){
9               sum -= s
10              display()
11          }
12      }
13  }
```

В наведеному прикладі визначено клас банківського рахунку **BankAccount**, який визначає властивість **sum**, яка зберігає суму на рахунку та функцію **display()** для виведення інформації про рахунок на консоль.

Крім того, у класі **BankAccount** визначено вкладений клас **Transaction**, який позначає операцію за рахунком. У цьому прикладі клас **Transaction** визначає функцію **pay()** для оплати з рахунку. Однак у ньому не можна звернутися до властивостей та функцій зовнішнього класу **BankAccount**.

Щоб вкладений клас міг отримати доступ до властивостей та функцій зовнішнього класу, необхідно визначити вкладений клас із ключовим словом **inner**. Такий клас називають внутрішнім класом (**inner class**), щоб відрізнятися від звичайних вкладених класів. Розглянемо наступний приклад:

```
1   fun main() {
2
3       val acc = BankAccount(3400);
4       acc.Transaction().pay(2500)
5   }
```

```
6   class BankAccount( private var sum: Int){
7
8       fun display(){
9           println( "sum = $sum" )
10      }
11
12      inner class Transaction{
13          fun pay(s: Int){
14              sum -= s
15              display()
16          }
17      }
18  }
```

В цьому прикладі клас **Transaction** визначений із ключовим словом **inner**, тому має повний доступ до властивостей та функцій зовнішнього класу **BankAccount**. Але тепер, якщо знадобиться використати об'єкт подібного вкладеного класу, то необхідно створити об'єкт зовнішнього класу, а саме:

```
1   val acc = BankAccount(3400);
2       acc.Transaction().pay(2500)
```

4.10.2. Збіг імен

Розглянемо випадок коли властивості та функції внутрішнього класу називаються так само, як властивості та функції зовнішнього класу. В цьому разі внутрішній клас може звернутися до властивостей та функцій зовнішнього через конструкцію **this@назва_класу.ім'я_властивості_або_функції**, наприклад:

```
1   class A{
2       private val n: Int = 1
3       inner class B{
4           private val n: Int = 1
5           fun action(){
6               println(n) // n із класу B
7               println ( this .n) // n із класу B
8               println( this @Bn) // n із класу B
9               println( this @An) // n із класу A
10          }
11      }
12  }
```

Для прикладу, перепишемо вище наведений програмний код з класами **Account** та **Transaction** наступним чином:

```
1   fun main() {
2
3       val acc = BankAccount(3400);
4       acc.Transaction(2400).pay()
```

```
5   }
6   class BankAccount( private var sum: Int){
7
8       fun display(){
9           println( "sum = $sum" )
10      }
11
12      inner class Transaction( private var sum: Int){
            fun pay(){
13              this @BankAccount.sum                    -
14  = this @Transaction.sum
15              display()
16          }
17      }
18  }
```

4.11. Data-класи

Іноді трапляється ситуації, коли класи потрібні лише для зберігання деяких даних. У мові Kotlin такі класи називаються **data-класи**. Вони визначаються з модифікатором **data**, а саме:

```
1   data class Person( val name:      String, val age:
    Int)
```

При компіляції такого класу компілятор автоматично додає до класу функції з певною реалізацією, яка враховує властивості класу, які визначені в первинному конструкторі, а саме:

- **equals()** : порівнює два об'єкти на рівність;
- **hashCode()** : повертає хеш-код об'єкта;
- **toString()** : повертає представлення об'єкта у вигляді рядка;
- **copy()** : копіює дані об'єкта в інший об'єкт.

Наприклад, розглянемо функцію **toString()**, яка повертає представлення об'єкта у вигляді рядка:

```
1   fun main() {
2
3       val olena: Person = Person( "Olena" , 24)
4       println(olena.toString())
5   }
6
7   class Person( val name: String, val age: Int)
```

Результат роботи програми наведено на рисунку 4.9.

```
Person@2a18f23c
```

Рисунок 4.9. Результат роботи програми

За замовчуванням представлення об'єкта у вигляді рядка практично нічого не говорить. Як правило, ця функція призначена для виведення інформації про стан об'єкта, але для цього її треба визначати. Тепер додамо модифікатор **data** до визначення класу, наприклад:

```
1   data class Person( val name:      String, val age:
    Int)
```

І результат роботи цієї програми відрізнятиметься, як наведено на рисунку 4.10.

```
Person(name=Olena, age=24)
```

Рисунок 4.10. Результат роботи програми

Тобто, можемо побачити, які дані зберігаються в об'єкті і які вони мають значення. Те саме стосується всіх інших функцій. Таким чином, у випадку з **data-класами** надається готова реалізація цих функцій. Їх не треба вручну перевизначати. Але цілком можливо, що існуюча реалізація не буде влаштовувати, тоді її можна визначити у власний спосіб, наприклад:

```
1   data class Person( val name:      String, val age:
    Int) {
2       override fun toString(): String {
3           return "Name: $name Age: $age"
4       }
5   }
```

У цьому прикладі для функції **toString()** компілятор не визначатиме реалізацію.

Іншим показовим прикладом є копіювання даних, а саме:

```
1   fun main() {
2
3       val olena: Person = Person( "Olena" , 24)
4       val kateryna = olena.copy(name = "Kateryna" )
5       println(olena.toString()) //   Person(name=Olena,
    age=24)
6       println(kateryna.toString()) //
    Person(name=Kateryna, age=24)
7   }
8
9   data class Person( var name: String, var age: Int)
```

В цьому прикладі компілятор знову ж таки генерує функцію копіювання за замовчуванням, яку можна використовувати надалі. Якщо ж необхідно, аби деякі дані у об'єкта відрізнялися, то їх можна вказати у функції **copy** у вигляді

іменованих аргументів, як у випадку з властивістю **name** у прикладі, наведеному раніше.

Варто зазначити, аби клас визначити як **data-клас**, він повинен відповідати низці умов:

1. Первинний конструктор повинен мати щонайменше один параметр;
2. Усі параметри первинного конструктора повинні оголошуватись з ключовими словами **var** або **var** , тобто позначати властивості;
3. Властивості, які визначаються поза первинним конструктором, не використовуються у функціях **toString**, **equals** та **hashCode**;
4. Клас не повинен визначатися з модифікаторами **open**, **abstract**, **sealed** або **inner**.

Також варто зазначити, що незважаючи на те, що можна визначати властивості в первинному конструкторі через **val**, і через **var**, наприклад:

```
1   data class Person( var name:      String, var age:
    Int)
```

Але взагалі в низці ситуацій рекомендується визначати властивості через **val**, тобто робити їх незмінними, оскільки на їх підставі обчислюється хеш-код, який використовується як ключ об'єкта в такій колекції як **HashMap**.

4.11.1. Декомпозиція data-класів

Мова Kotlin надає можливість для **data-класів** декомпозиції їх на змінні, приклад чого наведено нижче:

```
1   fun main() {
2
3       val olena: Person = Person( "Olena" , 24)
4
5       val (username, userage) = olena
6       println( "Name:   $username Age:   $userage" ) //
    Name: Olena Age: 24
7   }
8
9   data class Person( var name: String, var age: Int)
```

4.12. Перерахування enums

Enums або перерахування є типом даних, який дозволяє визначити набір логічно пов'язаних констант. Для визначення перерахування використовуються ключові слова **enum class**. Наприклад, визначимо перерахування:

```
1   enum class Day{
2       MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
```

```
    SATURDAY, SUNDAY
}
```

Це перерахування **Day** позначає дні тижня. Усередині перерахування визначаються константи. В наведеному прикладі це назви семи днів тижня. Константи визначаються через кому (**,**). Кожна константа фактично представляє об'єкт цього перерахування, наприклад:

```
fun main() {

    val day: Day = Day.FRIDAY
    println(day) // FRIDAY
    println(Day.MONDAY) // MONDAY
}
```

Класи перерахувань, як і звичайні, класи можуть містити конструктор. Крім того, для констант перерахування також може викликатись конструктор для їх ініціалізації, наприклад:

```
enum class Day( val value: Int) {
    MONDAY(1), TUESDAY(2), WEDNESDAY(3),
    THURSDAY(4),        FRIDAY(5),        SATURDAY(6),
SUNDAY(100500)
}

fun main() {

    val day: Day = Day.FRIDAY
    println(day.value) // 5
    println(Day.MONDAY.value) // 1
}
```

У цьому прикладі у класі перерахування через конструктор визначається властивість **value**. Відповідно, під час визначення констант перерахування необхідно кожну з цих констант ініціалізувати, передавши значення для властивості **value**.

При цьому перерахування – це не просто перелік значень. Вони можуть визначати також властивості та функції. Але якщо клас перерахування містить властивості або функції, то константи мають бути відокремлені крапкою з комою (**;**), наприклад:

```
enum class Day( val value: Int) {
    MONDAY(1), TUESDAY(2), WEDNESDAY(3),
    THURSDAY(4), FRIDAY(5), SATURDAY(6),
    SUNDAY(7);
    fun getDuration(day: Day): Int{
        return value - day.value;
    }
```

```
8   }
9
10  fun main() {
11
12      val day1: Day = Day.FRIDAY
13      val day2: Day = Day.MONDAY
14      println(day1.getDuration(day2)) // 4
15  }
```

В наведеному прикладі в перерахуванні визначено функцію **getDuration()**, яка обчислює кількість діб між двома днями тижня.

4.12.1. Вбудовані властивості та допоміжні методи

Всі перерахування мають дві вбудовані властивості:

- **name** : повертає назву константи у вигляді рядка;
- **ordinal** : повертає порядковий номер константи.

Приклад використання вбудованих властивостей **name** та **ordinal** наведено нижче:

```
1   enum class Day( val value: Int) {
2       MONDAY(1), TUESDAY(2), WEDNESDAY(3),
3       THURSDAY(4), FRIDAY(5), SATURDAY(6),
4       SUNDAY(7)
5   }
6
7   fun main() {
8
9       val day1: Day = Day.FRIDAY
10      println(day1.name) // FRIDAY
11      println(day1.ordinal) // 4
12  }
```

Крім того, у мові Kotlin доступні допоміжні функції, а саме:

- **valueOf(value: String)** : повертає об'єкт перерахування за назвою константи;
- **values()** : повертає масив констант поточного перерахування.

Приклад використання допоміжних функцій **valueOf(value: String)** та **values()** наведено нижче:

```
1   fun main() {
2
3       for (day in Day.values())
4           println(day)
5
6       println(Day.valueOf( "FRIDAY" ))
7   }
```

4.12.2. Анонімні класи та реалізація інтерфейсів

Константи перерахування можуть визначати анонімні класи, які можуть мати власні методи та властивості або реалізувати абстрактні методи класу перерахування, наприклад:

```
1   enum class DayTime{
2       DAY{
3           override val startHour = 6
4           override val endHour = 21
5           override fun printName(){
6               println( "День" )
7           }
8       },
9       NIGHT {
10          override val startHour = 22
11          override val endHour = 5
12          override fun printName(){
13              println( "Ніч" )
14          }
15      };
16      abstract fun printName()
17      abstract val startHour: Int
18      abstract val endHour: Int
19  }
20
21  fun main() {
22
23      DayTime.DAY.printName() // День
24      DayTime.NIGHT.printName() // Ніч
25
26      println( "Day from ${DayTime.DAY.startHour} to
    ${DayTime.DAY.endHour}" )
27
28  }
```

У цьому прикладі клас перерахування **DayTime** визначає абстрактний метод **printName()** і дві змінні – **startHour** (початкова година) та **endHour** (кінцева година). Самі ж константи визначають анонімні класи, що реалізують ці властивості та функцію.

Також класи перерахувань можуть застосовувати інтерфейси. Для цього для кожної константи визначається анонімний клас, який містить усі реалізовані властивості та функції, наприклад:

```
1   interface Printable{
2       fun printName()
3   }
```

```
4   enum class DayTime: Printable{
5       DAY{
6           override fun printName(){
7               println( "День" )
8           }
9       },
10      NIGHT {
11          override fun printName(){
12              println( "Ніч" )
13          }
14      }
15  }
16
17  fun main() {
18
19      DayTime.DAY.printName() // День
20      DayTime.NIGHT.printName() // Ніч
21  }
```

4.12.3. Зберігання стану

Часто перерахування використовуються для зберігання стану у програмі. Залежно від цього стану можна направляти функціонування програми певним шляхом. Наприклад, визначимо перерахування, яке позначає арифметичні операції, та функцію, яка залежно від переданої операції виконує ту чи іншу дію:

```
1   fun main() {
2
3       println(operate(5,       6,       Operation.ADD)) //
    11
4       println(operate(5, 6, Operation.SUBTRACT)) // -1
5       println(operate(5, 6, Operation.MULTIPLY)) // 30
6   }
7   enum class Operation{
8
9       ADD, SUBTRACT, MULTIPLY
10  }
11  fun operate(n1: Int, n2: Int, op: Operation): Int {
12
13      when (op) {
14          Operation.ADD -> return n1 + n2
15          Operation.SUBTRACT -> return n1 - n2
16          Operation.MULTIPLY -> return n1 *n2
17      }
18  }
```

В цьому прикладі функція **operate()** приймає два числа, а саме операнди операції та тип операції як перерахування **Operation**. І залежно від значення перерахування повертає або суму, або різницю, або добуток двох чисел.

4.13. Делегування

Делегування є патерном (шаблоном) об'єктно-орієнтованого програмування, який дозволяє одному об'єкту делегувати/перенаправити всі запити іншому об'єкту. Певною мірою делегування може бути альтернативою успадковування. І перевагою мови Kotlin в цьому випадку є те, що мова Kotlin нативно підтримує цей патерн, надаючи необхідний інструментарій.

Наведемо формальний синтаксис:

```
1   interface Base {
2       fun someFun()
3   }
4
5   class BaseImpl() : Base {
6       override fun someFun() { }
7   }
8
9   class Derived(someBase: Base) : Base by someBase
```

В цьому програмному коді є певний інтерфейс – **Base**, який визначає певний функціонал. І є його реалізація у вигляді класу **BaseImpl**.

Ще один клас **Derived** також застосовує інтерфейс **Base**. При чому, після того, як вказується використовуваний інтерфейс, ставиться ключове слово **by**, а після нього ідентифікатор об'єкта, якому будуть делеговані виклики.

```
1   class Derived(someBase: Base) : Base by someBase
```

Тобто в такій схемі клас **Derived** делегуватиме виклики об'єкту **someBase**, який представляє інтерфейс **Base** і передається через первинний конструктор. При цьому **Derived** може не реалізувати інтерфейс **Base** або реалізувати не повністю, наприклад тільки якісь окремі властивості та функції.

Для прикладу розглянемо наступні класи:

```
1   interface Messenger{
2       fun send(message: String)
3   }
4   class InstantMessenger( val programName:   String)   :
    Messenger{
5
6       override fun send(message: String){
7           println( "Message `$message` has been sent" )
8       }
9   }
```

```
10 class SmartPhone( val name:  String,  m:  Messenger):
   Messenger by m
```

В цьому прикладі визначено інтерфейс **Messenger**, який умовно позначає програму для надсилання повідомлень. Для умовного надсилання повідомлень визначено функцію **send()**.

Також є клас **InstantMessenger** – програма миттєвих повідомлень або месенджер, який застосовує інтерфейс **Messenger**, реалізуючи його функцію **send()**.

Далі визначено клас **SmartPhone**, який описує смартфон і також застосовує інтерфейс **Messenger**, але не реалізує його. Натомість він приймає через первинний конструктор об'єкт **Messenger** і делегує йому звернення до функції **send()**.

Застосуємо ці класи наступним чином:

```
1   fun main() {
2       val telegram = InstantMessenger( "Telegram" )
3       val pixel = SmartPhone( "Pixel 5" , telegram)
4       pixel.send( "Hello Kotlin" )
5       pixel.send( "Learn Kotlin on Sunshine.com" )
6   }
```

В цьому прикладі створено об'єкт **pixel**, який є класом **SmartPhone**. Оскільки **SmartPhone** застосовує інтерфейс **Messenger**, то можна викликати у об'єкта **pixel** функцію **send()** для надсилання умовного повідомлення. Проте сам клас **SmartPhone** не реалізує функцію **send()**, натомість саме виконання цієї функції делегується об'єкту **telegram**, який насправді виконує відправлення повідомлення. Відповідно під час виконання програми отримаємо наступний результат роботи програми, як показано на рисунку 4.11.

```
Message `Hello Kotlin` has been sent
Message `Learn Kotlin on Sunshine.com` has been sent
```

Рисунок 4.11. Результат роботи програми

4.13.1. Множинне делегування

Подібним чином, один об'єкт може делегувати виконання різних функцій різним об'єктам, наприклад:

```
1   fun main() {
2       val telegram = InstantMessenger( "Telegram" )
3       val photoCamera = PhotoCamera()
4       val pixel  =  SmartPhone( "Pixel  5" ,  telegram,
   photoCamera)
5       pixel.send( "Hello Kotlin" )
```

```
6       pixel.takePhoto()
7   }
8
9   interface Messenger{
10      fun send(message: String)
11  }
12  class InstantMessenger( val programName:   String)    :
    Messenger{
13      override fun send(message:         String)         =
    println( "Send message: `$message`" )
14  }
15  interface PhotoDevice{
16      fun takePhoto()
17  }
18  class PhotoCamera: PhotoDevice{
19      override fun takePhoto()    =    println( "Take    a
    photo" )
20  }
21  class SmartPhone( val name:  String,  m:  Messenger,  p:
    PhotoDevice)
22      : Messenger by m, PhotoDevice by p
```

В цьому прикладі клас **SmartPhone** також реалізує інтерфейс **PhotoDevice**, який надає функцію **takePhoto()** для зйомки фото. Але виконання цієї функції він делегує параметру **p**, який представляє інтерфейс **PhotoDevice** і в ролі якого виступає об'єкт **PhotoCamera**.

4.13.2. Перевизначення функцій

Клас може перевизначати частину функцій інтерфейсу, у цьому разі виконання цих функцій не делегується, наприклад:

```
1   fun main() {
2       val telegram = InstantMessenger( "Telegram" )
3       val pixel = SmartPhone( "Pixel 5" , telegram)
4       pixel.sendTextMessage()
5       pixel.sendVideoMessage()
6   }
7
8   interface Messenger{
9       fun sendTextMessage()
10      fun sendVideoMessage()
11  }
12  class InstantMessenger( val programName:   String)    :
    Messenger{
13      override fun sendTextMessage()   =   println( "Send
```

```
    text message" )
14      override fun sendVideoMessage()  =  println( "Send
    video message" )
15  }
16  class SmartPhone( val name:  String,  m:  Messenger)  :
    Messenger by m{
17      override fun sendTextMessage()  =  println( "Send
    sms" )
18  }
```

В цьому прикладі клас **SmartPhone** реалізує функцію **sendTextMessage()**, тому її виконання не делегується. Результат роботи програми наведено на рисунку 4.12.

```
Send sms
Send video message
```

Рисунок 4.12. Результат роботи програми

4.13.3. Делегування властивостей

За аналогією з функціями, об'єкт може делегувати звернення до властивостей, наприклад:

```
1   fun main() {
2       val telegram = InstantMessenger( "Telegram" )
3       val pixel = SmartPhone( "Pixel 5" , telegram)
4       println(pixel.programName) // Telegram
5   }
6   interface Messenger{
7       val programName: String
8   }
9   class InstantMessenger( override val programName:
    String) : Messenger
10  class SmartPhone( val name:  String,  m:  Messenger)  :
    Messenger by m
```

В цьому прикладі у інтерфейсі **Messenger** визначена властивість **programName** – назва програми відправки повідомлень. Клас **SmartPhone** не реалізує цю властивість, тому звернення до цієї властивості делегується об'єкту **m**.

Якби клас **SmartPhone** сам би реалізував властивість **programName**, то делегування не відбувалось би, як наведеного в наступному прикладі:

```
1   fun main() {
2       val telegram = InstantMessenger( "Telegram" )
3       val pixel = SmartPhone( "Pixel 5" , telegram)
4       println(pixel.programName) // Default Messenger
```

```
5   }
6   interface Messenger{
7       val programName: String
8   }
9   class InstantMessenger( override val programName:
    String) : Messenger
10  class SmartPhone( val name:  String,  m:  Messenger)  :
    Messenger by m{
11      override val programName = "Default Messenger"
12  }
```

4.14. Анонімні класи та об'єкти

Іноді виникає потреба створити об'єкт де-якого класу, який більше ніде у програмі не використовується. Тобто клас необхідний лише для створення лише одного об'єкта. У цьому випадку, звичайно, можна, як і зазвичай, визначити клас і створити об'єкт цього класу. Але мова Kotlin для таких ситуацій надає можливість визначити об'єкт анонімного класу.

Анонімні класи не використовують ключове слово **class** для визначення класу. Вони не мають імені, але як і звичайні класи можуть наслідувати інші класи або використовувати інтерфейси. Об'єкти анонімних класів називають анонімними об'єктами.

Для визначення анонімного об'єкта застосовується ключове слово **object**, наприклад:

```
1   fun main() {
2
3       val taras = object {
4           val name = "Taras"
5           var age = 37
6           fun sayHello(){
7               println( "Hi, my name is $name" )
8           }
9       }
10      println( "Name: ${taras.name} Age:
    ${taras.age}" )
11      taras.sayHello()
12  }
```

В цьому прикладі після ключового слова **object** йде блок коду у фігурних дужках, в яких розташовується визначення об'єкта. Як і у звичайному класі, анонімний об'єкт може містити властивості та функції. І далі використовуючи ім'я змінної стає можливим звертання до властивостей та функцій цього об'єкта.

4.14.1. Успадковування анонімними об'єктами

При успадковуванні анонімними об'єктами після слова **object** через двокрапку (**:**) вказується ім'я успадкованого класу або його первинний конструктор, наприклад:

```
1   fun main() {
2
3       val taras = object : Person( "Taras" ) {
4
5           val company = "JetBrains"
6           override fun sayHello(){
7               println( "Hi, my name is $name. I work in
    $company" )
8           }
9       }
10
11      taras.sayHello() // Hi, my name is Taras. I work
    in JetBrains
12  }
13  open class Person( val name: String){
14      open fun sayHello(){
15          println( "Hi, my name is $name" )
16      }
17  }
```

В цьому прикладі клас анонімного об'єкта успадковує клас **Person** і перевизначає його функцію **sayHello()**.

4.14.2. Анонімний об'єкт як аргумент функції

Анонімний об'єкт може передаватися як аргумент під час виклик функції, наприклад:

```
1   fun main() {
2       hello(
3           object : Person( "Stepan" ){
4               val company = "JetBrains"
5               override fun sayHello(){
                    println( "Hi, my name is $name. I
6   work in $company" )
7               }
8       })
9   }
10  fun hello(person: Person){
11      person.sayHello()
12  }
13  open class Person( val name: String){
```

```
14      open fun sayHello()  =  println( "Hi,  my  name  is
    $name" )
15  }
```

Оскільки клас анонімного об'єкта успадковується від класу **Person**, то можна передавати цей анонімний об'єкт параметру функції, який має тип **Person**.

4.14.3. Анонімний об'єкт як результат функції

У мові Kotlin функція може повертати анонімний об'єкт, як результат своєї роботи, наприклад:

```
1   fun main() {
2       val taras = createPerson( "Taras" , "JetBrains" )
3       taras.sayHello()
4   }
5   private fun createPerson(_name:    String,    _company:
    String) = object {
6       val name = _name
7       val company = _company
8       fun sayHello() = println( "Hi, my name is $name.
    I work in $company" )
9   }
```

Однак тут є певні нюанси. Для того, щоб можна було звертатися до властивостей та функцій анонімного об'єкта, функція, яка повертає цей об'єкт, має бути приватною, як у прикладі вище.

Якщо функція має модифікатор **public** або **private inline**, то в цьому випадку властивості та функції анонімного класу (за винятком успадкованих) недоступні, як видно з наступного прикладу:

```
1   fun main() {
2       val taras = createPerson( "Taras" , "JetBrains" )
3       println(taras.name) //  Норм  -  властивість  name
    успадковано від Person
4       println(taras.company) //! Помилка  -  властивість
    недоступна
5   }
6   private inline fun createPerson(_name:  String,  _comp:
    String) = object : Person(_name){
7       val company = _comp
8   }
9
10  open class Person( val name: String)
```

У цьому прикладі функція **createPerson()** має модифікатор **private inline**, тому анонімний об'єкт буде доступний лише успадкованим від класу **Person** властивостям і функціям, але власні властивості та функції будуть не доступні.

Резюме

В наведеному розділі розглянуто принципи і можливості об'єктно-орієнтованого програмування в Kotlin. Описано класи та об'єкти, їх створення, ініціалізацію за допомогою конструкторів (первинних і вторинних) та використання властивостей. Також пояснено, як працюють пакети та імпорт, модифікатори доступу (**private**, **protected**, **internal**, **public**) і механізм успадкування. Окрему увагу приділено гетерам і сетерам, що дозволяють контролювати доступ до змінних, а також перевизначенню методів і властивостей. Наприкінці розглядаються абстрактні класи та їх використання, що є передумовою для вивчення матеріалу, викладеного в наступних розділах, а також без чого не можливо створення мобільних, десктопних та веб-застосунків мовою Kotlin.

Контрольні запитання і завдання

1. Що таке клас і об'єкт у Kotlin?
2. Як створити клас у Kotlin? Наведіть приклад.
3. Чим відрізняється клас від об'єкта?
4. Як створити об'єкт класу у Kotlin?
5. Що таке конструктор у Kotlin? Які типи конструкторів існують?
6. Як визначити первинний конструктор у класі?
7. Як працює блок ініціалізації (**init**) у Kotlin?
8. Як визначити вторинний конструктор і для чого він потрібен?
9. Що таке властивості класу і як їх оголошувати?
10. Чим відрізняються властивості, оголошені через **val** і **var**?
11. Що таке гетери та сетери? Як їх використовувати?
12. Як реалізувати обчислюваний гетер у Kotlin?
13. Що таке успадкування в Kotlin і як його реалізувати?
14. Як успадкувати клас із первинним конструктором?
15. Які модифікатори доступу є в Kotlin? Що вони означають?
16. Як у Kotlin працюють пакети та імпорт?
17. Що таке абстрактний клас? Чим він відрізняється від звичайного?
18. Як перевизначити метод або властивість у Kotlin?
19. Як заборонити перевизначення методу або властивості?
20. Що таке **super** і для чого воно використовується?
21. Створіть клас **Car** з властивостями **brand** (марка авто) і **year** (рік випуску). Створіть об'єкт цього класу у **main()** і виведіть його дані в консоль.

22. Доповніть клас **Car**, додавши первинний конструктор, що приймає **brand** і **year**. Створіть кілька об'єктів із різними параметрами.
23. Додайте до класу **Car** вторинний конструктор, який приймає **brand** і встановлює **year** за замовчуванням у 2024.
24. Створіть клас **Person** із властивістю **age**. Додайте сетер, який не дозволяє встановлювати вік менше 0 або більше 120.
25. Створіть клас **Employee**, який успадковується від класу **Person**. Додайте властивість **company**. Створіть об'єкт класу **Employee** і виведіть його дані.
26. У класі **Person** створіть метод **introduce()**, який виводить ім'я людини. Перевизначте цей метод у класі **Employee**, щоб він виводив ще й компанію.
27. Модифікуйте клас **Employee**, щоб у перевизначеному методі **introduce()** спочатку викликався метод **introduce()** з базового класу **Person**, а потім додавалася інформація про компанію.
28. Створіть абстрактний клас **Animal** з методом **makeSound()**. Від нього успадкуйте класи **Dog** та **Cat**, реалізуйте метод так, щоб собака видавав звук «Woof», а кіт – «Meow».
29. Створіть клас **BankAccount** із приватною змінною **balance**. Додайте методи для поповнення рахунку та зняття грошей (тільки якщо сума зняття не перевищує баланс).
30. Створіть два пакети: **shapes** і **main**. У **shapes** додайте клас **Rectangle**, що містить довжину та ширину. У **main** створіть об'єкт **Rectangle** і виведіть його площу.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Що є правильним способом оголошення класу у Kotlin?

a) class Person {}

b) new class Person {}

c) Person class {}

d) create class Person {}

Правильна відповідь: a).

Який модифікатор доступу потрібно використати, щоб змінна була доступною тільки в межах свого класу?

a) private

b) protected
c) internal
d) public
Правильна відповідь: a).

Закриті запитання з кількома правильними відповідями

Які з наступних тверджень про val та var у Kotlin є правильними?
a) val оголошує змінну, значення якої не можна змінювати
b) var оголошує змінну, значення якої можна змінювати
c) val та var можна використовувати тільки у функціях
d) var працює тільки з цілими числами
Правильна відповідь: a), b).

Завдання на відповідність

Розташуйте етапи створення об'єкта у правильному порядку:
a) Оголошення класу → 1. Етап 1.
b) Створення змінної для збереження об'єкта → 2. Етап 2.
c) Виклик конструктора класу → 3. Етап 3.
Правильна відповідь:
a) → 1.
b) → 2.
c) → 3.

Співвіднесіть термін та його опис:
a) Клас → 1. Конкретний екземпляр класу
b) Об'єкт → 2. Шаблон, що описує структуру даних
c) Конструктор → 3. Метод для ініціалізації об'єкта
Правильна відповідь:
a) → 1.
b) → 2.
c) → 3.

Завдання з кодом

Що не так у наступному коді?
```
class Person {
    var name: String
}
```

```
fun main() {
    val p = Person()
    println(p.name)
}
```

a) У класі Person потрібно визначити конструктор

b) Змінну name потрібно ініціалізувати

c) У main() об'єкт p створено неправильно

d) Код коректний

Правильна відповідь: b).

Що виведе наступний код?

```
open class Person(val name: String) {
    open fun introduce() = "Hi, I'm $name"
}
class Student(name: String) : Person(name) {
    override fun introduce() = "I'm a student named $name"
}
fun main() {
    val person: Person = Student("Alice")
    println(person.introduce())
}
```

a) Hi, I'm Alice

b) I'm a student named Alice

c) Виникне помилка компіляції

d) null

Правильна відповідь: a), c).

Яке значення буде виведене?

```
class Counter {
    var count = 0
        set(value) {
            if (value >= 0) field = value
        }
}
fun main() {
    val c = Counter()
    c.count = -5
    println(c.count)
```

```
}
```

a) -5

b) 0

c) null

d) Виникне помилка

Правильна відповідь: b).

Що буде виведено в консоль?

```
open class Animal {
    open fun makeSound() = "Some sound"
}
class Dog : Animal() {
    fun bark() = "Woof"
}
fun main() {
    val a: Animal = Dog()
    println(a.makeSound())
}
```

a) Some sound

b) Woof

c) Some sound Woof

d) Виникне помилка

Правильна відповідь: a).

Відкриті запитання

Який рядок у коді містить помилку? Поясніть чому.

```
class Car(val brand: String) {
    private val year: Int
    fun getYear() = year
}
```

Правильна відповідь: Помилка в рядку «private val year: Int». Змінна year оголошена, але не ініціалізована.

5. УЗАГАЛЬНЕННЯ

5.1. Узагальнені класи та функції

Generics або узагальнення є прийомом, за допомогою якого методи та класи можуть використовувати об'єкти, типи яких на момент визначення класів та функцій невідомі. Узагальнення дозволяють визначати шаблони, у які можна підставляти різні типи.

5.1.1. Узагальнені типи

Узагальнені типи (**generic types**) є типами, у яких типи об'єктів параметризовані. Що це означає зрозуміємо далі.

Розглянемо наступний клас:

```
1   class Person<T>( val id: T, val name: String)
```

Клас **Person** використовує параметр **T**. Параметри вказуються після імені класу у кутових дужках (< >). Цей параметр **T** позначатиме певний тип даних, який на момент визначення класу невідомий.

Далі у первинному конструкторі визначається властивість **id**, яка описує деякий ідентифікатор. Вона матиме тип, який передається через параметр **T**. На момент визначення класу **Person** ще не відомо, який саме це буде тип.

Сама назва параметра довільна але вона не повинна співпадати з ключовими словами. Часто використовується **T**, як скорочення від слова **type**.

При використанні типу **Person** необхідно його типізувати певним типом, тобто вказати, який тип передаватиметься через параметр **T**, наприклад:

```
1   fun main() {
2
3       val taras: Person<Int> = Person(367, "Taras" )
4       val volodymyr:          Person<String>          =
    Person( "A65" , "Volodymyr" )
5
6       println( "${taras.id} - ${taras.name}" )
7       println( "${volodymyr.id} - ${volodymyr.name}" )
8   }
9
10  class Person<T>( val id: T, val name: String)
```

Для типізації об'єкта після назви типу у кутових дужках (< >) вказується конкретний тип, наприклад:

```
1   val taras: Person<Int>
```

У наведеному прикладі вказано, що параметр **T** фактично представлятиме тип **Int**. Тому у конструктор об'єкта **Person** для властивості **id** необхідно передати числове значення **Int**, наприклад:

```
Person(367, "Taras" )
```

Другий об'єкт типізується типом **String**, тому в конструкторі для властивості **id** передається рядок, наприклад:

```
val volodymyr:          Person<String>          =
Person( "A65" , "Volodymyr" )
```

Якщо конструктор використовує параметр **T**, то можна не вказувати, яким типом типізується об'єкт, адже конкретний тип буде виводитися з типу параметра конструктора автоматично, наприклад:

```
val taras = Person(367, "Taras" )
val volodymyr = Person( "A65" , "Volodymyr" )
```

При цьому параметри типу можуть застосовуватися всередині класу, тобто не тільки при визначенні властивостей, але і у функціях, наприклад:

```
fun main() {

    val taras = Person( "qwrtf2" , "Taras" )
    taras.checkId( "qwrtf2" ) // Однакові
    taras.checkId( "q34tt" ) // Різні
}

class Person<T>( val id: T, val name: String){

    fun checkId(_id: T){
        if (id == _id) {
            println( "Однакові" )
        }
        else {
            println( "Різні" )
        }
    }
}
```

В цьому прикладі клас **Person** визначає функцію **checkId()**, яка перевіряє, чи дорівнює **id** значенню параметра **_id**. При цьому параметр **_id** має тип **T**, тобто він представлятиме той же тип, що і властивість **id**.

Варто зазначити, що **generic-типи** широко використовуються в мові Kotlin. Найпоказовіший приклад, представлений класом **Array<T>**. Параметр класу тут визначає, елементи якого типу масив зберігатиме, наприклад:

```
val people:             Array<String>           =
arrayOf( "Taras" , "Volodymyr" , "Stepan" )
val numbers: Array<Int> = arrayOf(1, 2, 3, 4)
```

5.1.2. Застосування кількох параметрів

У мові Kotlin можна одночасно використовувати кілька параметрів, наприклад:

```
1   fun main() {
2
3       var word1:       Word<String,       String>       =
    Word( "one" , "один" )
4       var word2: Word<String, Int> = Word( "two" , 2)
5
6       println( "${word1.source}  -  ${word1.target}" ) //
    one - один
7       println( "${word2.source}  -  ${word2.target}" ) //
    two - 2
8   }
9
10  class Word<K, V>( val source: K, var target: V)
```

В наведеному прикладі клас **Word** застосовує два параметри **K** та **V**. При створенні об'єкта **Word** ці параметри можуть представляти однаковий тип, а можуть представляти і різні типи.

5.1.3. Узагальнені функції

У мові Kotlin функції, як і класи, можуть бути узагальненими, наприклад:

```
1   fun main() {
2
3       display( "Hello Kotlin" )
4       display(1234)
5       display( true )
6   }
7   fun <T> display(obj: T){
8       println(obj)
9   }
```

Функція **display()** в цьому прикладі параметризована параметром **T**. Параметр **T**, як і раніше, вказується у кутових дужках після слова **fun** та перед назвою самої функції. Функція приймає один параметр типу **T** та виводить його значення на консоль. При використанні цієї функції можна передавати у неї дані будь-якого типу.

Розглянемо інший більш практичний приклад, а саме визначимо функцію, яка повертатиме найбільший масив:

```
1   fun main() {
2
3       val arr1 = getBiggest(arrayOf(1,2,3,4), arrayOf(3,
```

```
    4, 5, 6, 7, 7))
4       arr1.forEach { item -> print( "$item " ) } //
    3 4 5 6 7 7
5
6       println()
7
8       val arr2                                          =
    getBiggest(arrayOf( "Taras" , "Stepan" , "Volodymyr" ),
    arrayOf( "Kateryna" , "Olena" ))
9       arr2.forEach { item -> print("$item ") } //
    Taras Stepan Volodymyr
10  }
11
12  fun <T> getBiggest(args1: Array<T>, args2: Array<T>):
    Array<T>{
13      if(args1.size > args2.size) return args1
14      else return args2
15  }
```

В цьому прикладі функція **getBiggest()** як параметри приймає два масиви. При цьому наперед точно не відомо, об'єкти якого типу ці масиви будуть містити. Однак обидва масиви типізовані параметром **T**, що гарантує, що обидва масиви зберігатимуть об'єкти одного і того ж типу. Усередині функції порівнюється розмір масивів за допомогою їхньої властивості **size** і повертається найбільший масив.

5.2. Обмеження узагальнень

Обмеження узагальнень **generic constraints** обмежують набір типів, які можуть бути передані замість параметра в узагальненнях.

Наприклад, необхідно визначити універсальну функцію для порівняння двох об'єктів та повернення з функції найбільшого об'єкта. На перший погляд здається, що можна просто визначити узагальнену функцію, наприклад:

```
1   fun <T> getBiggest(a: T, b: T): T {
2       if (a > b) return a // ! Помилка
3       else return b
4   }
```

Але компілятор не скомпілює цю функцію, тому що замість параметра типу **T** можуть передаватися різні типи, у тому числі такі, які не підтримують операцію порівняння.

Однак усі типи, які за замовчуванням підтримують цю операцію порівняння, використовують інтерфейс **Comparable**. Тобто необхідно, щоб ці два параметри представляли один і той самий тим, який реалізує тип **Comparable**. І в цьому

випадку можна визначити одну узагальнену функцію, яка буде обмежена типом **Comparable**, наприклад:

```
1   fun main() {
2
3       val result1 = getBiggest(1, 2)
4       println(result1)
5       val result2 = getBiggest( "Taras" , "Stepan" )
6       println(result2)
7
8   }
9   fun <T: Comparable<T>> getBiggest(a: T, b: T): T{
10      return if(a > b) a
11      else b
12  }
```

Обмеження вказується після назви параметра через двокрапку **<T: Comparable<T>>**, тобто в цьому прикладі тип **T** обмежений типом **Comparable<T>**, або інакше кажучи повинен представляти тип **Comparable<T>**. Причому тип **Comparable** є узагальненим.

Варто зазначити, що за замовчуванням до всіх параметрів типу також застосовується обмеження типу **Any?**. Тобто визначення параметра типу **<T>** фактично аналогічне до визначення **<T: Any?>**.

Подібним чином можна використовувати як обмеження власні типи. Наприклад, потрібно визначити функцію для умовного надсилання повідомлення:

```
1   fun <T:Message> send(message: T){
2       println(message.text)
3   }
4
5   interface Message{
6       val text: String
7   }
8   class EmailMessage( override val text:          String):
    Message
9   class SmsMessage( override val text: String): Message
```

Тут визначено інтерфейс **Message**, який містить одну властивість **text** та описує умовне повідомлення. Також тут є два класи, які реалізують цей інтерфейс, а саме **EmailMessage** та **SmsMessage**.

Функція **send()** використовує обмеження **<T:Message>**, тобто вона приймає об'єкт деякого типу, який має реалізувати інтерфейс **Message**.

Далі можна викликати цю функцію, передавши їй відповідний об'єкт, наприклад:

```
1   fun main() {
2       val email1 = EmailMessage( "Hello Kotlin" )
```

```
3       send(email1)
4       val sms1 = SmsMessage( "Привіт, не спиш?" )
5       send(sms1)
6   }
```

В цьому прикладі в обох викликах функція **send()** очікує на об'єкт **Message**. Однак також можна вказати точний тип, який буде використовуватись функцією.

5.2.1. Встановлення кількох обмежень

У наведеному раніше прикладі можна було передавати у функцію **getBiggest()** будь-який об'єкт, який реалізує інтерфейс **Comparable**. Але що потрібно модифікувати, якщо необхідно аби функція могла порівнювати лише числа? Варто згадати, що у мові Kotlin усі числові типи даних успадковуються від базового класу **Number**. Тож, можна задати ще одне обмеження, а саме щоб порівнюваний об'єкт представляв тип Number, наприклад:

```
1   fun <T>  getBiggest(a:  T,  b:  T):  T  where  T:
    Comparable<T>, T: Number {
2       return if(a > b) a
3       else b
4   }
```

Якщо параметру типу потрібно встановити кілька обмежень, то всі вони вказуються після типу функції, який повертається, після слова **where** через кому (**,**) у формі, наведені нижче:

```
1   параметр_типу: тип_обмежень
```

В цьому разі стає можливим передача у функцію об'єктів, які одночасно реалізують інтерфейс **Comparable** та є спадкоємцями класу **Number**, наприклад:

```
1   fun main() {
2
3       val result1 = getBiggest(1, 2)
4       println(result1) // 2
5
6       val result2 = getBiggest(1.6, -2.8)
7       println(result2) // 1.6
8
9       // val result3 = getBiggest("Taras", "Stepan") //
    ! Помилка - String не є похідною від класу Number
10      // println(result3)
11  }
```

Подібним чином можна використовувати власні типи як обмеження, наприклад:

```
1   fun main() {
2       val email1 = EmailMessage( "Hello Kotlin" )
```

```
3       send(email1)
4       val sms1 = SmsMessage( "Привіт, не спиш?" )
5       send(sms1)
6   }
7   fun <t>  send(message:  T)  where  T:  Message,  T:
    Logger{
8       message.log()
9   }
10
11  interface Message{ val text: String }
12  interface Logger{ fun log() }
13
14  class EmailMessage( override val text:      String):
    Message, Logger{
15      override fun log() = println( "Email: $text" )
16  }
17  class SmsMessage( override val text:        String):
    Message, Logger{
18      override fun log() = println( "SMS: $text" )
19  }
```

В цьому прикладі для функції **send()** встановлено два обмеження. Параметр типу **T** повинен представляти одночасно обидва інтерфейси і **Message**, і **Logger**.

5.2.2. Обмеження у класах

Класи, як і функції, можуть приймати обмеження на узагальнення. Наведемо приклад встановлення одного обмеження:

```
1   class Messenger<T:Message>(){
2       fun send(mes: T){
3           println(mes.text)
4       }
5   }
```

А ось приклад встановлення кількох обмежень:

```
1   fun main() {
2       val email1 = EmailMessage( "Hello Kotlin" )
3       val outlook = Messenger<EmailMessage>()
4       outlook.send(email1)
5
6       val skype = Messenger<SmsMessage>()
7       val sms1 = SmsMessage( "Привіт, не спиш?" )
8       skype.send(sms1)
9   }
10  class Messenger<T>() where T: Message, T: Logger{
11      fun send(mes: T){
12          mes.log()
```

```
13      }
14  }
15  interface Message{ val text: String }
16  interface Logger{ fun log() }
17
18  class EmailMessage( override val text:        String):
    Message, Logger{
19      override fun log() = println( "Email: $text" )
20  }
21  class SmsMessage( override val text:          String):
    Message, Logger{
22      override fun log() = println( "SMS: $text" )
23  }
```

Тут варто звернути увагу на те, що оскільки конструктор класу **Messenger** не приймає параметрів типу **T**, то необхідно явно вказати, який саме тип використовуватиметься, наприклад:

```
1   val outlook = Messenger<EmailMessage>()
```

В якості альтернативи можна було б явно вказати тип змінної:

```
1  val outlook:        Messenger<EmailMessage>        =
   Messenger()
```

5.3. Варіантність, коваріантність і контрваріантність

Варіантність визначає, як узагальнені типи, типізовані класами з однієї ієрархії успадковування, співвідносяться один з одним.

5.3.1. Інваріантність

Інваріантність передбачає, що, якщо є класи **Base** і **Derived**, де **Derived** похідний клас від **Base**, то клас **C<Base>** не є ні базовим класом для **C<Derived>** ні похідним. Наприклад, у є наступні типи:

```
1   interface Messenger<T: Message>()
2
3   open class Message( val text: String)
4   class EmailMessage(text: String): Message(text)
```

В цьому прикладі не можна присвоїти об'єкт **Messenger<EmailMessage>** змінній типу **Messenger<Message>** і навпаки, вони ніяк між собою не співвідносяться, незважаючи на те, що **EmailMessage** успадковується від **Message**, наприклад:

```
1   fun changeMessengerToEmail(obj:
    Messenger<EmailMessage>){
2       val messenger:  Messenger<Message>  =  obj //
    ! Помилка
```

```
}
fun changeMessengerToDefault(obj:
Messenger<Message>){
    val messenger:    Messenger<EmailMessage>    =
obj // ! Помилка
}
```

Можна присвоїти за замовчуванням змінним лише об'єкти їх типів, наприклад:

```
fun changeMessengerToDefault(obj:
Messenger<Message>){
    val messenger: Messenger<Message> = obj
}
fun changeMessengerToEmail(obj:
Messenger<EmailMessage>){
    val messenger: Messenger<EmailMessage> = obj
}
```

5.3.2. Коваріантність

Коваріантність передбачає, що, якщо є класи **Base** та **Derived**, де **Base** є базовим класом для **Derived**, то клас **SomeClass<Base>** є базовим класом для **SomeClass<Derived>**.

Для визначення узагальненого типу як коваріантного параметр узагальнення визначається з ключовим словом **out**, наприклад:

```
interface Messenger<out T: Message>
open class Message( val text: String)
class EmailMessage(text: String): Message(text)
```

В цьому прикладі інтерфейс **Messenger** є **коваріантним**, оскільки його параметр визначено зі словом **out**, а саме: **interface Messenger <out T>**. І тепер змінній типу **Messenger<Message>** можна надати значення типу **Messenger<EmailMessage>**, наприклад:

```
fun changeMessengerToEmail(obj:
Messenger<EmailMessage>){
    val messenger: Messenger<Message> = obj
}
```

Використання саме ключового слово **out** взагалі то не випадково. Воно вказує, що узагальнений тип може повертати з функції значення типу **T**, наприклад:

```
fun main() {

    val messenger:        Messenger<Message>        =
EmailMessenger()
    val message    =    messenger.writeMessage( "Hello
Kotlin" )
```

```
5       println(message.text) // Email: Hello Kotlin
6   }
7   open class Message(val text: String)
8   class EmailMessage(text: String): Message(text)
9
10  interface Messenger<out T: Message>{
11      fun writeMessage(text: String): T
12  }
13  class EmailMessenger(): Messenger<EmailMessage>{
14      override fun writeMessage(text:          String):
    EmailMessage {
15          return EmailMessage( "Email: $text" )
16      }
17  }
```

У цьому прикладі узагальнений інтерфейс **Messenger** визначає функцію **writeMessage()** для створення об'єкта **Message**. Клас **EmailMessenger** застосовує інтерфейс **Messenger** та реалізує цю функцію. В такому разі тип **EmailMessenger** по суті є типом **Messenger<EmailMessage>**.

Оскільки в **Messenger** параметр **T** визначений з анотацією **out**, то можна присвоїти змінній типу **Messenger<Message>** значення типу **EmailMessenger**, а по суті значення типу **Messenger<EmailMessage>**, наприклад:

```
1  val messenger:        Messenger<Message>        =
   EmailMessenger()
```

У той же час, тип **T** не можна використовувати як тип вхідних параметрів функції. Наприклад, у наступному коді компілятор сповістить нас про помилку:

```
1   interface Messenger<out T: Message>{
2       fun writeMessage(text: String): T
3       fun sendMessage(message:  T) // Помилка - тип T
    може представляти тільки тип, що повертається
4   }
```

5.3.3. Контрваріантність

Контрваріантність передбачає певною мірою зворотну ситуацію. Контрваріантність передбачає, що, якщо є класи **Base** і **Derived**, де **Base** базовий клас для **Derived**, то об'єкту **SomeClass<Derived>** можна надати значення **SomeClass<Base>**. При коваріантності ж навпаки, об'єкту **SomeClass<Base>** можна присвоїти значення **SomeClass<Derived>**.

Для визначення узагальненого типу як контрваріантного параметр узагальнення визначається з ключовим словом **in**, наприклад:

```
1   interface Messenger< in T: Message>
2   open class Message( val text: String)
```

```
3   class EmailMessage(text: String): Message(text)
```

В наведеному прикладі інтерфейс **Messenger** є **контраваріантним**, тому що його параметр визначено з ключовим словом **in**, а саме: **interface Messenger <in T>**. І тепер змінній типу **Messenger<EmailMessage>** можна надавати значення типу **Messenger<Message>**, наприклад:

```
1   fun changeMessengerToDefault(obj:
    Messenger<Message>){
2       val messenger: Messenger<EmailMessage> = obj
3   }
```

Застосування анотації **in** означає, що узагальнений тип може набувати значення типу **T** через параметр функції, наприклад:

```
1   fun main() {
2
3       val messenger:      Messenger<EmailMessage>      =
    InstantMessenger() //    InstantMessenger    -    це
    Messenger<Message>
4
5       val message = EmailMessage( "Hi Kotlin" )
6       messenger.sendMessage(message)
7   }
8   open class Message( val text: String)
9   class EmailMessage(text: String): Message(text)
10
11  interface Messenger< in T: Message>{
12      //fun writeMessage(text: String): T
13      fun sendMessage(message: T)
14  }
15
16  class InstantMessenger(): Messenger<Message>{
17      override fun sendMessage(message: Message){
18          println( "Send message: ${message.text}" )
19      }
20  }
```

У наведеному прикладі узагальнений інтерфейс **Messenger** визначає функцію **sendMessage()**, яка приймає об'єкт **Message** як параметр. Клас **InstantMessenger** застосовує інтерфейс **Messenger** та реалізує цю функцію. Тобто в цьому разі тип **InstantMessenger** по суті є типом **Messenger<Message>**.

Оскільки в інтерфейсі **Messenger** параметр **T** визначений за допомогою інструкції **in**, то можна присвоїти змінній типу **Messenger<EmailMessage>** значення типу **InstantMessenger**, тобто значення типу **Messenger<Message>**, наприклад:

```
1   val messenger:      Messenger<EmailMessage>      =
```

```
InstantMessenger()
```

У той же час тип **T** не можна використовувати як тип результату функції. Наприклад, у наступному коді компілятор сповістить нас про помилку:

```
1   interface Messenger< in T: Message>{
2       fun writeMessage(text: String): T // Помилка - тип
    T може представляти лише параметр функції
3       fun sendMessage(message: T)
4   }
```

Резюме

В наведеному розділі розглядаються узагальнені класи та функції (**generics**) у Kotlin, які дозволяють працювати з невизначеними наперед типами. Описано, як використовувати параметризовані типи в класах і функціях, а також застосування кількох параметрів одночасно. Ознайомлено з обмеженнями узагальнень, які допомагають контролювати допустимі типи через інтерфейси або базові класи. Також пояснені поняття варіантності: інваріантність (типи не є взаємозамінними), коваріантність (використання **out** дозволяє передавати типи у вихідних значеннях) і контрваріантність (використання **in** обмежує типи лише для вхідних параметрів). Усе це допомагає писати гнучкий і безпечний код, особливо при роботі з колекціями та інтерфейсами, без чого не можливо створення мобільних, десктопних та веб-застосунків мовою Kotlin.

Контрольні запитання і завдання

1. Що таке узагальнення (**generics**) у Kotlin?
2. Як оголошується узагальнений клас у Kotlin?
3. Як працює параметр **T** у **generics**-класах?
4. Чому використання узагальнених типів робить код гнучкішим?
5. Чи можна створювати об’єкти узагальнених класів без явного зазначення типу?
6. Як можна використовувати кілька узагальнених параметрів у класах?
7. Яка різниця між **Person<T>** і **Word<K, V>**?
8. Чи можуть узагальнені параметри мати різні типи у різних об’єктах одного класу?
9. Як оголошується узагальнена функція?
10. Що означає **<T>** перед назвою функції?
11. Чи можна викликати узагальнену функцію без вказівки типу?
12. Навіщо потрібні обмеження узагальнень у Kotlin?
13. Як використовується інтерфейс **Comparable** в узагальнених функціях?

14. Як обмежити узагальнені типи до певного класу або інтерфейсу?
15. Що означає синтаксис **where T: Number, T: Comparable<T>**?
16. Що таке інваріантність у **generics**?
17. Як працює коваріантність і для чого використовується **out**?
18. Що означає контрваріантність і яке призначення має ключове слово **in**?
19. У яких випадках узагальнені типи не можна використовувати у вхідних параметрах функцій?
20. Чим відрізняються **Messenger<out T>** і **Messenger<in T>**?
21. Створіть узагальнений клас **Box<T>**, який зберігає одне значення та має метод для його отримання.
22. Напишіть узагальнену функцію **swap<T>**, яка приймає масив і змінює місцями два елементи за індексами.
23. Оголосіть узагальнений клас **PairContainer<A, B>**, який зберігатиме два значення різних типів.
24. Створіть узагальнену функцію **findMax<T>**, яка приймає два значення і повертає більше з них. Обмежте параметр **T**, щоб можна було порівнювати значення.
25. Реалізуйте узагальнений клас **Logger<T>**, який працює тільки з класами, що реалізують інтерфейс **Loggable**. Інтерфейс повинен мати метод **log()**.
26. Оголосіть інтерфейс **Producer<out T>** з методом **produce()**, який повертає значення типу **T**. Реалізуйте його у класі **StringProducer**, який повертає рядок.
27. Створіть інтерфейс **Consumer<in T>** з методом **consume(value: T)**. Реалізуйте його у класі **IntConsumer**, який приймає та виводить числа.
28. Напишіть функцію **printMessages(messenger: Messenger<Message>)**, яка приймає об’єкт **Messenger** та викликає в ньому метод **sendMessage()**. Додайте клас **EmailMessenger**, який успадковується від **Messenger<EmailMessage>**, та передайте його у функцію.
29. Реалізуйте узагальнену функцію **getLongest<T>**, яка приймає два масиви та повертає найдовший.
30. Створіть узагальнений клас **Cache<T>**, який зберігає дані у **Map (MutableMap<String, T>)** і має методи **put(key: String, value: T)** та **get(key: String): T?**. Протестуйте його з різними типами.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Яка основна перевага використання generics у Kotlin?

a) Generics збільшують швидкість виконання програм

b) Generics дозволяють використовувати будь-який тип без перевірок

c) Generics дозволяють писати узагальнений код, що працює з різними типами

d) Generics змінюють стандартні типи Kotlin

Правильна відповідь: c).

Яке обмеження потрібно додати, щоб наступна функція працювала тільки з числами?

```
fun <T> sum(a: T, b: T): T {
    return a + b
}
```

a) <T: Any>

b) <T: Comparable<T>>

c) <T: Number>, оскільки тільки числові типи підтримують арифметичні операції

d) <T: String>

Закриті запитання з кількома правильними відповідями

Які з наведених класів є правильними generics-класами у Kotlin?

a) class Box<T>(val item: T)

b) class Container(var value: Any)

c) class Pair<K, V>(val first: K, val second: V)

d) class Data(val info: T)

Правильна відповідь: a), c).

Завдання на відповідність

Співвіднесіть ключові слова з їхнім призначенням:

a) out → 1. Використовується для вихідних значень у generics-класах

b) in → 2. Використовується для вхідних параметрів у функціях

c) where → 3. Використовується для встановлення обмежень на типи

Правильна відповідь:

a) → 1.

b) → 2.
c) → 3.

Розташуйте кроки створення generics-класу у правильному порядку:
a) Оголошення класу з параметром типу → 1. Крок 1
b) Використання параметра типу у полях і методах → 2. Крок 2
c) Створення об'єктів цього класу → 3. Крок 3
Правильна відповідь:
a) → 1.
b) → 2.
c) → 3.

Визначте тип узагальнення для кожного випадку:
a) class Box<T> → 1. Інваріантність
b) class Producer<out T> → 2. Коваріантність
c) class Consumer<in T> → 3. Контрваріантність
Правильна відповідь:
a) → 1.
b) → 2.
c) → 3.

Вставити пропущені слова

Доповніть код, щоб він компілювався без помилок:
```
class Storage<_____________>(val data: _____________)
```
Правильна відповідь: …T…, … T …

Завдання з кодом

Що виведе наступний код?
```
fun <T> printType(value: T) {
    println(value::class.simpleName)
}

fun main() {
    printType(42)
    printType("Hello")
}
```
a) Int Hello
b) 42 Hello

c) Int String

d) Compilation error

Правильна відповідь: c).

Відкриті запитання

Знайдіть помилку у коді generics-класу:

```
class Box<T> {
    fun put(item: T) { println(item) }
    fun get(): T { return "Test" } // Помилка
}
```

Правильна відповідь:

Необхідно змінити метод get() так, щоб він повертав значення типу T, а саме:

```
fun get(): T { return item }
```

При цьому item повинен бути змінною класу, визначеною в Box<T>.

У якому рядку цього коду буде помилка? Поясніть чому.

```
interface Processor<in T> {
    fun process(value: T)
    fun create(): T
}
```

Правильна відповідь: Помилка у методі create(), оскільки T позначено як in, а отже, його не можна використовувати у вихідному значенні.

6. ДОДАТКОВІ МОЖЛИВОСТІ ООП

6.1. Обробка винятків

Винятком називається подія, що виникає при виконанні програми та порушує її нормальний перебіг. Наприклад, при передачі файлу через мережу може обірватися мережне з'єднання, і внаслідок чого може бути згенеровано виняток. Якщо виняток не оброблено, програма «падає» і припиняє свою роботу. Тому у разі появи винятків їх слід обробляти.

Для обробки винятків у мові Kotlin застосовується конструкція **try..catch..finally**. У блоці **try** розташовують ті операції, які можуть викликати виняток (наприклад, передача файлу по мережі; відкриття файлу і т.і.). Блок **catch** перехоплює виняток і обробляє його. Блок **finally** виконує деякі завершальні операції, наприклад.

```
try {
    // код, що генерує виняток
}
catch (e: Exception) {
    // обробка винятку
}
finally {
    // постобробка
}
```

Після оператора **catch** у дужках зазначається параметр, який описує тип винятку. За цим параметром можна отримати інформацію про виняток.

Блок **finally** є необов'язковим, його можна не використовувати. Блок **catch** також може бути відсутній, однак обов'язково повинен бути блок **try** і як мінімум один із блоків або **catch**, або **finally**. Також ця конструкція може містити кілька блоків **catch** для обробки кожного типу винятків, що можуть виникнути.

Блок **catch** виконується, якщо виник виняток. Блок **finally** виконується у будь-якому разі, навіть якщо виняток не стався.

Наприклад, при діленні на нуль Kotlin генерує виняток:

```
fun main() {

    try {
        val x : Int = 0
        val z : Int = 0/x
        println( "z = $z" )
    }
    catch (e: Exception) {
        println( "Exception" )
```

```
10          println(e.message)
11      }
12  }
```

Дія, яка може викликати виняток, тобто операція ділення в наведеному прикладі, розташовується у блоці **try**. У блоці **catch** перехоплюється виняток. При цьому кожен виняток має певний тип. В наведеному вище прикладі використовується загальний тип винятків, а саме клас **Exception**, а результат роботи програми наведено на рисунку 6.1.

Рисунок 6.1. Результат роботи програми

Якщо є необхідність у якісь завершальних діях, їх можна додати у блоці **finally**. Наприклад, якщо під час роботи з файлом виникає виняток, то у блоці **finally** можна прописати закриття файлу, наприклад:

```
1   try {
2       val x : Int = 0
3       val z : Int = 0/x
4       println( "z = $z" )
5   }
6   catch(e: Exception){
7       println( "Exception" )
8       println(e.message)
9   }
10  finally {
11      println( "Program has been finished" )
12  }
```

У цьому прикладі результат роботи програми буде мати вигляд як показано на рисунку 6.2.

Рисунок 6.2. Результат роботи програми

6.1.1. Інформація про винятки

Базовий клас винятків **Exception** реалізує низку властивостей, які дозволяють отримати різну інформацію про виняток:

- **message**: повідомлення про виняток;
- **stackTrace**: трасування стеку винятку, що складається з набору рядків, де було згенеровано виняток.

Серед низки функцій класу **Exception** слід виділити функцію **printStackTrace()**, яка виводить інформацію, яка зазвичай відображається при необробленому винятку.

Розглянемо приклади застосування властивостей:

```
1   fun main() {
2
3       try {
4           val x : Int = 0
5           val z : Int = 0/x
6           println( "z = $z" )
7       }
8       catch (e: Exception) {
9           println(e.message)
10          for (line in e.stackTrace) {
11              println( "at $line" )
12          }
13      }
14  }
```

Результат роботи програми наведено на рисунку 6.3.

```
/ by zero
at AppKt.main(app.kt:5)
at AppKt.main(app.kt)
```

Рисунок 6.3. Результат роботи програми

6.1.2. Обробка кількох винятків

Одна програма чи один код може генерувати одразу кілька винятків. Для обробки кожного окремого типу винятків можна визначити окремий блок **catch**. Наприклад, для одного винятку необхідно виконати одні дії, а для іншого – інші:

```
1   try {
2       val nums = arrayOf(1, 2, 3, 4)
3       println(nums[6])
4   }
5   catch (e:ArrayIndexOutOfBoundsException){
6       println( "Out of bound of array" )
7   }
8   catch (e: Exception) {
9       println(e.message)
10  }
```

В наведеному прикладі при доступі за недійсним індексом у масиві буде генеруватися виняток типу **ArrayIndexOutOfBoundsException**, який буде оброблено за допомогою блоку **catch(e:ArrayIndexOutOfBoundsException)**.

Якщо ж в програмі будуть інші винятки, які не представляють тип **ArrayIndexOutOfBoundsException**, то вони будуть оброблятись другим блоком **catch**, тому що **Exception** це загальний тип, який підходить під всі типи винятків. При цьому варто зазначити, що спочатку обробляється виняток більш вузького типу, а саме **ArrayIndexOutOfBoundsException**, і тільки потім більш узагальненого типу **Exception**.

6.1.3. Оператор throw

Можливо, у деяких ситуаціях потрібно самостійно вручну згенерувати виняток. Для генерації винятків застосовується оператор **throw**, після якого вказується об'єкт винятку.

Наприклад, у функції перевірки віку необхідно згенерувати виняток, якщо вік не вкладається в деякий діапазон:

```
1   fun main() {
2
3       val checkedAge1 = checkAge(5)
4       val checkedAge2 = checkAge(-115)
5   }
6   fun checkAge(age: Int): Int{
7       if (age < 1 || age > 110) throw Exception( "Invalid
    value $age. Age must be greater than 0 and less than
    110" )
8       println( "Age $age is valid" )
9       return age
10  }
```

Після оператора **throw** вказано об'єкт винятку. Для визначення об'єкта **Exception** застосовується конструктор, який приймає як параметр повідомлення про виняток. У наведеному прикладі це повідомлення про некоректність введеного значення.

Якщо при виклику функції **checkAge**() до неї буде передано число менше 1 або більше 110, то буде згенеровано виняток. Тож, результат роботи програми буде таким, як показано на рисунку 6.4.

```
Age 5 is valid
Exception in thread "main" java.lang.Exception: Invalid value -115. Age must be
greater than 0 and less than 110
 at AppKt.checkAge(app.kt:7)
 at AppKt.main(app.kt:4)
 at AppKt.main(app.kt)
```

Рисунок 6.4. Результат роботи програми

Але, оскільки виняток, що генерується в цьому прикладі, не оброблений, то програма при генерації винятку аварійно завершить роботу. Щоб цього не сталося, потрібно обробити виняток, що генерується, наприклад:

```
1   fun main() {
2       try {
3           val checkedAge1 = checkAge(5)
4           val checkedAge2 = checkAge(-115)
5       }
6       catch (e: Exception) {
7           println(e.message)
8       }
9   }
10  fun checkAge(age: Int): Int{
11      if (age < 1 || age > 110) throw Exception ("Invalid
    value $age. Age must be greater than 0 and less than
    110" )
12      println( "Age $age is valid" )
13      return age
14  }
```

6.1.4. Повернення значення

У мові Kotlin конструкція **try** може повертати значення, наприклад:

```
1   fun main() {
2       val checkedAge1  = try {  checkAge(5)  } catch (e:
    Exception) { null }
3       val checkedAge2       = try {       checkAge(-125)
    } catch (e: Exception) { null }
4       println(checkedAge1) // 5
5       println(checkedAge2) // null
6   }
7   fun checkAge(age: Int): Int{
8       if (age       <       1       ||       age       >
    110) throw Exception( "Invalid value $age. Age must
    be greater than 0 and less than 110" )
9       println( "Age $age is valid" )
10      return age
11  }
```

У цьому прикладі змінна **checkedAge1** отримує результат функції **checkAge()**. Якщо станеться виняток, тоді змінна **checkedAge1** отримає те значення, яке вказано в блоці **catch**, тобто в наведеному прикладі значення **null**.

За необхідності в блок **catch** можна додати інші вирази або повернути інше значення, наприклад:

```
1   fun main() {
```

```
2       val checkedAge2       = try {       checkAge(-125)
    } catch (e: Exception) { println(e.message); 18 }
3       println(checkedAge2)
4   }
5   fun checkAge(age: Int): Int{
6       if (age       <       1       ||       age       >
    110) throw Exception( "Invalid value $age. Age must
    be greater than 0 and less than 110" )
7       println( "Age $age is valid" )
8       return age
9   }
```

В цьому прикладі, якщо буде згенеровано виняток, то конструкція **try** виведе у консоль інформацію про виняток і поверне число 18. Значення, що повертається, вказується після всіх інших інструкцій в блоці **catch**.

6.2. Null і nullable-типи

Ключове слово **null** представляє спеціальний літерал, який вказує, що змінна як така не має ні якого значення. Тобто, у неї по суті відсутнє значення, а саме:

```
1   val n = null
2   println(n) // null
```

Подібне значення може бути корисним у низці випадках, коли необхідно використовувати дані, але при цьому точно невідомо, а чи є насправді ці дані. Наприклад, при отриманні даних через мережу, дані можуть прийти або не прийти. Або може бути ситуація, коли необхідно явно вказати, що дані не встановлені.

Однак змінним стандартних типів, наприклад, типу **Int** або **String** або будь-яких інших класів, не можна просто взяти та присвоїти значення **null**, наприклад:

```
1   val n:  Int  = null //! Помилка,  змінна  типу  Int
    допускає лише числа
```

Надати значення **null** можна тільки змінній, яка представляє тип **Nullable**. Щоб перетворити звичайний тип на тип **nullable**, достатньо поставити після назви типу знак питання (**?**), наприклад:

```
1   // val n: Int = null //! помилка, Int не допускає
    значення null
2   val d  :  Int? = null //  нормально,  Int? допускає
    значення null
```

При цьому можна передавати змінним **nullable-типів** як значення **null**, так і конкретні значення, що укладаються в діапазон значень цього типу, наприклад:

```
1  var age: Int? = null
2  age = 34 // Int? допускає null та числа
```

```
3  var name: String? = null
4  name= "Taras" // String? допускає null та рядки
```

Nullable-типи можуть представляти створювані розробником власні класи, наприклад:

```
1   fun main() {
2
3       var volodymyr: Person = Person( "Volodymyr" )
4       // volodymyr = null //! Помилка - volodymyr
    представляє тип Person і не допускає null
5       var taras: Person? = Person( "Taras" )
6       taras = null // Норм - taras представляє тип
    Person? і припускає null
7   }
8   class Person( val name: String)
```

У той же час слід розуміти, що **String?** та **Int?** – це не те саме, що і **String** і **Int**. **Nullable-типи** мають ряд обмежень, а саме:

1. Значення **nullable-типів** не можна присвоїти безпосередньо змінним, які допускають значення **null**, наприклад:

```
1   var message: String? = "Hello"
2   val hello: String = message //! Помилка - Hello не
    допускає значення null
```

2. У об'єктів **nullable-типів** не можна викликати ті ж функції та властивості, які є у звичайних типів, наприклад:

```
1   var message: String? = "Hello"
2   // у типі String властивість length повертає
    довжину рядка
3   println( "Message                    length:
    ${message.length}" ) // ! Помилка
```

3. Не можна передавати значення **nullable-типів** як аргумент у функцію, де потрібне конкретне значення, яке може представляти **null**.

6.2.1. Оператор ?:

Однією з переваг мови Kotlin є те, що її система типів дозволяє виявляти проблеми, пов'язані з використанням **null**, під час компіляції, а не під час виконання. Наприклад, розглянемо наступний код:

```
1   var name: String? = "Taras"
2   val userName: String = name //! Помилка
```

Змінна **name** зберігає рядок **"Taras"**. Змінна **userName** представляє тип **String** і також може зберігати рядки, проте безпосередньо не можна передати значення зі змінної **name** в **userName**. В наведеному прикладі компілятору

невідомо, яким значенням ініціалізована змінна **name**. Адже змінна **name** може містити і значення **null**, яке є неприпустимим для типу **String**.

У цьому випадку можна використовувати оператор **?:** , який дозволяє надати альтернативне значення, якщо значення, що присвоюється, дорівнює **null**, наприклад:

```
1   var name: String? = "Taras"
2   val userName: String = name?: "Undefined" // якщо name
    = null, то присвоюється "Undefined"
3
4   var age: Int? = 23
5   val userAge: Int = age?: 0 // якщо age дорівнює null,
    то присвоюється число 0
```

Оператор **?:** приймає два операнди. Якщо перший операнд не дорівнює **null**, то повертається значення першого операнда. Якщо перший операнд дорівнює **null**, то повертається значення другого операнда.

Тобто це все одно, якби було написано наступним чином:

```
1   var name: String? = "Taras"
2   val userName: String
3   if (name!= null ){
4
5       userName = name
6   }
```

Отже, оператор **?:** дозволяє скоротити подібну конструкцію.

6.2.2. Оператор ?.

Оператор **?.** дозволяє об'єднати перевірку значення об'єкта на **null** та звернутися до функцій чи властивостей цього об'єкта.

Наприклад, рядки мають властивість **length**, яка повертає довжину рядка в символах. У об'єкті **String?** просто так не можна звернутися до властивості **length**, тому що, якщо об'єкт **String?** дорівнює **null**, то рядку як такого не існує, і відповідно довжину рядка не можна визначити. Саме в цьому випадку можна застосувати оператор **?.** , наприклад:

```
1   var message: String? = "Hello"
2   val length: Int? = message?.length
```

Якщо змінна **message** раптом дорівнюватиме **null**, то змінна **length** отримає значення **null**. Якщо ж змінна **name** міститиме рядок, то повернеться довжина цього рядка. Насправді вираз **val length: Int? = message?.length** еквівалентний наступному коду:

```
1   val length: Int?
2   if (message != null )
```

```
3       length = message.length
4   else
5       length = null
```

Отже. за допомогою оператора **?.** подібним чином можна звертатися до будь-яких властивостей та функцій об'єкта.

Також, у наведеному вище прикладі, можна було б поєднати обидва вище розглянуті оператори, а саме:

```
1   val message: String? = "Hello"
2   val length: Int = message?.length ?:0
```

Тепер змінна **length** не допускає значення **null**. І якщо змінна **name** не визначена, то **length** отримає число 0.

Використовуючи оператор **?.** , можна створювати ланцюжки перевірок на **null**, наприклад:

```
1   fun main() {
2
3       var taras: Person? = Person( "Taras" )
4       val tarasName: String? = taras?.name?.uppercase()
        println(tarasName) // TOM
5
6       var volodymyr: Person? = null
7       val volodymyrName:                          String? =
    volodymyr?.name?.uppercase()
8       println(volodymyrName) // null
9
10      var stepan: Person? = Person( null )
11      val stepanName:                             String? =
    stepan?.name?.uppercase()
12      println(stepanName) // null
13
14  }
15  class Person( val name: String?)
```

В цьому прикладі клас **Person** у первинному конструкторі набуває значення типу **String?**, тобто це може бути рядок, а може бути **null**.

Припустимо, потрібно отримати передане через конструктор ім'я користувача у верхньому регістрі, тобто великими літерами. Для переведення тексту у верхній регістр клас **String** має функцію **uppercase()**. Однак може скластися ситуація, коли або об'єкт **Person** дорівнює **null**, або його властивість **name**, яка представляє тип **String?**, дорівнюватиме **null**. І саме в цьому випадку перед викликом функції **uppercase()** потрібно перевірити всі ці об'єкти на **null**. А оператор **?.** дозволяє скоротити код перевірки, наприклад:

```
1   val tarasName: String? = taras?.name?.uppercase()
```

Тобто якщо **taras** не дорівнює **null**, то звертаємось до його властивості **name**. Далі якщо **name** не дорівнює **null**, то звертаємось до її функції **uppercase()**. Якщо якась ланка в цій перевірці поверне **null**, то змінна **tarasName** теж дорівнюватиме **null**.

Але тут так само можна уникнути фінального повернення **null** і присвоїти значення за замовчуванням, наприклад:

```
1   val tarasName: String = taras?.name?.uppercase()
    ?: "Undefined"
```

6.2.3. Оператор !!

Оператор **!!** (**not-null assertion operator**) приймає один операнд. Якщо операнд дорівнює **null**, то генерується виняток. Якщо операнд не дорівнює **null**, то повертається його значення, наприклад:

```
1   fun main() {
2       try {
3           val name: String? = "Taras"
4           val id: String = name!!
5           println(id)
6       } catch (e: Exception) { println(e.message)}
7   }
```

Оскільки цей оператор повертає об'єкт, який не представляє **nullable-тип**, то після застосування оператора можна звертатися до методів та властивостей цього об'єкта, а саме:

```
1   val name: String? = null
2   val length :Int = name!!.length
```

6.3. Делеговані властивості

Делеговані властивості дозволяють делегувати отримання або присвоєння їх значення ззовні, тобто іншому класу. Це дозволяє програмістам додавати деяку додаткову логіку під час операції з властивостями, наприклад: логування; якусь передобробку, тощо.

Формальний синтаксис делегованої властивості наступний:

```
1   val / var ім'я_властивості: тип_даних by вираз
```

Після типу даних властивості вказується ключове слово **by**, з подальшим записом виразу. Вираз представляє клас, що умовно називається делегатом. Делегати властивостей можуть не застосовувати жодних інтерфейсів, однак вони повинні надавати функції **getValue()** та **setValue()**. А виконання методів доступу **get()** і **set()**, які є у властивості, делегується функціям **getValue()** та **setValue()** класу делегата.

Варто зазначити, що не можна оголошувати делеговані властивості у первинному конструкторі.

6.3.1. Делеговані властивості для читання

Для властивостей лише читання, тобто **val**-властивостей, делегат повинен надавати функцію **getValue()**, яка приймає наступні параметри:

- **thisRef**: повинен представляти той самий тип, як і властивість, до якого застосовується делегат. Це може бути і батьківський тип;
- **property**: повинен представляти той самий тип **KProperty<*>** або його батьківський тип.

При цьому функція **getValue()** повинна повертати результат того ж типу, що і тип властивості, або його похідного типу.

Розглянемо наступний приклад:

```
1   import kotlin.reflect.KProperty
2
3   fun main() {
4
5       val taras = Person()
6       println(taras.name) // Taras
7
8       val volodymyr = Person()
9       println(volodymyr.name) // Taras
10  }
11  class Person{
12      val name: String by LoggerDelegate()
13  }
14  class LoggerDelegate {
15      operator fun getValue(thisRef: Person, property:
    KProperty<*?>): String {
16          println( "Запитана                властивість:
    ${property.name}" )
17          return "Taras"
18      }
19  }
```

В цьому прикладі клас **Person** визначає властивість **name**, яка є делегованою. Вона делегує операції отримання значення функції **getValue()** класу **LoggerDelegate**.

Оскільки властивість визначено у класі **Person**, то перший параметр функції **getValue()** представляє тип **Person**. Завдяки цьому можна отримати з цього параметра якусь додаткову інформацію про об'єкт, якщо вона потрібна.

Оскільки властивість представляє тип **String**, то функція також повертає значення типу **String**, яке буде повертатися самою властивістю **name**. У наведеному вище прикладі повертатиметься рядок **"Taras"**. Тобто при кожному зверненні до властивості **name** об'єкта **Person** повертатиметься рядок **"Taras"**.

Дещо видозмінимо код прикладу, а саме:

```
1  import kotlin.reflect.KProperty
2  fun main() {
3      val taras = Person( "Taras" )
4      println(taras.name)
5
6      val volodymyr = Person( "Volodymyr" )
7      println(volodymyr.name)
8  }
9  class Person(_name: String){
10     val name: String by LoggerDelegate(_name)
11 }
12 class LoggerDelegate( val personName: String) {
       operator fun getValue(thisRef:          Person,
13 property: KProperty<*>): String {
           println( "Запитана              властивість
14 ${property.name}" )
           println( "Значення,   що   встановлюється:
15 $personName" )
           return personName
17     }
18 }
```

Тепер первинний конструктор **Person** приймає встановлюване значення для властивості **name**. Далі воно передається до конструктора класу **LoggerDelegate**, який використовує його для логування за допомогою консолі. І наприкінці повертає його як значення властивості **name**.

6.3.2. Змінювані властивості

Для змінюваних властивостей, тобто **var**-властивостей, делегат повинен також надати функцію **setValue()**, яка приймає наступні параметри:

- **thisRef**: повинен представляти той самий тип, як і властивість, до якої застосовується делегат. Це може бути і батьківський тип;
- **property**: повинен представляти той самий тип **KProperty<*>** або його батьківський тип;
- **value**: повинен представляти той самий тип, як і властивість, або його батьківський тип.

Розглянемо наступний приклад:

```
1   import kotlin.reflect.KProperty
2
3   fun main() {
4
5       val taras = Person( "Taras" , 37)
6       println(taras.age) // 37
7       taras.age = 38
8       println(taras.age) // 38
9       taras.age = -139
10      println(taras.age) // 38
11
12  }
13  class Person( val name: String, _age: Int) {
14      var age: Int by LoggerDelegate(_age)
15  }
16  class LoggerDelegate(private var personAge: Int) {
17      operator      fun getValue(thisRef:      Person,
    property: KProperty<*>): Int{
18          return personAge
19      }
20      operator      fun setValue(thisRef:      Person,
    property: KProperty<*>, value: Int){
21          println( "Значення,    що    встановлюється:
    $value" )
22          if(value  >  0  &&  value  <  110)  personAge  =
    value
23      }
24  }
```

В цьому прикладі клас **Person** визначає делеговану властивість **age**. Вона делегує встановлення та отримання значення класу **LoggerDelegate** та його функціям **getValue()** та **setValue()**. Саме ж значення зберігається у властивості **personAge** класу **LoggerDelegate**. А функція **getValue()** просто повертає значення цієї властивості.

Функція **setValue()** за допомогою третього параметра **value**, який представляє той самий тип, як і властивість, тобто тип **Int**, отримує значення, що встановлюється. І якщо воно відповідає деякому діапазону, то передається у властивість **personAge**. Результат роботи програми наведено на рисунку 6.5.

```
37
Значення, що встановлюється: 38
38
Значення, що встановлюється: -139
38
```

Рисунок 6.5. Результат роботи програми

6.4. Перетворення типів

Нерідко може виникати необхідність у перетворенні типів, наприклад, щоб використовувати дані одного типу в контексті, де потрібні дані іншого типу. У цьому випадку мова Kotlin передбачає низку можливостей з перетворення типів.

6.4.1. Вбудовані методи перетворення типів

Для перетворення даних одного типу на інший можна використовувати наступні вбудовані функції, які є у базових типів **Int**, **Long**, **Double** і т.д., проте варто зазначити, що конкретний набір функцій для різних базових типів може дещо відрізнятися. Наведемо кілька таких функцій:

- **toByte();**
- **toShort();**
- **toInt();**
- **toLong();**
- **toFloat();**
- **toDouble();**
- **toChar().**

Всі ці функції перетворюють дані на той тип, що вказано після префікса **to**, наприклад **toByte**. Або наприклад:

```
1   val s: String = "12"
2   val d: Int = s.toInt()
3   println(d)
```

У цьому прикладі рядок **s** перетворюється на число **d**. Просто так передати рядок змінній типу **Int** не можна, незважаючи на те, що рядок начебто і містить число 12, наприклад:

```
1   val d: Int = "12" //! Помилка
```

Однак треба враховувати, що значення не завжди може бути перетворено до певного типу. В цьому випадку генерується виняток. Відповідно в таких випадках бажано відстежувати згенеровані винятки, наприклад:

```
1   val s: String = "taras"
2   try {
```

```
3       val d: Int = s.toInt()
4       println(d)
5   }
6   catch (e: Exception) {
7       println(e.message)
8   }
```

6.4.2. Smart cast та оператор is

Оператор **is** дозволяє перевіряти вираз на належність до певного типу даних. Формальній опис наступний:

```
1   значення is тип_даних
```

Цей оператор повертає **true**, якщо значення ліворуч від оператора належить типу, вказаному праворуч від оператора. Якщо локальна змінна або властивість успішно пройде перевірку на належність певного типу, далі немає потреби додатково приводити значення до цього типу. Ці перетворення ще називають **smart casts** або «розумні перетворення».

Оператор **is** можна застосовувати як до базових типів, так і до власних класів та інтерфейсів, які знаходяться в одній і тій самій ієрархії успадковування, наприклад:

```
1   fun main() {
2
3       val taras = Person( "Taras" )
4       val volodymyr =
    Employee( "Volodymyr" , "JetBrains" )
5
6       checkEmployment(taras) //  Taras  не  має  місця
    роботи
7       checkEmployment(volodymyr) //  Volodymyr  працює  в
    JetBrains
8   }
9
10  fun checkEmployment(person: Person){
11      //      println("${person.name}      works      in
    ${person.company}") //  Помилка  -  у  Person  немає
    властивості company
12      if (person is Employee) {
13          println( "${person.name}       працює       в
    ${person.company}" )
14      }
15      else {
16          println( "${person.name}    не    має    місця
    роботи" )
17      }
```

```
18 }
19 open class Person( val name: String)
20 class Employee(name:   String, val company:   String):
   Person(name)
```

В цьому прикладі клас **Employee** успадковується від класу **Person**. У функції **checkEmployment()** отримується об'єкт **Person**. За допомогою оператора **is** перевіряється, чи він належить типу **Employee**, бо не кожен об'єкт **Person** може представляти тип **Employee**. Якщо він представляє тип **Employee**, то виводиться назва його компанії, якщо ж він не представляє тип **Employee**, то виводиться повідомлення, що він безробітний.

Причому навіть якщо значення представляє тип **Employee**, то до застосування оператора **is** воно тим не менше належить типу **Person**. І тільки застосування оператора **is** перетворює значення типу **Person** в тип **Employee**.

Також можна застосовувати іншу форму цього оператора, а саме **!is**. Вона повертає **true** якщо значення НЕ представляє вказаний тип даних, наприклад:

```
1  fun main() {
2
3      val taras = Person( "Taras" )
4      val volodymyr                                    =
   Employee( "Volodymyr" , "JetBrains" )
5
6      checkEmployment(taras) //  Taras  не  має  місця
   роботи
7      checkEmployment(volodymyr) //  Volodymyr  працює  в
   JetBrains
8  }
9
10 fun checkEmployment(person: Person){
11     //      println("${person.name}      works      in
   ${person.company}") //  Помилка  -  у  Person  немає
   властивості company
12     if (person ! is Employee) {
13         println( "${person.name}  не  має  місця  роботи
   " )
14     }
15     else {
16         println( "${person.name}       працює       в
   ${person.company}" )
17     }
18 }
19 open class Person( val name: String)
20 class Employee(name:   String, val company:   String):
   Person(name)
```

Розглянемо, що трапиться, якщо властивість **company** має порожній рядок, наприклад, **val volodymyr = Employee("Volodymyr", "")**, тобто, компанія насправді не вказана. При цьому вимагається, щоб назва компанії виводилась на консоль, якщо ця властивість має який-небудь вміст. У цьому випадку можна виконати перевірку на довжину рядка відразу після застосування оператора **is**, наприклад:

```
1   fun checkEmployment(person: Person){
2       if (person is Employee && person.company.length > 0){
3           println( "${person.name} працює в ${person.company}" )
4       }
5       else {
6           println( "${person.name} не має місця роботи" )
7       }
8   }
```

В цьому прикладі у виразі:

```
1   person.company.length > 0) {
```

компілятор вже бачить, що **person** – це об'єкт типу **Employee**, тому дозволяє звертатися до його властивостей та функцій.

Якщо необхідно визначити різні дії в залежності від типу об'єкта, то зручно використовувати конструкцію **when**, наприклад:

```
1   fun identifyPerson(person: Person){
2       when (person){
3           is Manager -> println( "${person.name} is a manager" )
4           is Employee -> println( "${person.name} is an employee" )
5           is Person -> println( "${person.name} is just a person" )
6       }
7   }
8
9   open class Person( val name: String)
10  open class Employee(name: String, val company: String): Person(name)
11  class Manager(name: String, company: String): Employee(name, company)
```

6.4.2.1. Обмеження розумних перетворень

Подібні **smart-перетворення** мають певні обмеження. Вони можуть застосовуватися лише якщо компілятор може гарантувати, що змінна не змінила

свого значення у проміжку між перевіркою та використанням. Для **smart-перетворень** діють наступні правила:

- **smart-перетворення** застосовуються до локальних **val-змінних**, за винятком делегованих властивостей;
- **smart-перетворення** застосовуються до **val-властивостей**, за винятком властивостей з модифікатором **open**, тобто відкритих до перевизначення у похідних класах, або властивостей, для яких явно визначено гетер;
- **smart-перетворення** застосовуються до локальних **var-змінних**, якщо змінна не змінює свого значення у проміжку між перевіркою та використанням і не використовується у лямбда-виразі, який змінює її, а також не є локальною делегованою властивістю;
- до **var-властивостей smart-перетворення** не застосовуються.

Розглянемо деякі приклади. Візьмемо останнє правило про **var-властивості**, а саме:

```
1   fun main() {
2
3       val taras = Person( "Taras" )
4
5       if (taras.phone is SmartPhone) {
6
7           println( "SmartPhone: ${taras.phone.name}, OS:
    ${taras.phone.os}" ) // ! Помилка
8       }
9       else {
10          println( "Phone: ${taras.phone.name}" )
11      }
12  }
13  open class Phone( val name: String)
14  class SmartPhone(name:   String, val os:   String)   :
    Phone(name)
15
16  open class Person( val name: String){
17      var phone:   Phone   =   SmartPhone( "Pixel   5",
    "Android" )
18  }
```

В цьому прикладі клас **Person** зберігає **var-властивість** класу **Phone**, якому присвоюється об'єкт класу **SmartPhone**. Відповідно у виразі:

```
1   if (taras.phone is SmartPhone) {
```

видно, що властивість **taras.phone** є класом **SmartPhone**, проте оскільки ця властивість визначена за допомогою ключового слова **var**, то до неї не

застосовуються **smart-перетворення**. Тобто, якщо в наведеному вище прикладі замінити **var** на **val** , то ні яких проблем не виникне.

Для розгляду другого правила змінимо клас **Person**, визначивши для якості гетер, наприклад:

```
1   open class Person( val name: String){
2       val phone: Phone
3           get() =                 SmartPhone( "Pixel
    5" , "Android" )
4   }
```

Відповідно до другого правила знову ж таки до такої властивості не застосовуються **smart-перетворення**, тому що вона має гетер.

6.4.3. Явні перетворення та оператор as

За допомогою оператора **as** можна перетворювати значення одного типу до іншого типу, тобто:

```
1   значення as тип_даних
```

Зліва від оператора **as** вказується значення, а праворуч – тип даних, у який потрібно перетворити значення. Наприклад, перетворимо значення типу **String?** в тип **String**:

```
1   fun main() {
2
3       val hello: String? = "Hello Kotlin"
4       val message: String = hello as String
5       println(message)
6   }
```

Тут змінна **hello** зберігає рядок і може бути успішно перетворена на значення типу **String**. Однак що відбудеться, якщо змінна **hello** дорівнюватиме **null**, наприклад:

```
1   val hello: String? = null
2   val message: String = hello as String
3   println(message)
```

То у цьому випадку перетворення завершиться невдало, адже значення **null** не можна перетворити до значення типу **String**. Тому буде згенеровано виняток **NullPointerException**.

Щоб уникнути генерації винятків, можна застосовувати більш безпечну версію оператора **as**, а саме **as?**, яка у разі невдачі перетворення повертає **null**, наприклад:

```
1   val hello: String? = null
2       val message: String? = hello as? String
3       println(message)
```

У цьому прикладі оператор **as?** поверне **null**, тому що **null** не можна перетворити на значення типу **String**.

Також можна застосовувати оператор **as?** до перетворення власних типів, наприклад:

```
1   fun main() {
2
3       val taras = Person( "Taras" )
4       val volodymyr                =                Employee
    ( "Volodymyr" , "JetBrains" )
5       checkCompany(taras)
6       checkCompany(volodymyr)
7   }
8   fun checkCompany(person: Person){
9       val employee=person as ? Employee
10      if (employee!= null ){
11          println( "${employee.name}       працює       в
    ${employee.company}" )
12      }
13      else {
14          println( "${person.name} не має місця роботи
    " )
15      }
16  }
17  open class Person( val name: String)
    open class Employee(name:          String, var company:
18  String): Person(name)
```

В цьому прикладі функція **checkCompany()** приймає об'єкт класу **Person** і намагається перетворити його на об'єкт типу **Employee**, який успадкований від **Employee**. Але якщо кожен об'єкт **Employee** також представляє об'єкт **Person**, тобто кожен працівник є людиною, то не кожен об'єкт **Person** представляє об'єкт **Employee**, тобто не кожна людина є працівником. В цьому випадку, щоб отримати значення типу **Employee**, застосовується оператор **as?**. Якщо об'єкт **Person** є типом **Employee**, то повертається об'єкт цього типу, інакше повертається **null**. І далі вже можна перевіряти значення **null** і виконувати ті чи інші операції. Результат роботи програми наведено на рисунку 6.6.

```
Taras не має місця роботи
Volodymyr працює в JetBrains
```

Рисунок 6.6. Результат роботи програми

6.5. Функції розширення

Функції розширення (**extension function**) дозволяють додати функціонал до вже попередньо визначених типів. При цьому типи можуть бути визначені в іншому місці, наприклад, у стандартній бібліотеці.

Функція розширення визначається наступним чином:

```
fun тип.ім'я_функції(параметри)      :      тип_що_
повертається {
    тіло функції
}
```

За великим рахунком визначення аналогічне до визначення звичайної функції за винятком того, що після ключового слова **fun** вказується назва типу, для якого визначається функція, і через крапку (**.**) назва функції.

Визначимо кілька функцій розширення до стандартних типів **Int** та **String**, наприклад:

```
fun main() {

    val hello: String = "hello world"
    println(hello.wordCount( 'l' )) // 3
    println(hello.wordCount( 'o' )) // 2
    println(4.square()) // 16
    println(6.square()) // 36
}

fun String.wordCount(c: Char) : Int{
    var count = 0
    for (n in this ){
        if (n == c) count++
    }
    return count
}
fun Int.square(): Int{
    return this * this
}
```

В наведеному прикладі для типу **Int** визначено функцію зведення у квадрат. У кожній функції розширення через ключове слово **this** можна посилатися на поточний об'єкт того типу, для якого створюється функція. Наприклад, як у функції:

```
fun Int.square(): Int{
    return this * this
}
```

В цьому фрагменті коду через **this** звертаються до того об'єкта, для якого буде викликатися функція. І потім її можна буде викликати наступним чином:

```
4.square() // 16
```

Для типу **String** визначено функцію **wordCount**, яка підраховує скільки разів зустрічається певний символ у рядку.

Слід враховувати, що у функціях розширення можна звертатися до будь-яких загальнодоступних властивостей та методів об'єкта, проте не можна звертатися до властивостей та методів з модифікаторами **private** та **protected**.

Також слід враховувати, що функції розширення не перевизначають функції, які вже визначені у класі. Якщо функція розширення має таку ж сигнатуру, що і вже наявна функція класу, то компілятор буде просто ігнорувати подібну функцію розширення.

6.5.1. Scope-функції

Scope-функції, які можна перекласти як «функції контексту» або «функції області видимості», дозволяють виконати для деякого об'єкта деякий код у вигляді лямбда-виразу. При виклику подібної функції створюється тимчасова область видимості. У цій області видимості можна звертатися до об'єкта без його імені.

У мові Kotlin є п'ять подібних функцій, а саме: **let**, **run**, **with**, **apply** та **also**. Ці функції схожі за своєю дією і розрізняються насамперед за параметрами та результатами, що повертаються.

6.5.1.1. Функція let

Лямбда-вираз у функції **let** як параметр **it** отримує об'єкт, для якого викликається функція. Результатом функції **let**, що повертатиметься, є результат даного лямбда-виразу, наприклад:

```
inline fun <T, R> T.let(block: (T) -> R): R
```

Поширеним сценарієм, де застосовується ця функція, є перевірка на **null**, наприклад:

```
fun main() {

    val stepan = Person( "Stepan" , "stepan@gmail.com" )
    stepan.email?.let{ println( "Email: $it" ) } // Email: stepan@gmail.com
    // аналог без функції let
    //if(stepan.email!=null) println("Email:${stepan.email}")

    val taras = Person( "Taras" , null )
    taras.email?.let{ println( "Email: $it" ) } //
```

```
    функція let не виконуватиметься
10
11  }
12  data class Person( val name:      String, val email:
    String?)
```

Припустимо, що необхідно вивести значення властивості **Email** об'єкта **Person**. Але ця властивість може зберігати значення **null**, наприклад, якщо електронна адреса користувача не встановлена. За допомогою виразу **stepan.email?** перевіряємо значення властивості **email** на **null**. Якщо **email** не дорівнює **null**, то у рядку для властивості **email** викликається функція **let**, яка створює тимчасову область видимості і передає в неї значення властивості **email** через параметр **it**. Якщо ж властивість **email** дорівнює **null**, то функція **let** не викликається.

Якщо лямбда-вираз викликає лише одну функцію, в яку передається параметр **it**, то можна скоротити виклик, а саме вказати після оператора **::** назву функції, наприклад:

```
1   fun main() {
2
3       val stepan                                          =
    Person( "Stepan" , "stepan@gmail.com" )
4       stepan.email?.let(::println) // stepan@gmail.com
5
6       val taras = Person( "Taras" , "taras@gmail.com" )
7       taras.email?.let(::printEmail) //            Email:
    taras@gmail.com

8   }
9
10  fun printEmail(email: String){
11      println( "Email: $email" )
12  }
13  data class Person(val name:     String,     var email:
14  String?)
```

6.5.1.2. Функція with

Лямбда-вираз у функції **with** як параметр **this** отримує об'єкт, для якого викликається функція. Результатом функції **with**, який повертається, є результат даного лямбда-виразу, наприклад:

```
1   inline fun <T, R> with(receiver: T, блок: T.() ->
    R): R
```

Функція **with** приймає об'єкт, для якого треба виконати блок коду як параметр. Далі в самому блоці коду можна звертатися до загальнодоступних властивостей та методів об'єкта без його імені.

Зазвичай функція **with()** застосовується, коли треба виконати над об'єктом набір операцій як єдине ціле, наприклад:

```
1   fun main() {
2
3       val taras = Person( "Taras" , null )
4       val emailOfTaras = with(taras) {
5           if (email== null )
6               email = "${name.lowercase()}@gmail.com"
7           email
8       }
9       println(emailOfTaras) // taras@gmail.com
10  }
11  data class Person( val name:        String, var email:
    String?)
```

В наведеному прикладі функція **with** отримує об'єкт **taras**, перевіряє його властивість **email**, і якщо вона дорівнює **null**, то встановлює його на основі його імені. Як результат функції повертається значення властивості **email**.

6.5.1.3. Функція run

Лямбда-вираз у функції **run** приймає в якості параметра **this** об’єкт, для якого викликається функція. Результатом функції **run**, що повертається, є результат даного лямбда-виразу, наприклад:

```
1   inline fun <T, R> T.run(block: T.() -> R): R
```

Функція **run** схожа на функцію **with** за тим винятком, що **run** реалізована як функція розширення. Функція **run** також приймає об'єкт, для якого треба виконати блок коду в якості параметру. Далі в самому блоці коду можна звертатися до загальнодоступних властивостей та методів об'єкта без його імені, наприклад:

```
1   fun main() {
2
3       val taras = Person( "Taras" , null )
4       val emailOfTaras = taras.run {
5           if (email== null )
6               email = "${name.lowercase()}@gmail.com"
7           email
8       }
9       println(emailOfTaras) // taras@gmail.com
10  }
11  data class Person( val name:        String, var email:
```

```
String?)
```

В наведеному прикладі функція **run** виконує дії, аналогічні прикладу з функцією **with**, який наводився раніше.

Реалізація **run** як функції розширення полегшує перевірку на **null**, наприклад:

```
1   fun main() {
2
3       val taras = Person( "Taras" , null )
4       val validationResult      =      taras.email?.run
    { "valid" } ?: "undefined"
5       println(emailOfTaras) // undefined
6   }
7   data class Person( val name:       String, var email:
    String?)
```

Також є інший різновид функції **run()**, який просто дозволяє виконати деякий лямбда-вираз, наприклад:

```
1   fun main() {
2
3       val randomText = run{ "hello world" }
4       println(randomText) // hello world
5
6       run{    println( "run    function" )} //    run
    function
7   }
```

6.5.1.4. Функція apply

Лямбда-вираз у функції **apply** в якості параметра **this** отримує об'єкт, для якого викликається функція. Результатом функції, що повертається, фактично є об'єкт, який передається у функцію, для якого і виконується функція, а саме:

```
1   inline fun T.apply(block: T.() -> Unit): T
```

Розглянемо наступний приклад:

```
1   fun main() {
2
3       val taras = Person( "Taras" , null )
4       taras.apply {
5           if (email== null )
6               email = "${name.lowercase()}@gmail.com"
7       }
8       println(taras.email) // taras@gmail.com
9   }
10  data class Person( val name:       String, var email:
    String?)
```

У наведеному прикладі, як і в прикладах з функціями **with** та **run**, перевіряється значення властивості **email**, і якщо вона дорівнює **null**, то встановлюємо її, використовуючи властивість **name**.

Поширеним сценарієм застосування функції **apply()** є побудова об'єкта у вигляді одного зі способів реалізації патерну «Будівельник», наприклад:

```
1   fun main() {
2
3       val volodymyr = Employee()
4       volodymyr.name( "Volodymyr" )
5       volodymyr.age(26)
6       volodymyr.company( "JetBrains" )
7       println( "${volodymyr.name}  (${volodymyr.age})  -
    ${volodymyr.company}" ) // Volodymyr (26) - JetBrains
8   }
9
10  data class Employee( var name:  String  = "" , var age:
    Int = 0, var company: String = "") {
11      fun age(_age: Int): Employee = apply { age = _age
    }
12      fun name(_name: String): Employee = apply { name =
    _name }
13      fun company(_company: String): Employee = apply {
    company = _company }
14  }
```

В наведеному прикладі клас **Employee** містить три методи, які встановлюють кожну з властивостей класу. Причому кожен подібний метод викликає функцію **apply()**, яка передає значення відповідної властивості та повертає поточний об'єкт **Employee**.

6.5.1.5. Функція also

Лямбда-вираз у функції **also** як параметр **it** отримує об'єкт, для якого викликається функція. Результатом функції, що повертається, фактично є об'єкт що передається в функцію, і для якого виконується функція, а саме:

```
1   inline fun T.also(block: (T) -> Unit): T
```

Ця функція аналогічна функції **apply**, за винятком того, що всередині **apply** об'єкт, над яким виконується блок коду, доступний через параметр **it**, наприклад:

```
1   fun main() {
2
3       val taras = Person( "Taras" , null )
4       taras.also {
5           if (it.email== null )
6               it.email
    = "${it.name.lowercase()}@gmail.com"
```

```
7       }
8       println(taras.email) // taras@gmail.com
9   }
10  data class Person( val name:       String, var email:
    String?)
```

6.6. Інфіксна нотація

Інфіксна нотація полягає у розташування оператора чи функції перед операндами чи аргументами. Для визначення інфіксної функції необхідно перед її визначенням вказати ключове слово **infix**, а саме:

```
1   infix fun назва_функції (параметр:    тип_параметра):
    тип_значення_що_повертається {
2       // дії функції
3   }
```

Інфіксна функція має приймати лише один параметр. При цьому параметр не повинен мати значення за замовчуванням та не повинен представляти невизначений набір значень.

Існують два способи визначення інфіксної функції – або всередині класу, або як функції розширення.

Розглянемо використання інфіксної функції всередині класу:

```
1   fun main() {
2
3       val acc = Account(1000)
4       acc put 150
5       // рівноцінно виклику
6       acc.put(150)
7       acc.printSum() // 1300
8   }
9   class Account( var sum: Int) {
10
11      infix fun put(amount: Int){
12          sum = sum + amount
13      }
14      fun printSum() = println(sum)
15  }
```

В цьому прикладі клас **Account** – це клас банківського рахунку, який через конструктор отримує початкову суму на рахунку. За допомогою інфіксної функції **put()** визначається додавання на рахунок суми, переданої через параметр функції.

Виклик функції виглядає наступним чином:

```
1   acc put 150
```

Перший параметр, в цьому прикладі змінна **acc**, позначає об'єкт, який викликає функцію. А другий параметр – це дані, які безпосередньо

передаватимуться до інфіксної функції через її параметр. Тобто цей виклик фактично аналогічний наступному виклику:

```
acc.put(150)
```

Також інфіксна функція може визначатися як функція розширення. Наприклад, перепишемо вище використану функцію **put()** як функцію розширення:

```
fun main() {

    val acc = Account(1000)
    acc put 150
    acc.put(150)
    acc.printSum() // 1300
}
infix fun Account.put(amount: Int){
    this .sum = this .sum + amount
}
class Account(sum: Int) {
    fun printSum() = println(sum)
}
```

Варто відзначити, що функція розширення на відміну від функції всередині класу має доступ тільки до тих властивостей, які є публічними, тобто **public**.

Та все ж таки, використання функцій розширення дозволяє додати інфіксні функції до вже існуючих типів. Наприклад, визначимо інфіксну функцію для підрахунку кількості появи деякого символу в певному рядку:

```
fun main() {

    val hello = " hello world "
    val lCount = hello wordCount 'l'
    val oCount = hello wordCount 'o'
    println(lCount) // 3
    println(oCount) // 2
}

infix fun String.wordCount(c: Char) : Int{
    var count = 0
    for (n in this ){
        if (n == c) count++
    }
    return count
}
```

В цьому прикладі функція **wordCount** проходить по всіх символах рядка та підраховує, скільки разів зустрічається деякий символ, який передається через

параметр функції. Отриманий результат повертається цією функцією. Потім можна застосувати інфіксну нотацію, а саме:

```
1   val lCount = hello wordCount 'l'
```

Оскільки функція повертає результат типу **Int**, то можна передати цей результат в якусь змінну.

Резюме

В наведеному розділі розглядаються додаткові можливості об'єктно-орієнтованого програмування (ООП) у Kotlin. Детально описано обробку винятків за допомогою конструкції **try..catch..finally**, оператор **throw** для створення власних винятків та можливість повернення значень із t**r**y. Також розглянуто nullable-типи, оператори **?:**, **?.** і **!!** для роботи з **null**. Висвітлено делеговані властивості, які дозволяють передавати їхню реалізацію іншим класам. Розглянуто перетворення типів, **smart cast (is)**, явні перетворення (**as**) та їхні обмеження. Описані функції розширення, scope-функції (**let**, **run**, **with**, **apply**, **also**) і їхнє застосування. Наостанок пояснено інфіксну нотацію, що дозволяє спрощувати виклик функцій.

Ознайомлення із цим розділом дозволить отримати знання про додаткові можливості об’єктно-орієнтованого програмування, а саме: обробку винятків; Null і nullable-типах; делегованих властивостях; перетворенні типів; функціях розширення, що є передумовою для вивчення матеріалу, викладеного в наступних розділах, а також без чого не можливо створення мобільних, десктопних та веб-застосунків мовою Kotlin.

Контрольні запитання і завдання

1. Що таке виняток у програмуванні?
2. Які блоки містить конструкція **try..catch..finally** у Kotlin?
3. Чи можна використовувати **try** без **catch** або **finally**?
4. Як у Kotlin згенерувати виняток вручну?
5. Які властивості має клас Exception для отримання інформації про виняток?
6. Як обробити кілька винятків у Kotlin?
7. Як у Kotlin змінну зробити **nullable**?
8. Для чого використовується оператор **?:**?
9. Чим оператор **!!** відрізняється від **?.**?
10. Що станеться, якщо використати **!!** для змінної зі значенням **null**?
11. Що таке делеговані властивості у Kotlin?

12. Які функції повинен реалізовувати делегат для val-властивості?
13. Чим відрізняються делеговані val- і var-властивості?
14. Які функції використовуються для перетворення числових типів у Kotlin?
15. Як працює оператор **is** і що таке **smart cast**?
16. Чому **smart cast** не працює для var-властивостей?
17. Як працює оператор **as** і чим він відрізняється від **as?**?
18. Що таке функція розширення та як її визначити?
19. Які є scope-функції у Kotlin та чим вони відрізняються?
20. Чим функція **apply** відрізняється від **also**?
21. Напишіть програму, яка зчитує число від користувача та ділить 100 на це число. Обробіть можливий виняток при введенні нуля.
22. Модифікуйте попередню програму, щоб у разі винятку виводилося повідомлення про помилку, а в блоці **finally** виводився текст «Операція завершена».
23. Створіть змінну nullable-типу **String**?, присвойте їй значення **null**, а потім використайте оператор **?:**, щоб у разі **null** виводився рядок «Немає значення».
24. Напишіть функцію, яка приймає nullable-рядок і повертає його довжину. Використайте оператор **?.**, щоб уникнути помилки при **null**.
25. Створіть клас **User**, у якому властивість **name** буде делегована іншому класу, що логуватиме доступ до **name**.
26. Реалізуйте делеговану властивість **age**, яка не дозволяє встановлювати від'ємні значення.
27. Напишіть програму, яка отримує число у вигляді рядка **(String)**, конвертує його в **Int** і обробляє виняток, якщо конвертація неможлива.
28. Реалізуйте функцію **checkType(obj: Any)**, яка виводить, чи є **obj** числом (**Int**), рядком (**String**) або іншим типом, використовуючи **is**.
29. Створіть функцію розширення для типу **Int**, яка повертає квадрат числа.
30. Використайте **apply** або **also**, щоб ініціалізувати об'єкт класу **Person**, задаючи ім'я та вік у його властивості.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Який оператор у Kotlin використовується для обробки винятків?
а) error

b) try..catch..finally
c) throw..catch..error
d) handle..catch..finally
Правильна відповідь: b).

Що означає !! у Kotlin?
a) Привласнює змінній значення за замовчуванням
b) Викидає виняток, якщо змінна дорівнює null
c) Ігнорує винятки
d) Викликає функцію без перевірки
Правильна відповідь: b).

Який оператор використовується для перевірки типу змінної?
a) as?
b) is
c) typeof
d) cast
Правильна відповідь: b).

Закриті запитання з кількома правильними відповідями

Які блоки можуть міститися в конструкції try?
a) catch
b) finally
c) recover
d) throw
Правильна відповідь: a), b).

Які з наступних операторів дозволяють уникати NullPointerException?
a) ?.
b) ?:
c)!!
d) as
Правильна відповідь: a), b).

Які функції розширення є в Kotlin?
a) apply
b) let
c) execute
d) run

Правильна відповідь: a), b), d).

Завдання на відповідність

Співвіднесіть оператори та їх функції:
a) !! → 1. Викидає виняток при null
b) ?. → 2. Перевіряє null перед доступом до властивості
c) ?: → 3. Повертає значення за замовчуванням при null
d) as? →4. Безпечне приведення типів
Правильна відповідь:
a) → 1.
b) → 2.
c) → 3.
d) → 4.

Вкажіть правильний порядок обробки винятків у try..catch..finally:
a) Виконання блоку try → 1. Крок 1
b) Обробка винятку у catch (якщо він є) → 2. Крок 2
c) Виконання блоку finally → 3. Крок 3
Правильна відповідь:
a) → 1.
b) → 2.
c) → 3.

Вставити пропущені слова

Доповніть код, щоб уникнути NullPointerException:
```
val name: String? = null
val result = name _____________ "No Name"
println(result)
```
Правильна відповідь: ?:

Завдання з кодом

Який результат виведе наступний код?
```
fun main() {
    val num: Int? = null
    val result = num?.times(2) ?: 10
    println(result)
}
```

a) null
b) 10
c) 0
d) Виняток
Правильна відповідь: b).

Що станеться при виконанні цього коду?
```
fun main() {
    val x: Int? = null
    val y: Int = x!!
    println(y)
}
```
a) Виведе null
b) Виведе 0
c) Виведе null без помилки
d) Програма аварійно завершиться
Правильна відповідь: d).

Відкриті запитання

Знайдіть і виправте помилку:
```
fun main() {
    val number: Int = null
    println(number)
}
```
Правильна відповідь:
`val number: Int? = null`

Виправте код, щоб він не викидав виняток:
```
fun main() {
    val name: String? = null
    println(name.length)
}
```
Правильна відповідь:
`println(name?.length)`

7. КОЛЕКЦІЇ ТА ПОСЛІДОВНОСТІ

7.1. Змінювані та незмінювані колекції

Колекції представляють собою контейнери, що використовуються для зберігання даних. Залежно від типу колекції різняться і методи роботи з даними.

Мова Kotlin не має власної бібліотеки колекцій та повністю покладається на класи колекцій, які надає Java. В той же час ці колекції у мові Kotlin розширюються для надання додаткових можливостей.

Так, у мові Kotlin колекції поділяються на змінювані (**mutable**) та незмінювані (**immutable**) колекції.

Mutable-колекція може змінюватися, до неї можна додавати, у ній можна змінювати та видаляти елементи. **Immutable-колекція** також підтримує додавання, заміну та видалення даних, однак у процесі подібних операцій колекція буде знову перестворюватися.

Всі колекції у мові Kotlin розміщуються в пакеті **kotlin.collections,** як показано на рисунку 7.1. Повний список інтерфейсів та класів, якими володіють колекції, можна знайти за посиланням https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html[1] [13].

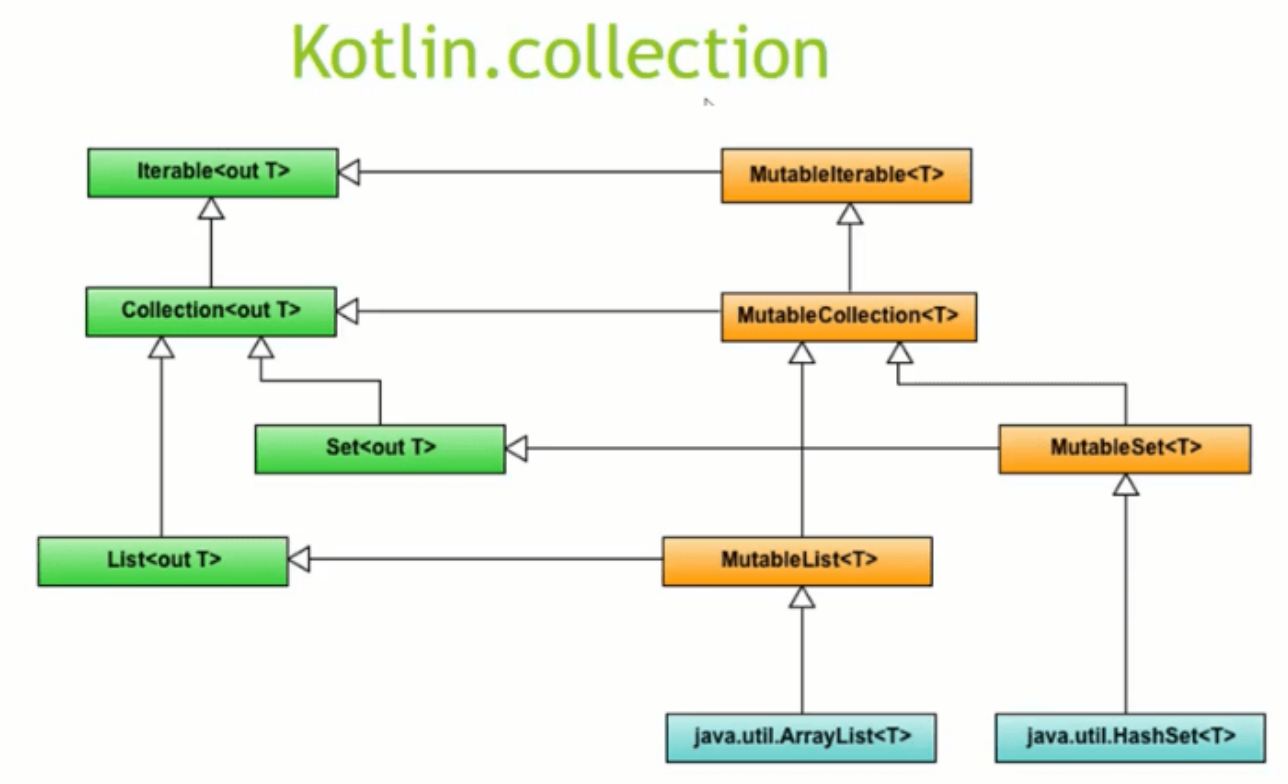

Рисунок 7.1. Повний склад колекцій у Kotlin

[1] https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html

7.1.1. Незмінювані колекції

На вершині ієрархії знаходиться інтерфейс **Iterable**, який визначає функцію ітератор для перебору елементів колекції.

Основним інтерфейсом, що дозволяє працювати з колекціями, є **kotlin.Collection**. Цей інтерфейс визначає функціональність для перебору елементів, перевірки наявності елементів, читання даних. Однак він не надає можливості додавання та видалення даних. Його основними компонентами є:

- **size**: повертає кількість елементів у колекції;
- **isEmpty()**: повертає **true**, якщо колекція порожня;
- **contains(element)**: повертає **true**, якщо колекція містить елемент;
- **containsAll(collection)**: повертає **true**, якщо колекція містить елементи колекції **collection**.

Цей інтерфейс розширюється іншими інтерфейсами, які представляють незмінювані колекції, а саме: **List**, який представляє звичайний список; та **Set**, який представляє невпорядковану колекцію елементів, що не допускає дублювання елементів.

Окреме місце має інтерфейс **Map**. Він не розширює **Collection** і представляє набір пар ключ-значення, де кожен ключ відповідає деякому значенню. При цьому всі ключі у колекції є унікальними.

7.1.2. Змінювані колекції

Всі змінювані колекції реалізують інтерфейс **MutableIterable**. Він є функцією ітератором для перебору елементів колекції.

Для зміни даних у мові Kotlin також визначено інтерфейс **kotlin.MutableCollection**, який розширює інтерфейс **kotlin.Collection** і надає методи видалення та додавання елементів. Зокрема наступні:

- **add(element)**: додає елемент;
- **remove(element)**: видаляє елемент;
- **addAll(elements)**: додає набір елементів;
- **removeAll(elements)**: видаляє набір елементів;
- **clear()**: видаляє всі елементи з колекції.

Цей інтерфейс розширюється інтерфейсами **MutableList**, який представляє список, що змінюється, і **MutableSet**, який представляє змінювану невпорядковану колекцію унікальних елементів.

Ще одна змінювана колекція представлена інтерфейсом **MutableMap** – це змінюваний **Map**, де кожен елемент представляє пару ключ-значення.

7.1.3. Операції з колекціями

Окрім вище розглянутих методів, інтерфейс **Iterable** також надає низку функцій для виконання різних операцій над колекціями.

Розглянемо основні операції:

- **all(predicate: (T) -> Boolean): Boolean**: повертає **true**, якщо всі елементи відповідають предикату, який передається у функцію як параметр;
- **any(): Boolean**: повертає **true**, якщо колекція містить хоча б один елемент;
- **any(predicate: (T) -> Boolean): Boolean**: додаткова версія повертає **true**, якщо хоча б один елемент відповідає предикату, який передається у функцію як параметр;
- **asSequence(): Sequence<T>**: створює з колекції послідовність;
- **average(): Double**: повертає середнє значення для числової колекції типів **Byte**, **Int**, **Short**, **Long**, **Float**, **Double**;
- **chunked(size: Int): List<List<T>>**: розщеплює колекцію на список, що складається з об'єктів **List**, а параметр **size** встановлює максимальну кількість елементів у кожному списку;
- **chunked(size: Int, transform: (List<T>) -> R): List<R>**: додаткова версія як другий параметр отримує функцію перетворення, яка перетворює кожен список на елемент нової колекції;
- **contains(element: T): Boolean**: повертає **true**, якщо колекція містить елемент **element**;
- **count(): Int**: повертає кількість елементів у колекції;
- **count(predicate: (T) -> Boolean): Int**: додаткова версія повертає кількість елементів, які відповідають предикату;
- **distinct(): List<T>**: повертає нову колекцію, яка містить лише унікальні елементи;
- **distinctBy(selector: (T) -> K): List<T>**: повертає нову колекцію, яка містить лише унікальні елементи з урахуванням функції селектора, яка передається як параметр;
- **drop(n: Int): List<T>**: повертає нову колекцію, яка містить усі елементи за винятком перших **n** елементів;
- **dropWhile(predicate: (T) -> Boolean): List<T>**: повертає нову колекцію, яка містить усі елементи за винятком перших елементів, що відповідають предикату;

- **elementAt(index: Int): T**: повертає елемент за індексом **index**. Якщо індекс виходить за межі колекції, то генерується виняток типу **IndexOutOfBoundsException**;
- **elementAtOrElse(index: Int, defaultValue: (Int) -> T): T**: повертає елемент за індексом **index**. Якщо індекс виходить за межі колекції, то повертається значення, яке встановлюється функцією з параметра **defaultValue**;
- **elementAtOrNull(index: Int): T?**: повертає елемент за індексом **index**. Якщо індекс виходить за межі колекції, повертається **null**;
- **filter(predicate: (T) -> Boolean): List<T>**: повертає нову колекцію з елементів, що відповідають предикату;
- **filterNot(predicate: (T) -> Boolean): List<T>**: повертає нову колекцію з елементів, які не відповідають предикату;
- **filterNotNull(): List<T>**: повертає нову колекцію з елементів, які не дорівнюють **null**;
- **find(predicate: (T) -> Boolean): T?**: повертає перший елемент, що відповідає предикату. Якщо елемент не знайдено, то повертається **null**;
- **findLast(predicate: (T) -> Boolean): T?**: повертає останній елемент, що відповідає предикату. Якщо елемент не знайдено, то повертається **null**;
- **first(): T**: повертає перший елемент колекції;
- **first(predicate: (T) -> Boolean): T**: додаткова версія повертає перший елемент, який відповідає предикату. Якщо елемент не знайдено, то генерується виняток типу **NoSuchElementException**;
- **firstOrNull(): T?**: повертає перший елемент колекції;
- **firstOrNull(predicate: (T) -> Boolean): T?**: додаткова версія повертає перший елемент, який відповідає предикату. Якщо елемент не знайдено, то повертається **null**;
- **flatMap(transform: (T) -> List<R>): List<R>**: перетворює колекцію елементів типу **T** на колекцію елементів типу **R**, використовуючи функцію перетворення, яка передається як параметр;
- **fold(initial: R, operation: (acc: R, T) -> R): R**: повертає значення, яке є результатом дії функції **operation** над кожним елементом колекції. Перший параметр функції **operation** є результатом роботи функції над попереднім елементом колекції (при першому виклику – значення з параметра **initial**), а другий параметр – поточний елемент колекції;
- **forEach(action: (T) -> Unit)**: виконує для кожного елемента колекції операцію **action**;

- **groupBy(keySelector: (T) -> K): Map<K, List<T>>**: групує елементи ключа, який повертається функцією **keySelector**. Результатом функції є колекція **Map**, де ключ є, власне, ключем елементів, а значення – це список **List** з елементів, які відповідають цьому ключу;
- **groupBy(keySelector: (T) -> K, значенняTransform: (T) -> V): Map<K, List<V>>**: додаткова версія, яка приймає функцію перетворення елементів;
- **indexOf(element: T): Int**: повертає індекс першого входження елемента **element**. Якщо елемент не знайдено, повертається **-1**;
- **indexOfFirst(predicate: (T) -> Boolean): Int**: повертає індекс першого елемента, що відповідає предикату. Якщо елемент не знайдено, повертається **-1**;
- **indexOfLast(predicate: (T) -> Boolean): Int**: повертає індекс останнього елемента, що відповідає предикату. Якщо елемент не знайдено, повертається **-1**;
- **intersect(other: Iterable): Set**: повертає всі елементи поточної колекції, які є у колекції **other**;
- **joinToString(): String**: генерує з колекції рядок;
- **last(): T**: повертає останній елемент колекції;
- **last(predicate: (T) -> Boolean): T**: додаткова версія, яка повертає останній елемент, який відповідає предикату. Якщо елемент не знайдено, то генерується виняток типу **NoSuchElementException**;
- **lastOrNull(): T?**: повертає останній елемент колекції;
- **lastOrNull(predicate: (T) -> Boolean): T?**: додаткова версія, яка повертає останній елемент, який відповідає предикату. Якщо елемент не знайдено, то повертається **null**;
- **lastIndexOf(element: T): Int**: провертає останній індекс елемента **element**. Якщо елемент не знайдено, повертається **-1**;
- **map(transform: (T) -> R): List<R>**: застосовує до елементів колекції функцію трансформації та повертає нову колекцію, що складається з нових елементів;
- **mapIndexed(transform: (index: Int, T) -> R): List<R>**: застосовує до елементів колекції та їх індексів функцію трансформації та повертає нову колекцію, що складається з нових елементів;
- **mapNotNull(transform: (T) -> R?): List<R>**: застосовує до елементів колекції функцію трансформації та повертає нову колекцію, що складається з нових елементів, які не дорівнюють **null**;

- **maxOf(selector: (T) -> Double): Double**: повертає максимальне значення на основі селектора;
- **maxOfOrNull(selector: (T) -> Double): Double?**: повертає максимальне значення з урахуванням селектора. Якщо колекція порожня, повертається **null**;
- **maxOrNull(): Double?**: повертає максимальне значення. Якщо колекція порожня, повертається **null**;
- **minOf(selector: (T) -> Double): Double**: повертає мінімальне значення на основі селектора;
- **minOfOrNull(selector: (T) -> Double): Double?**: повертає мінімальне значення з урахуванням селектора. Якщо колекція порожня, повертається **null**;
- **minOrNull(): Double?**: повертає мінімальне значення. Якщо колекція порожня, повертається **null**;
- **minus(element: T): List<T>**: повертає нову колекцію, яка містить всі елементи поточної колекції за винятком елемента **element**. Має різновиди, які дозволяють виключити з колекції набори елементів, наприклад:

```
1  minus(elements: Array<T>): List<T>
2  minus(elements: Iterable<T>): List<T>
3  minus(elements: Sequence<T>): List<T>
```

- **plus(element: T): List<T>:** повертає нову колекцію, яка містить всі елементи поточної за винятком плюс елемент element. Має різновиди, які дозволяють включити до колекції набори елементів, наприклад:

```
1  plus(elements: Array<T>): List<T>
2  plus(elements: Iterable<T>): List<T>
3  plus(elements: Sequence<T>): List<T>
```

- **reduce(operation: (acc: S, T) -> S): S**: повертає значення, яке є результатом дії функції **operation** над кожним елементом колекції. Перший параметр функції **operation** – це результат роботи функції над попереднім елементом колекції, а другий параметр – це поточний елемент колекції;
- **shuffled(): List<T>**: умовно перемішує колекцію;
- **sorted(): List<T>**: впорядковує колекцію за зростанням;
- **sortedBy(selector: (T) -> R?): List<T>**: впорядковує колекцію за зростанням на основі селектора;
- **sortedByDescending(selector: (T) -> R?): List<T>**: впорядковує колекцію за спаданням на основі функції-селектора;

- **sortedDescending(): List<T>**: впорядковує колекцію за спаданням;
- **sum(): Int**: повертає суму елементів колекції;
- **subtract(other: Iterable): Set**: повертає набір елементів, які є в поточній колекції та відсутні в іншій колекції;
- **sum(): Int**: повертає суму елементів колекції;
- **sumOf(selector: (T) -> Int): Int**: повертає суму елементів колекції на основі функції-селектора;
- **take(n: Int): List<T>**: повертає нову колекцію, яка містить **n** перших елементів поточної колекції;
- **takeWhile(predicate: (T) -> Boolean): List<T>**: повертає нову колекцію, яка містить **n** перших елементів поточної колекції, що відповідають функції-предикату;
- **toHashSet(): HashSet<T>**: створює з колекції об'єкт **HashSet**;
- **toList(): List<T>**: створює з колекції об'єкт **List**;
- **toMap(): Map<K, V>**: створює з колекції об'єкт **Map**;
- **toSet(): Set<T>**: створює з колекції об'єкт **Set**;
- **union(other: Iterable): Set**: повертає набір унікальних елементів, які є у поточній колекції та колекції **other**.

7.2. Колекція List

List представляє послідовний список елементів. При цьому **List** представляє незмінювану (**immutable**) колекцію, яка лише забезпечує отримання елементів за позицією.

Інтерфейс **List** розширює інтерфейс **Collection**, тому володіє всіма його можливостями.

Для створення об'єкта **List** застосовується метод **listOf()**, наприклад:

```
1   var numbers = listOf(1, 2, 3, 4, 5, null )
2   var numbers2: List<Int> = listOf(5, 6, 7)
```

Тип **List** дозволяє отримати дані за допомогою різних методів. Основні з них:

- **get(index)**: повертає елемент за індексом;
- **elementAt(index)**: повертає елемент за індексом, Якщо індекс виходить за межі колекції, то генерується виняток типу **IndexOutOfBoundsException**;
- **elementAtOrNull(index)**: повертає елемент за індексом, якщо такого елемента не виявиться, повертає **null;**
- **first()**: повертає перший елемент;

- **last()**: повертає останній елемент;
- **indexOf(element)**: повертає перший індекс елемента;
- **lastIndexOf(element)**: повертає останній індекс елемента;
- **contains(element)**: повертає **true**, якщо елемент є у списку.

Приклади використання колекцій **List** наведені нище:

```
fun main() {
    val numbers : List<Int> = listOf(1, 2, 3, 4, 5)

    // Перебір списку
    for (n in numbers){
        print(n)
    }
    println()

    println(numbers.get(1)) // 2
    println(numbers.indexOf(2)) // 1
    println(numbers.lastIndexOf(3)) // 2
    println(numbers.first()) // 1
    println(numbers.last()) // 5
    println(numbers.size) //5
    println(numbers.contains(4)) // true
    println(numbers.elementAt(1)) // 2
    println(numbers.elementAtOrNull(9)) // null
}
```

7.2.1. Змінювані списки

Змінювані списки представлені інтерфейсом **MutableList**. Він розширює інтерфейс **List** і дозволяє додавати та видаляти елементи. Цей інтерфейс реалізується класом **ArrayList**.

Для створення списків, що змінюються, можна використовувати низку методів, а саме:

- **arrayListOf()**: створює об'єкт **ArrayList**;
- **mutableListOf()**: створює об'єкт **MutableList**.

Наведемо кілька прикладів створення змінюваних списків:

```
var numbers : ArrayList<Int> = arrayListOf(1, 2, 3, 4, 5)
var numbers2: MutableList<Int> = mutableListOf(5, 6, 7)
```

Для додавання або видалення елемента необхідно використовувати наступні методи **MutableList**:

- **add(index, element)**: додає елемент за індексом;
- **add(element)**: додає елемент;

- **addAll(collection)**: додає колекцію елементів;
- **remove(element)**: видаляє елемент;
- **removeAt(index)**: видаляє елемент за індексом;
- **clear()**: видаляє всі елементи колекції.

Наведемо кілька прикладів:

```
fun main() {

    val numbers1 : ArrayList<Int> = arrayListOf(1, 2, 3, 4, 5)
    numbers1.add(4)
    numbers1.clear()

    val numbers2: MutableList<Int> = mutableListOf(5, 6, 7)

    numbers2.add(12)
    numbers2.add(0, 23)
    numbers2.addAll(0, listOf(-3, -2, -1))
    numbers2.removeAt(0)
    numbers2.remove(5)

    for (n in numbers2){ println(n) }
}
```

7.3. Колекція Set

Інтерфейс **Set** представляє невпорядкований набір елементів, що зберігає лише унікальні об'єкти. Інтерфейс **Set** представляє незмінюваний (**immutable**) набір. **Set** розширює інтерфейс **Collection**.

Наведемо деякі методи інтерфейсу **Set**:

- **contains(element)**: повертає **true**, якщо елемент міститься у наборі;
- **isEmpty()**: повертає **true**, якщо набір порожній;
- **minus(element)**: повертає новий набір, де відсутній **element**. При цьому вихідний набір не змінюється;
- **plus(element)**: повертає новий набір, до якого доданий новий **element**. Вихідний набір також не змінюється.

Для створення незмінюваного (**immutable**) набору використовується функція **setOf()**, наприклад:

```
var numbers: Set<Int> = setOf(5, 6, 7)
```

Розглянемо приклад використання **Set**:

```
fun main() {
    val items = setOf(1, 2, 3, 4, 5)
```

```
3       println(items.size) // 5
4       println(items.contains(4)) // true
5       println(items.isEmpty()) // false
6       println(items.minus(3)) // [1, 2, 4, 5]
7       println(items.plus(7)) // [1, 2, 3, 4, 5, 7]
8
9       for (n in items) { print (n) } // 1 2 3 4 5
10  }
```

7.3.1. Змінювання колекції

Змінювані (**mutable**) набори представлені інтерфейсом **MutableSet**, який розширює інтерфейси **Set** та **MutableCollection**.

Для створення змінюваних (**mutable**) наборів використовується функція **mutableSetOf()**, а саме:

```
1   val numbers: MutableSet<Int> = mutableSetOf(35, 36,
    37)
```

Інтерфейс **MutableSet** реалізується наступними типами змінюваних наборів:

- **LinkedHashSet**: об'єднує можливості хеш-таблиці та зв'язаного списку. Створюється за допомогою функції **linkedSetOf()**;
- **HashSet**: представляє хеш-таблицю. Створюється за допомогою функції **hashSetOf()**.

Наведемо кілька прикладів:

```
1   val numbers1: HashSet<Int> = hashSetOf(5, 6, 7)
2   val numbers2:  LinkedHashSet<Int>  =  linkedSetOf(25,
    26, 27)
3   val numbers3: MutableSet<Int> = mutableSetOf(35, 36,
    37)
```

Зміна набору за допомогою **MutableSet** здійснюється наступним чином:

```
1   val numbers:  MutableSet<Int>  =  mutableSetOf(35,  36,
    37)
2   println(numbers.add(2))
3   println(numbers.addAll(setOf(4, 5, 6)))
4   println(numbers.remove(36))
5
6   for (n in numbers){ println(n) } // 35 37 2 4 5 6
7   numbers.clear()
```

7.4. Колекція Map

Колекція **Map** представляє колекцію об'єктів, де кожен елемент має ключ і зіставне з ним значення. При цьому всі ключі колекції є унікальними. На відміну від **List** та **Set** інтерфейс **Map** не розширює інтерфейс **Collection**.

Map представляє незмінювану колекцію, для створення якої застосовується метод **mapOf()**, а саме:

```
1  val countries: Map<String, Int> = mapOf( "США" to
   300, "Франція " to 60, "Німеччина" to 81)
2
3  println(countries[ "США" ]) // 300
4  for(country in countries){
5      println( "${country.key} - ${country.value}" )
6  }
7  println(countries)
```

Функція **mapOf** приймає набір елементів, кожен з яких за допомогою оператора **to** зіставляє ключ з відповідним значенням, наприклад, **"USA" to 300**.

Результа роботи програми наведено на рисунку 7.2.

```
США - 300
Франція - 60
Німеччина - 81
{США=300, Франція=60, Німеччина=81}
```

Рисунок 7.2. Результат роботи програми

Колекції, що змінюються, представлені інтерфейсом **MutableMap**, який розширює інтерфейс **Map**. Для створення об'єкта **MutableMap** використовується функція **mutableMapOf()**.

Інтерфейс **MutableMap** реалізується низкою колекцій, а саме:

- **HashMap**: найпростіша реалізація інтерфейсу **MutableMap**, яка не гарантує порядок елементів у колекції, та створюється за допомогою функції **hashMapOf()**;
- **LinkedHashMap**: представляє комбінацію **HashMap** та пов'язаного списку, та створюється за допомогою функції **linkedMapOf()**.

Розглянемо наступний приклад використання змінюваних колекцій:

```
1  val countries: MutableMap<String, Int> =
   mutableMapOf( "США" to 300, "Франція" to 60, "
   Німеччина" to 81)
2
   countries.put( "Іспанія" , 33) // додаємо новий
3  елемент з ключем "Іспанія" та значенням 33
```

```
countries.remove( "Франція" ) // видаляємо елемент із ключем "Франція"

for (country in countries) {
    println( "${country.key} - ${country.value}" )
}
println(countries)


val map1: LinkedHashMap<Int, String> = linkedMapOf(1 to "1" , 2 to "2" )
val map2: HashMap<Int, String> = hashMapOf(1 to "1" , 2 to "2" )
```

Результа роботи програми наведено на рисунку 7.3.

```
США - 300
Німеччина - 81
Іспанія - 33
{США =300, Німеччина=81, Іспанія=33}
```

Рисунок 7.3. Результат роботи програми

7.5. Послідовності

Поряд із колекціями мова Kotlin надає ще один тип наборів елементів – послідовності (**sequences**). Послідовності надають схожу функціональність, що і інтерфейс **Iterable**, який реалізується типами колекцій. Ключова різниця полягає у тому, як обробляються елементи послідовності при застосуванні до них набору операцій.

7.5.1. Створення послідовності

7.5.1.1 Функція sequenceOf

Послідовності представляють інтерфейс **Sequence<T>**. Для створення об'єкта цього типу можна використовувати вбудовану функцію **sequenceOf()**. Як параметр вона приймає набір елементів, які входитимуть у послідовність, наприклад:

```
val people = sequenceOf("Taras", "Stepan", "Volodymyr")    //тип Sequence<String>
println(people.joinToString())  // Taras, Stepan, Volodymyr
```

У цьому прикладі визначається послідовність типу **Sequence<String>**, яка містить три елементи. Для виведення послідовності на консоль застосовується її перетворення на рядок за допомогою функції **joinToString()**.

7.5.1.2. Метод колекцій asSequence

Також можна створити послідовність з об'єкта **Iterable**, наприклад з об'єктів типу **List** або **Set**, використовуючи метод **asSequence()**, а саме:

```
1   val employees = listOf("Taras", "Stepan",
    "Volodymyr") // об’єкт List<String>
2   val people = employees.asSequence()          //тип
    Sequence<String>
3   println(people.joinToString())    // Taras, Stepan,
    Volodymyr
```

7.5.1.3. Функція generateSequence

Третій спосіб створення послідовності надає функція **generateSequence**, наприклад:

```
1   var number = 0
2   val numbers = generateSequence{ number += 2; number}
3   println(numbers.take(5).joinToString())    //
    отримуємо перші 5 елементів послідовності - 2, 4, 6,
    8, 10
```

Як параметр функція **generateSequence()** приймає функцію, яка повертає деяке значення. Це значення потім стане елементом послідовності. У цьому прикладі збільшується значення змінної **number** на 2 і повертається її значення. Тобто послідовність буде містити числа з кроком у 2, а саме: 2, 4, 6, 8, 10, 12.

Варто враховувати, що ця стандартна функція генерує нескінченну послідовність. Тому в прикладі, наведеному вище, під час виведення елементів на консоль за допомогою методу **take()** отримуємо тільки перші п'ять елементів. Щоб обмежити послідовність, що генерується, функція у **generateSequence()** повинна повертати **null**, наприклад:

```
1   var number = 0
2   val numbers = generateSequence{ number += 2; if(number
    > 8) null else number}
3   println(numbers.joinToString())    // 2, 4, 6, 8
```

В цьому прикладі послідовність містить лише числа від 2 до 8 включно.

Функція **generateSequence()** має кілька варіацій. Зокрема, як перший параметр можна передати перше значення за замовчуванням, а другий параметр як і раніше може представляти функцію, яка генерує послідовність, наприклад:

```
1   val numbers = generateSequence(5){ if(it == 25) null
    else it + 5}
```

```
println(numbers.joinToString())     // 5, 10, 15, 20, 25
```

У наведеному прикладі у функцію для першого параметру передається число 5, тобто це буде перший елемент послідовності. Функція створення послідовності через параметр **it** отримує попередній результат функції, фактично попередній елемент послідовності (при першому виклику функції це значення першого параметра). В даному випадку створюється кінцева послідовність, де кожен наступний елемент більший за попередній на 5. Якщо останній елемент дорівнює 25, то повертається **null**, тим самим завершується генерація послідовності. Варто зазначити, навіть якщо функція поверне **null**, то послідовність міститиме один елемент, а саме **generateSequence(5){ null }**.

7.5.1.4. Функція sequence

Розглянемо четвертий спосіб із застосуванням функції **sequence()**. У цій функції можна генерувати елементи послідовності за допомогою функцій **yield()** та **yieldAll()**, наприклад:

```
val numbers = sequence {
    yield(1)
    yield(4)
    yield(7)
}
println(numbers.joinToString())     // 1, 4, 7
```

Функція **yield()** фактично повертає назовні деяке значення, яке їй передається через параметр. Тобто, при першому зверненні до функції **sequence** спрацює виклик **yield(1)**, який поверне значення 1. При другому зверненні спрацює виклик **yield(4)**, який поверне 4. І при третьому зверненні спрацює виклик **yield(7)**. Таким чином, послідовність міститиме 3 елементи: 1, 4, 7.

Можна створити і нескінченну послідовність, наприклад:

```
val numbers = sequence {
    var start = 0
    while(true) yield(start++)
}
println(numbers.take(5).joinToString())     // 0, 1, 2, 3, 4
```

В цьому прикладі при кожному зверненні до функції **sequence** повертається одне з чисел, починаючи з 0.

Якщо потрібно створити послідовності на основі іншої послідовності або колекції, то зручніше джерело даних передати у функцію **yieldAll()**, наприклад:

```
val personal = sequence {
    val data = listOf("Olena", "Kateryna", "Ivanna")
    yieldAll(data)
```

```
4   }
5   println(personal.joinToString())     // Olena, Kateryna,
    Ivanna
```

7.5.2. Перебір послідовності

Для перебору послідовності можна використовувати стандартний цикл **for**, наприклад:

```
1   val people = sequenceOf("Taras", "Stepan",
    "Volodymyr")
2   for(person in people) println(person)
```

7.5.3. Операції із послідовностями

Інтерфейс **Sequence** надає низку функцій, які дозволяють виконувати різні операції з послідовностями. Усі операції з послідовностями поділяються на декілька типів.

Насамперед операції відрізняються за наявністю стану. Одні операції можуть зберігати стан (**statefull-операції**), наприклад, операція **distinct()**. Подібний стан зазвичай пропорційний кількості елементів у послідовності. А є операції, які не мають стану (**stateless-операції**), наприклад, функції **map()** та **filter()**, або вимагають дуже невеликого константного стану, як функції **take()** та **drop()**. Подібні операції обробляють кожен елемент незалежно від інших.

Інша класифікація операцій полягає в тому, коли саме виконується операції. У такому разі вони бувають проміжними (**intermediate**). Такі операції зазвичай повертають іншу послідовність. Причому вони не виконуються відразу під час їхнього виклику.

А є й термінальні чи кінцеві операції (**terminal**). Такі операції передбачають вибір елементів послідовності, наприклад, коли необхідно отримати кількість елементів за допомогою функції **count()** або елемент за індексом. Такі операції виконуються відразу під час їхнього виклику. Те саме можна сказати про перебір за допомогою циклу **for**, який витягує елементи з послідовності і відповідно також запускає процес виконання всіх операцій послідовності.

Розглянемо основні операції:

- **all(predicate: (T) -> Boolean): Boolean**: термінальна операція, яка повертає **true**, якщо всі елементи відповідають предикату, який передається у функцію як параметр;
- **any(): Boolean:** термінальна операція, яка повертає **true**, якщо послідовність містить хоча б один елемент;

- **any(predicate: (T) -> Boolean): Boolean**: додаткова версія, яка повертає **true**, якщо хоча б один елемент відповідає предикату, який передається у функцію як параметр;
- **average(): Double**: термінальна операція, яка повертає середнє значення для числової послідовності типів **Byte**, **Int**, **Short**, **Long**, **Float**, **Double**;
- **chunked(size: Int): Sequence<List<T>>**: проміжна операція, яка розщеплює послідовність на послідовність зі списків **List**, параметр **size** встановлює максимальну кількість елементів у кожному зі списків;
- **chunked(size: Int,transform: (List<T>) -> R): Sequence<R>**: додаткова версія, яка як другий параметр отримує функцію перетворення, яка перетворює кожен список на елемент нової послідовності;
- **contains(element: T): Boolean**: термінальна операція, яка повертає **true**, якщо послідовність містить елемент **element**;
- **count(): Int**: термінальна операція, яка повертає кількість елементів у послідовності;
- **count(predicate: (T) -> Boolean): Int**: додаткова версія, яка повертає кількість елементів, які відповідають предикату;
- **distinct(): Sequence<T>**: проміжна операція, яка повертає нову послідовність, яка містить лише унікальні елементи;
- **distinctBy(selector: (T) -> K): Sequence<T>**: проміжна операція, яка повертає нову послідовність, яка містить лише унікальні елементи з урахуванням функції селектора, яка передається як параметр;
- **drop(n: Int): Sequence<T>**: проміжна операція, яка повертає нову послідовність, яка містить усі елементи за винятком перших **n** елементів;
- **dropWhile(predicate: (T) -> Boolean): Sequence<T>**: проміжна операція, яка повертає нову послідовність, яка містить усі елементи за винятком перших елементів, які відповідають предикату;
- **elementAt(index: Int): T**: термінальна операція, яка повертає елемент за індексом **index**. Якщо індекс виходить за межі послідовності, то генерується виняток типу **IndexOutOfBoundsException**;
- **elementAtOrElse(index: Int, defaultValue: (Int) -> T): T**: термінальна операція, яка повертає елемент за індексом **index**. Якщо індекс виходить за межі послідовності, то повертається значення, яке встановлюється функцією з параметра **defaultValue**;

- **elementAtOrNull(index: Int): T?**: термінальна операція, яка повертає елемент за індексом **index**. Якщо індекс виходить межі послідовності, то повертається **null**;
- **filter(predicate: (T) -> Boolean): Sequence<T>**: проміжна операція, яка повертає нову послідовність із елементів, які відповідають предикату;
- **filterNot(predicate: (T) -> Boolean): Sequence<T>**: проміжна операція, яка повертає нову послідовність із елементів, які НЕ відповідають предикату;
- **filterNotNull(): Sequence<T>**: проміжна операція, яка повертає нову послідовність з елементів, які не дорівнюють **null**;
- **find(predicate: (T) -> Boolean): T?**: термінальна операція, яка повертає перший елемент, що відповідає предикату. Якщо елемент не знайдено, то повертається **null**;
- **findLast(predicate: (T) -> Boolean): T?**: термінальна операція, яка повертає останній елемент, що відповідає предикату. Якщо елемент не знайдено, то повертається **null**;
- **first(): T**: термінальна операція, яка повертає перший елемент послідовності;
- **first(predicate: (T) -> Boolean): T**: додаткова версія, яка повертає перший елемент, який відповідає предикату. Якщо елемент не знайдено, то генерується виняток типу **NoSuchElementException**;
- **firstOrNull(): T?**: термінальна операція, яка повертає перший елемент послідовності;
- **firstOrNull(predicate: (T) -> Boolean): T?**: додаткова версія, яка повертає перший елемент, який відповідає предикату. Якщо елемент не знайдено, то повертається **null**;
- **flatMap(transform: (T) -> Sequence<R>): Sequence<R>**: проміжна операція, яка перетворює послідовність елементів типу **T** на послідовність елементів типу **R**, використовуючи функцію перетворення, яка передається як параметр;
- **fold(initial: R, operation: (acc: R, T) -> R): R**: проміжна термінальна операція, яка повертає значення, яке є результатом дії функції **operation** над кожним елементом послідовності. Перший параметр функції **operation** – це результат роботи функції над попереднім елементом послідовності (при першому виклику значення береться з параметра **initial**), а другий параметр – це поточний елемент послідовності;

- **forEach(action: (T) -> Unit)**: термінальна операція, яка виконує для кожного елемента послідовності дію **action**;
- **groupBy(keySelector: (T) -> K): Map<K, List<T>>**: термінальна операція, яка групує елементи ключа, який повертається функцією **keySelector**. Результатом функції є колекція **Map**, де ключ є, власне, ключем елементів, а значення є списком **List** з елементів, які відповідають цьому ключу;
- **groupBy(keySelector: (T) -> K, значенняTransform: (T) -> V): Map<K, List<V>>**: додаткова версія, яка приймає функцію перетворення елементів;
- **indexOf(element: T): Int**: термінальна операція, яка повертає індекс першого входження елемента **element**. Якщо елемент не знайдено, повертається -1;
- **indexOfFirst(predicate: (T) -> Boolean): Int**: термінальна операція, яка повертає індекс першого елемента, що відповідає предикату. Якщо елемент не знайдено, повертається -1;
- **indexOfLast(predicate: (T) -> Boolean): Int**: термінальна операція, яка повертає індекс останнього елемента, що відповідає предикату. Якщо елемент не знайдено, повертається -1;
- **joinToString(): String**: термінальна операція, яка генерує з послідовності рядок;
- **last(): T**: термінальна операція, яка повертає останній елемент послідовності;
- **last(predicate: (T) -> Boolean): T**: додаткова версія, яка повертає останній елемент, який відповідає предикату. Якщо елемент не знайдено, то генерується виняток типу **NoSuchElementException**;
- **lastOrNull(): T?**: термінальна операція, яка повертає останній елемент послідовності;
- **lastOrNull(predicate: (T) -> Boolean): T?**: додаткова версія, яка повертає останній елемент, який відповідає предикату. Якщо елемент не знайдено, то повертається **null**;
- **lastIndexOf(element: T): Int**: термінальна операція, яка повертає останній індекс елемента. Якщо елемент не знайдено, повертається -1;
- **map(transform: (T) -> R): Sequence<R>**: проміжна операція, яка застосовує до елементів послідовності функцію трансформації та повертає нову послідовність із нових елементів;

- **mapIndexed(transform: (index: Int, T) -> R): Sequence<R>**: проміжна операція, яка застосовує до елементів послідовності та їх індексів функцію трансформації та повертає нову послідовність із нових елементів;
- **mapNotNull(transform: (T) -> R?): Sequence<R>**: проміжна операція, яка застосовує до елементів послідовності функцію трансформації і повертає нову послідовність нових елементів, які не дорівнюють **null**;
- **maxOf(selector: (T) -> Double): Double**: термінальна операція, яка повертає максимальне значення на основі селектора;
- **maxOfOrNull(selector: (T) -> Double): Double?**: термінальна операція, яка повертає максимальне значення з урахуванням селектора. Якщо послідовність порожня, то повертається **null**;
- **maxOrNull(): Double?**: термінальна операція, яка повертає максимальне значення. Якщо послідовність порожня, то повертається **null**;
- **minOf(selector: (T) -> Double): Double**: термінальна операція, яка повертає мінімальне значення на основі селектора;
- **minOfOrNull(selector: (T) -> Double): Double?**: термінальна операція, яка повертає мінімальне значення з урахуванням селектора. Якщо послідовність порожня, то повертається **null**;
- **minOrNull(): Double?**: термінальна операція, яка повертає мінімальне значення. Якщо послідовність порожня, то повертається **null**;
- **minus(element: T): Sequence<T>**: проміжна операція, яка повертає нову послідовність, яка містить всі елементи поточної за винятком елемента **element**. Має різновиди, які дозволяють виключити з послідовності набори елементів, а саме:

```
1  minus(elements: Array<T>): Sequence<T>
2  minus(elements: Iterable<T>): Sequence<T>
3  minus(elements: Sequence<T>): Sequence<T>
```

- **plus(element: T): Sequence<T>**: проміжна операція, яка повертає нову послідовність, яка містить усі елементи поточної за плюс елемент **element**. Має різновиди, які дозволяють включити до послідовності набори елементів, а саме:

```
1   plus(elements: Array<T>): Sequence<T>
2   plus(elements: Iterable<T>): Sequence<T>
3   plus(elements: Sequence<T>): Sequence<T>
```

- **reduce(operation: (acc: S, T) -> S): S**: проміжна операція, яка повертає значення, яке є результатом дії функції **operation** над кожним елементом

послідовності. Перший параметр функції **operation** - це результат роботи функції над попереднім елементом послідовності, а другий параметр - це поточний елемент послідовності;

- **shuffled(): Sequence<T>**: проміжна операція, яка умовно перемішує послідовність;
- **sorted(): Sequence<T>**: проміжна операція, яка упорядковує послідовність за зростанням;
- **sortedBy(selector: (T) -> R?): Sequence<T>**: проміжна операція, яка впорядковує послідовність за зростанням на основі селектора;
- **sortedByDescending(selector: (T) -> R?): Sequence<T>**: проміжна операція, яка впорядковує послідовність за спаданням на основі функції-селектора;
- **sortedDescending(): Sequence<T>**: проміжна операція, яка впорядковує послідовність за спаданням;
- **sum(): Int**: термінальна операція, яка повертає суму елементів послідовності;
- **sumOf(selector: (T) -> Int): Int**: термінальна операція, яка повертає суму елементів послідовності на основі функції-селектора;
- **take(n: Int): Sequence<T>**: проміжна операція, яка повертає нову послідовність, яка містить **n** перших елементів поточної послідовності;
- **takeWhile(predicate: (T) -> Boolean): Sequence<T>**: проміжна операція, яка повертає нову послідовність, яка містить **n** перших елементів поточної послідовності, що відповідають функції-предикату;
- **toHashSet(): HashSet<T>**: термінальна операція, яка створює з послідовності об'єкт **HashSet**;
- **toList(): List<T>**: термінальна операція, яка створює із послідовності об'єкт **List**;
- **toMap(): Map<K, V>**: термінальна операція, яка створює з послідовності об'єкт **Map**;
- **toSet(): Set<T>**: термінальна операція, яка створює із послідовності об'єкт Set.

7.6. Відмінності послідовностей від колекцій Iterable

7.6.1. Основні відмінності послідовностей від колекцій Iterable

І послідовності, і колекції, що реалізують інтерфейс **Iterable**, по суті, є набором елементів. Більше того, надають схожий набір операцій для обробки

елементів. Але відмінність полягає у тому, як ці операції обробляють елементи при застосуванні відразу кількох операцій.

Так, при застосуванні набору операцій до колекції **Iterable**, кожна окрема операція повертає проміжний результат, так звану проміжну колекцію. А при обробці послідовності весь набір операцій виконується лише тоді, коли потрібний кінцевий результат обробки.

Також змінюється порядок застосування операцій. Колекція застосовує кожну операцію послідовно до кожного елемента. Тобто спочатку виконує першу операцію для всіх елементів, потім другу операцію для елементів колекції, отриманих після першої операції. І так далі.

Послідовність застосовує весь набір операцій окремо до кожного елемента. Тобто спочатку весь набір операцій застосовується до першого елемента, потім другого елемента і так далі. Таким чином, послідовність дозволяє уникнути створення проміжних колекцій і одночасно підвищує продуктивність при виконанні набору операцій особливо для великого набору даних. Однак, при невеликих наборах даних та малій кількості операцій може бути ефективніше використовувати колекції **Iterable**.

Розглянемо низку прикладів.

Спочатку розглянемо колекції **Iterable**, а саме **List**, наприклад:

```
1   fun main(){
2
3       var people = listOf(
4           Person("Taras", 37),
5           Person("Stepan", 25),
6           Person("Olena", 33)
7       )
8       people = people.filter { println("Age filter:
    ${it}"); it.age > 30 }
9                       .filter{ println("Name filter:
    ${it}"); it.name.length == 3 }
10      println("Result:")
11      for(person in people) println(person)
12  }
13  data class Person(val name: String, val age: Int)
```

В наведному прикладі створюється колекція, а саме список людей (об'єкт типу **List**), який містить об'єкти **Person**. Далі до цього списку застосовуються дві операції фільтрації у вигляді методу **filter()**. Спочатку отримуються всі об'єкти **Person**, у яких властивість **age** більше 30, а саме:

```
1   people.filter { println("Age filter: ${it}"); it.age
    > 30 }
```

Ця операція **filter()** поверне проміжну колекцію, яка містить усі об'єкти **Person** з віком більше 30. Для наочності в цьому прикладі об'єкти, до яких застосовується ця операція, виводиться на консоль.

Потім виконується друга операція **filter()**, яка повертає з проміжної колекції об'єкти **Person**, у яких довжина властивості **name** дорівнює 3, а саме:

```
1   filter{ println("Name filter: ${it}"); it.name.length
    == 3 }
```

Для наочності в цьому прикладі на консоль виводяться об'єкти, до яких застосовується ця операція.

Результатом буде друга колекція, яка буде присвоєна змінній **people**, яку в кінці за допомогою циклу **foreach** виводиться на консоль. Результа роботи програми наведено на рисунку 7.4.

```
Age filter: Person(name=Taras, age=37)
Age filter: Person(name=Stepan, age=25)
Age filter: Person(name=Olena, age=33)
Name filter: Person(name=Taras, age=37)
Name filter: Person(name=Olena, age=33)
Result:
Person(name=Taras, age=37)
```

Рисунок 7.4. Результат роботи програми

Отже видно, що перша операція **filter** пробігається по всіх елементах початкової колекції і повертає колекцію з двома елементами. Друга операція **filter** пробігається отриманою колекцією з двох елементів і повертає колекцію з одного елемента.

Тепер замість колекцій розглянемо приклад із застосуванням послідовності, а саме:

```
1   fun main(){
2
3       var people = sequenceOf(
4           Person("Taras", 37),
5           Person("Stepan", 25),
6           Person("Olena", 33)
7       )
8       people = people.filter { println("Age filter:
    ${it}"); it.age > 30 }
9                       .filter{ println("Name filter:
    ${it}"); it.name.length == 3 }
10      println("Result:")
```

```
11      for(person in people) println(person)
12  }
13  data class Person(val name: String, val age: Int)
```

В цьому прикладі абсолютно такий же код, як і в попередньому прикладі, тільки змінна **people** тепер представляє послідовність об'єктів **Person**. Проте результат роботи програми, який наведено на рисунку 7.5, буде зовсім іншим.

```
Result:
Age filter: Person(name=Taras, age=37)
Name filter: Person(name=Taras, age=37)
Person(name=Taras, age=37)
Age filter: Person(name=Stepan, age=25)
Age filter: Person(name=Olena, age=33)
Name filter: Person(name=Olena, age=33)
```

Рисунок 7.5. Результат роботи програми

По-перше, варто звернути увагу та те, що рядок «**Result**:» виводиться на консоль до виконання всіх операцій. Тому що одержання результату фактично відбувається у циклі **for** при зверненні до послідовності. До цього немає сенсу виконувати операції, якщо елементи послідовності не використовуються.

По-друге, також звернемо увагу на застосування операцій до окремих елементів. Спочатку обробляється перший елемент - **Person(name=Taras, age=37)**. Оскільки він відповідає обом фільтрам, то він зрештою виводиться на консоль у циклі **for**. Далі обробляється другий елемент - **Person(name=Stepan, age=25)**, проте після застосування першої операції **filter** його обробка завершується, оскільки він не відповідає умові першого фільтра. Наприкінці обробляється третій елемент - **Person(name=Olena, age=33)**, до нього застосовують дві операції **filter**, але потім його обробка завершується, оскільки він не відповідає умові другого фільтра.

Отримання елементів послідовності відбувається тоді, коли йде безпосереднє звернення до елементів послідовності. Наприклад, можна змінити попередній приклад наступним чином:

```
1   fun main(){
2
3       var people = sequenceOf(
4           Person("Taras", 37),
5           Person("Stepan", 25),
6           Person("Olena", 33)
7       )
```

```
8       people = people.filter { println("Age filter:
    ${it}"); it.age > 30 }
9       println("Between Age filter and Name filter")
10      people = people.filter{ println("Name filter:
    ${it}"); it.name.length == 3 }
11      for(person in people) println(person)
12  }
13  data class Person(val name: String, val age: Int)
```

В цьому прикладі результат кожного фільтра присвоюється змінній **people**. А між фільтрами здійснюється виведення повідомлення на консоль. Але все одно, оскільки безпосереднє отримання елементів послідовності відбувається у циклі **for**, то саме в цій точці коду виконуватимуться всі операції з послідовністю, що можна побачити з результату роботи програми, який наведено на рисунку 7.6.

```
Between Age filter and Name filter
Age filter: Person(name=Taras, age=37)
Name filter: Person(name=Taras, age=37)
Person(name=Taras, age=37)
Age filter: Person(name=Stepan, age=25)
Age filter: Person(name=Olena, age=33)
Name filter: Person(name=Olena, age=33)
```

Рисунок 7.6. Результат роботи програми

7.6.2. Скорочення набору операцій

Застосування послідовностей може значно скоротити кількість операцій, що застосовуються. Розглянемо наступний приклад:

```
1   fun main(){
2
3       var people = listOf(
4           Person("Taras", 37),
5           Person("Stepan", 25),
6           Person("Olena", 33)
7       )
8       people = people.filter { println("Age filter:
    ${it}"); it.age > 30 }
9                       .take(1)
10      for(person in people) println(person)
11
12  }
13  data class Person(val name: String, val age: Int)
```

В цьому прикладі до списку **people** застосовується фільтр за віком і потім за допомогою виклику **take(1)** вибирається один об'єкт у результуючу колекцію. В цьому разі отримаємо результат роботи програми, який наведено на рисунку 7.7.

```
Age filter: Person(name=Taras, age=37)
Age filter: Person(name=Stepan, age=25)
Age filter: Person(name=Olena, age=33)
Person(name=Taras, age=37)
```

Рисунок 7.7. Результат роботи програми

В цьому прикладі видно, що виклик **filter()** застосовується до всіх елементів, з яких формується проміжна колекція, з якої в результаті вибирається лише один об'єкт.

Змінимо тип набору на послідовність, як наведено нижче:

```
fun main(){

    var people = sequenceOf(
        Person("Taras", 37),
        Person("Stepan", 25),
        Person("Olena", 33)
    )
    people = people.filter { println("Age filter: ${it}"); it.age > 30 }
                        .take(1)
    for(person in people) println(person)

}
data class Person(val name: String, val age: Int)
```

Результат роботи програми наведено на рисунку 7.8.

```
Age filter: Person(name=Taras, age=37)
Person(name=Taras, age=37)
```

Рисунок 7.8. Результат роботи програми

В цьому коді програми спочатку обробляється перший об'єкт - **Person(name=Taras, age=37)**, для якого застосовується виклик **filter()**. Оскільки цей об'єкт відповідає фільтру, він переходить до виконання виклику **take(1)**. Цей виклик вибирає в результуючу колекцію перший об'єкт. Але оскільки результуюча колекція повинна містити лише 1 об'єкт, інші елементи послідовності немає сенсу обробляти. І на цьому обробка послідовності

закінчується. Таким чином, замість 3 операцій **filter** в наведеному прикладі ця операція виконується лише 1 раз. Відповідно на більшій кількості даних та операцій скорочення набору операцій може бути же більшим.

7.7. Фільтрація

7.7.1. Фільтрація за умовою

Фільтрація є однією з найпоширеніших операцій. Для фільтрації за умовою застосовується функція **filter()**, яка як параметр приймає умову-предикат у вигляді функції **(T) -> Boolean**, а саме:

```
filter(predicate: (T) -> Boolean): List<T>/Map<K, V>/Sequence<T>
```

Функція предикату приймає як параметр елемент набору. Якщо елемент відповідає умові, то повертається **true**, а цей елемент поміщається у набір, що повертається.

Для колекцій **List** і **Set** ця функція повертає об'єкт **List**, для **Map** ця функція повертає об'єкт **Map**, а для послідовностей **Sequence** ця функція повертає також об'єкт **Sequence**. Розглянемо наступний приклад:

```
fun main(){

    var people = sequenceOf("Taras", "Stepan", "Myhailo", "Volodymyr", "Olena")
    people = people.filter{it.length == 5}    // отримуємо значення з довжиною у 5 символів
    println(people.joinToString())  // Taras, Olena

    var employees = listOf(
        Person("Taras", 37),
        Person("Volodymyr", 41),
        Person("Stepan", 25)
    )
    employees = employees.filter{it.age > 30} // отримуємо всіх Person, у яких age > 30
    println(employees.joinToString())  // Person(name=Taras, age=37), Person(name=Volodymyr, age=41)
}
data class Person(val name: String, val age: Int)
```

Якщо потрібно отримати елементи, які, навпаки, не відповідають умові, то можна застосувати функцію **filterNot()**, яка працює аналогічно, наприклад:

```
var people = sequenceOf("Taras", "Stepan", "Myhailo",
```

```
    "Volodymyr", "Olena")
2   people = people.filterNot{it.length == 5}     //
    отримуємо значення з довжиною, яка не дорівнює 5
    символам
3   println(people.joinToString())  // Stepan, Myhailo,
    Volodymyr
```

7.7.2. Фільтрація за індексом

Розглянемо функцію **filterIndexed()**, яка отримує індекс поточного елемента:

```
1   filterIndexed(predicate: (index: Int, T) -> Boolean):
    List<T>
```

Для прикладу, отримаємо з колекції рядків елементи з парними індексами та довжиною у 5 символів, а саме:

```
1   val people = listOf("Taras", "Myhailo", "Stepan",
    "Volodymyr", "Olena")
2   // отримуємо значення з довжиною у 5 символів на
    парних індексах
3   val filtered = people.filterIndexed{ index, s ->
    (index % 2 == 0) && (s.length == 5)}
4   println(filtered)  // [Taras, Olena]
```

7.7.3. Фільтрація за типом

Якщо колекція чи послідовність містить елементи різних типів, то за допомогою функції **filterIsInstance()** можна отримати елементи певного типу. Розглянемо наступний приклад:

```
1   fun main(){
2
3       val people = listOf(
4           Person("Taras"), Employee("Volodymyr"),
5           Person("Stepan"), Employee("Myhailo")
6       )
7       // отримуємо тільки елементи типу Employee
8       val employees =
    people.filterIsInstance<Employee>();
9       println(employees)  // [Volodymyr, Myhailo]
10  }
11  open class Person(val name: String){
12      override fun toString(): String  = name
13  }
14  class Employee(name: String): Person(name)
```

В цьому прикладі з колекції **people** отримується лише ті об'єкти, які представляють тип **Employee**. Щоб вказати тип одержуваних об'єктів, функція типізується відповідним типом.

7.7.4. Фільтрація по null

Функція **filterNotNull()** дозволяє відфільтрувати всі значення, які дорівнюють **null**, а саме:

```
filterNotNull(): List<T>/Sequence<T>
```

Розглянемо наступний приклад:

```
fun main(){

    val people = listOf(Person("Taras"), null, Person("Stepan"), null)
    println(people)     // [Taras, null, Stepan, null]
    val filtered = people.filterNotNull()
    println(filtered)  // [Taras, Stepan]
}
open class Person(val name: String){
    override fun toString(): String  = name
}
```

7.8. Перевірка елементів

7.8.1. Перевірка відповідності елементів умові

Окремий вид операцій дозволяє перевірити наявність елементів.

Функція **all** перевіряє, чи всі елементи колекції чи послідовності відповідають умові предикату, а саме:

```
all(predicate: (T) -> Boolean): Boolean
```

Функція предикату отримує кожен елемент та повертає **true**, якщо елемент відповідає умові.

Функція **any** перевіряє, чи відповідає хоча б один елемент колекції або послідовності умові предикату, а саме:

```
any(predicate: (T) -> Boolean): Boolean
```

Функція **none** повертає **true**, якщо жоден з елементів не відповідає умові предикату, а саме:

```
none(predicate: (T) -> Boolean): Boolean
```

Нижче наведено декілька прикладів застосування цих функцій.

```
val people = listOf("Taras", "Kateryna", "Stepan",
```

```
    "Olena", "Volodymyr")
2   // приклад застосування функції all
3   println(people.all{it.length == 5})      // false
4   println(people.all{it.length != 15})      // true
5
6   // приклад застосування функції none
7   println(people.none{it.length == 5})      // false
8   println(people.none{it.length == 4})      // true
9
10  // приклад застосування функції any
11  println(people.any{it.length == 5})      // true
12  println(people.any{it.length == 15})     // false
```

Функції **any()** та **none()** також мають версії без параметрів. У цьому випадку функція **any()** повертає **true**, якщо колекція чи послідовність містить хоча б один елемент, а функція **none()** повертає **true**, якщо колекція чи послідовність порожня, наприклад:

```
1   val people = listOf("Taras", "Kateryna", "Stepan",
    "Olena", "Volodymyr")
2   val empty: List<String> = listOf()
3   // приклад застосування функції any
4   println(people.any())      // true
5   println(empty.any())      // false
6
7   // приклад застосування функції none
8   println(people.none())      // false
9   println(empty.none())      // true
```

Функція **contains()** повертає **true**, якщо в колекції чи послідовності є певний елемент, наприклад:

```
1   val people = listOf("Taras", "Stepan", "Kateryna",
    "Volodymyr", "Olena")
2   println(people.contains("Stepan"))      // true
3   println(people.contains("Vadym"))     // false
```

Функція **containsAll()** повертає **true**, якщо колекція містить усі елементи іншої колекції, наприклад:

```
1   val people = listOf("Taras", "Stepan", "Kateryna",
    "Volodymyr", "Olena")
2   println(people.containsAll(listOf("Taras", "Stepan")) )
    // true
3   println(people.containsAll(listOf("Taras", "Vadym")) )
    // false
```

7.9. Трансформації

7.9.1. Функція map()

Для трансформації однієї колекції чи послідовності використовується функція **map()**, а саме:

```
map(transform: (T) -> R): List<R>/Sequence<R>
```

Як параметр функція **map()** приймає функцію перетворення. Функція перетворення отримує поточний елемент колекції чи послідовності та повертає результат перетворення. Причому типи вхідних і вихідних даних можуть збігатися, а можуть і відрізнятися, наприклад:

```
fun main(){

    val people = listOf(Person("Taras"), Person("Stepan"), Person("Volodymyr"))

    val names = people.map { it.name } // повертаємо ім'я кожного користувача
    println(names)  // [Taras, Stepan, Volodymyr]
}
class Person(val name: String)
```

В цьому прикладі із списку **List<Person>** отримується список **List<String>**, який містить імена користувачів.

Розглянемо інший приклад, а саме трансформацію послідовності чисел у послідовність квадратів цих чисел:

```
val numbers = listOf(1, 2, 3, 4, 5)
val squares = numbers.map { it * it }
println(squares)    // [1, 4, 9, 16, 25]
```

7.9.2. Функція mapIndexed()

Функція **mapIndexed()** передає у функцію перетворення індекс поточного елемента, а саме:

```
mapIndexed(transform: (index: Int, T) -> R): List<R>
```

Приклад застосування функції **mapIndexed()** наведено нижче:

```
fun main(){

    val people = listOf(Person("Taras"), Person("Stepan"), Person("Volodymyr"))

    val names = people.mapIndexed{ index, p-> "${index+1}.${p.name}"}
    println(names)  // [1.Taras, 2.Stepan,
```

```
    3.Volodymyr]
}
class Person(val name: String)
```

7.9.3. Функції mapNotNull() та mapIndexedNotNull().

Якщо необхідно відсіяти значення **null**, які можуть виникати при перетворенні, то можна застосовувати аналоги вищезазначених функцій, а саме **mapNotNull()** та **mapIndexedNotNull()**, наприклад:

```
fun main(){

    val people = listOf(
        Person("Taras"), Person("Stepan"),
        Person("Volodymyr"), Person("Olena")
    )
    // елементи з довжиною імені, яка не дорівнює 5, перетворюємо в null
    val names1 = people.mapNotNull{ if(it.name.length!=5) null else it.name }

    // елементи на парних позиціях перетворюємо в null
    val names2 = people.mapIndexedNotNull{ index, p -> if(index%2==0) null else p.name }

    println(names1)  // [Taras, Olena]
    println(names2)  // [Stepan, Olena]
}
class Person(val name: String)
```

7.9.4. Функція flatten()

Функція **flatten()** дозволяє перетворити колекцію чи послідовність, яка містить вкладені колекції чи послідовності, а саме:

```
flatten(): List<T>/Sequence<T>
```

Ця функція розташовує елементи всіх вкладених колекцій в одну, наприклад:

```
val personal = listOf(listOf("Taras", "Volodymyr"), listOf("Stepan", "Myhailo", "Kateryna"), listOf("Taras", "Vadym"))
val people = personal.flatten()
println(people)    // [Taras, Volodymyr, Stepan, Myhailo, Kateryna, Taras, Vadym]
```

7.10. Групування

Для групування елементів колекції чи послідовності застосовується функція **groupBy()**, а саме:

```
groupBy(keySelector: (T) -> K): Map<K, List<T>>
groupBy(keySelector: (T) -> K,valueTransform: (T) -> V): Map<K, List<V>>
```

Обидві версії функції **groupBy()** як параметр приймають функцію, яка визначає критерій групування. Друга версія на додаток приймає функцію перетворення. Результатом функції є об'єкт **Map**, який зберігає набір груп. Ключами є критерії групування (ключі груп), а значеннями є списки **List**, які відповідають цим критеріям групування та представляють групи.

Наприклад, згрупуємо співробітників за їхніми компаніями:

```
fun main(){

    val employees = listOf(
        Employee("Taras", "Microsoft"),
        Employee("Volodymyr", "JetBrains"),
        Employee("Stepan", "Google"),
        Employee("Olena", "Microsoft"),
        Employee("Kateryna", "Google")
    )
    val companies = employees.groupBy { it.company }    // обєкт Map<String, List<Employee>>

    println(companies) // {Microsoft=[Taras, Olena], JetBrains=[Volodymyr], Google=[Stepan, Kateryna]}

    // перебір груп
    for (company in companies){
        println(company.key) // назва компанії
        // перебір списку співробітників
        for (employee in company.value){
            println(employee.name)
        }
        println() // переведення рядка для розмежування груп
    }
}
class Employee(val name: String, val company: String){
    override fun toString(): String = name
}
```

В цьому прикладі критерієм групування є властивість **company** об'єкта **Employee**. Відповідно ключем групи в об'єкті **Map**, який є результатом, буде

значення цієї властивості, а значенням буде список **Employee**. Потім можна отримати всі значення кожної групи стандартним перебором об'єкта **Map**. Результат роботи програми наведено на рисунку 7.9.

```
{Microsoft=[Taras, Olena], JetBrains=[Volodymyr], Google=[Stepan, Kateryna]}
Microsoft
Taras
Olena

JetBrains
Volodymyr

Google
Stepan
Kateryna
```

Рисунок 7.9. Результат роботи програми

Тепер застосуємо іншу версію функції **groupBy** для виконання трансформації, а саме:

```
1   fun main(){
2
3       val employees = listOf(
4           Employee("Taras", "Microsoft"),
5           Employee("Volodymyr", "JetBrains"),
6           Employee("Stepan", "Google"),
7           Employee("Olena", "Microsoft"),
8           Employee("Kateryna", "Google")
9       )
10      val companies = employees.groupBy({it.company})
    { it.name }  // обєкт Map<String, List<String>>
11
12      println(companies) // {Microsoft=[Taras, Olena],
    JetBrains=[Volodymyr], Google=[Stepan, Kateryna]}
13      // перебір груп
14      for (company in companies){
15          println(company.key) // назва компанії
16          // перебір списку співробітників
17          for (employee in company.value){
18              println(employee)
19          }
20          println() // переведення рядка для
```

```
    розмежування груп
21      }
22  }
23  class Employee(val name: String, val company:
    String){
24      override fun toString(): String = name
25  }
```

В наведеному прикладі як критерій групування виступає властивість **company** об'єктів **Employee** (**{ it.company }**), проте сама група представлятиме список рядків, тобто об'єкт **List<String>**, оскільки функція перетворення витягує значення властивості **name** об'єкта **Employee** (**{ it.name }**).

7.11. Впорядковування

Для впорядковування колекції чи послідовності застосовуються функції **sorted()** (впорядкування за зростанням) та **sortedDescending()** (впорядкування за спаданням).

Розглянемо, що в цьому випадку означає впорядкування за зростанням чи за спаданням. За замовчуванням процес впорядкування спирається на реалізацію інтерфейсу **Comparable**, яка визначає, який об'єкт буде більшим, а який меншим. Так, для вбудованих базових типів діє наступна логіка:

- числа порівнюються як у математиці виходячи з їх значення;
- символи (**Char**) і рядки (**String**) порівнюються виходячи з лексикографічного порядку, тобто "**Taras**" більше, ніж "**Olena**", оскільки перший символ **T** розташовується в алфавіті після символу **A**.

Приклади впорядкування чисел та рядків наведено нижче:

```
1   val people = listOf("Taras", "Myhailo", "Volodymyr",
    "Stepan", "Olena")
2   val numbers = listOf(3, 5, 2, -4, -6, 9, 1)
3
4   // впорядкування за зростанням
5   val sortedPeople = people.sorted()
6   val sortedNumbers = numbers.sorted()
7   println(sortedPeople)    // [Volodymyr, Myhailo,
    Olena, Stepan, Taras]
8   println(sortedNumbers)   // [-6, -4, 1, 2, 3, 5, 9]
9
10  // впорядкування за спаданням
11  println(people.sortedDescending())  // [Taras,
    Stepan, Olena, Myhailo, Volodymyr]
12  println(numbers.sortedDescending()) // [9, 5, 3, 2,
    1, -4, -6]
```

7.11.1. Реалізація інтерфейсу Comparable

Якщо визначаються власні типи та вимагається, щоб їх об'єкти також можна було впорядковувати, то в цьому випадку необхідно реалізувати інтерфейс **Comparable**, а саме:

```
public interface Comparable<in T> {
    public operator fun compareTo(other: T): Int
}
```

Його функція **compareTo()** має визначати логіку порівняння. Як параметр вона приймає об'єкт, який порівнюється з поточним об'єктом.

В якості результату повертається число. Якщо поточний об'єкт дорівнює об'єкту **other**, то функція повинна повернути **0**. Якщо поточний об'єкт більший за об'єкт **other**, то повертається позитивне число, якщо менше, то негативне.

Розглянемо наступний приклад реалізації інтерфейсу **Comparable**:

```
fun main(){

    val people = listOf(
        Person("Taras", 37),
        Person("Volodymyr",41),
        Person("Stepan", 25)
    )
    // впорядкування за зростанням
    val sortedPeople = people.sorted()

    // впорядкування за зростанням
    println(sortedPeople)   // [Volodymyr (41), Stepan (25), Taras (37)]

    // впорядкування за спаданням
    println(people.sortedDescending())  // [Taras (37), Stepan (25), Volodymyr (41)]
}
class Person(val name: String, val age: Int): Comparable<Person> {
    override fun compareTo(other: Person): Int = name.compareTo(other.name)
    override fun toString(): String = "$name ($age)"
}
```

В наведеному прикладі клас **Person** реалізує інтерфейс **Comparable**, однак у самому методі **compareTo** фактично покладаються на реалізацію цього інтерфейсу для рядків. Тобто фактично повертається результат порівняння двох рядків – імен користувачів, а саме:

```
name.compareTo(other.name)
```

У результаті об'єкти **Person** впорядковуватимуться відповідно до розташування перших літер їх імен у лексикографічному порядку, тобто за стандартним впорядкуванням для рядків.

Тепер змінимо принцип впорядкування та порівняємо два об'єкти за їх віком, тобто за значенням властивості **age**, наприклад:

```
fun main(){

    val people = listOf(
        Person("Taras", 37),
        Person("Volodymyr",41),
        Person("Stepan", 25)
    )
    // впорядкування за зростанням
    val sortedPeople = people.sorted()

    // впорядкування за зростанням
    println(sortedPeople)   // [Stepan (25), Taras (37), Volodymyr (41)]

    // впорядкування за спаданням
    println(people.sortedDescending())  // [Volodymyr (41), Taras (37), Stepan (25)]
}
class Person(val name: String, val age: Int): Comparable<Person> {
    override fun compareTo(other: Person): Int = age - other.age
    override fun toString(): String = "$name ($age)"
}
```

В цьому прикладі об'єкт **Person** вважається «більшим», якщо у нього більше значення властивості **age**.

7.11.2. Функція sortedWith та інтерфейс Comparator

Інтерфейс **Comparable** дозволяє легко визначити логіку впорядкування, проте буває так, що код класів, які необхідно впорядкувати, недоступний. Або необхідно встановити для вже існуючих типів, тобто для тих, які вже реалізують **Comparable**, інший спосіб впорядкування. У цьому випадку можна використовувати інтерфейс **Comparator**, так званий компаратор, наприклад:

```
public expect fun interface Comparator<T> {
    public fun compare(a: T, b: T): Int
}
```

Функція **compare()** приймає два параметри, а саме два об'єкти, що порівнюються, і також повертає ціле число. Якщо перший параметр більший за другий, то повертається позитивне число, якщо менше, то повертається негативне число. Якщо об'єкти однакові, то повертається 0.

Мова Kotlin має вбудовану функцію **sortedWith()**, яка як параметр приймає компаратор і на його основі впорядковує колекцію чи послідовність, а саме:

```
sortedWith(comparator: Comparator<in T>): List<T>
```

Розглянемо наступний приклад:

```
fun main(){

    val people = listOf(
        Person("Taras", 37),
        Person("Volodymyr",41),
        Person("Stepan", 25)
    )
    val personComparator = Comparator{ p1: Person,
p2: Person -> p1.age - p2.age }

    val sortedPeople =
people.sortedWith(personComparator)
    println(sortedPeople)   // [Stepan (25), Taras
(37), Volodymyr (41)]
}
class Person(val name: String, val age: Int){
    override fun toString(): String = "$name ($age)"
}
```

У цьому прикладі компаратор визначено у вигляді змінної **personComparator**. У реалізації інтерфейсу як і у попередньому прикладі порівнюємо користувачів з урахуванням віку, тобто якщо значення властивості **age** більше, то і об'єкт **Person** умовно вважається «більшим».

Крім того, у цьому випадку можна скоротити виклик функції впорядкування наступним чином:

```
val sortedPeople = people.sortedWith{ p1: Person, p2:
Person -> p1.age - p2.age }
```

або так:

```
val sortedPeople = people.sortedWith(compareBy{
it.age })
```

Завдяки компаратору можна встановити власну логіку впорядкування до вже наявних типів. Наприклад, впорядкуємо рядки за довжиною:

```
 fun main(){

     val people = listOf("Taras", "Olena", "Kateryna",
```

```
    "Stepan", "Volodymyr")
4       // впорядкуємо за довжиною рядка
5       val sorted  = people.sortedWith(compareBy{
    it.length })
6       println(sorted)   // [Taras, Olena, Stepan,
    Volodymyr]
7   }
```

7.11.3. Впорядкування на основі критерію

Ще дві функції дозволяють впорядкувати об'єкти за певним критерієм, а саме:

```
1   sortedBy(crossinline selector: (T) -> R?): List<T> /
    Sequence<T>
2   sortedByDescending(crossinline selector: (T) -> R?):
    List<T> / Sequence<T>
```

Функція **sortedBy()** впорядковує за зростанням, а **sortedByDescending()** впорядковує за спаданням. Як параметр вони приймають функцію, яка отримує елемент та повертає значення, що застосовується для впорядкування, наприклад:

```
1   fun main(){
2
3       val people = listOf( Person("Taras", 37),
    Person("Volodymyr",41), Person("Stepan", 25))
4
5       // впорядкування за властивістю name за зростанням
6       val sortedByName = people.sortedBy { it.name }
7       println(sortedByName)   // [Volodymyr (41), Stepan
    (25), Taras (37)]
8
9       // впорядкування за властивістю age за зростанням
10      val sortedByAge = people.sortedBy { it.age }
11      println(sortedByAge)   // [Stepan (25), Taras
    (37), Volodymyr (41)]
12
13      // впорядкування за властивістю age за спаданням
14      val sortedByAgeDesc = people.sortedByDescending {
    it.age }
15      println(sortedByAgeDesc)   // [Volodymyr (41),
    Taras (37), Stepan (25)]
16  }
17  class Person(val name: String, val age: Int){
18      override fun toString(): String = "$name ($age)"
19  }
```

7.11.4. Функції reverse та shaffle

Варто відзначити ще дві функції, які не впорядковують, а просто змінюють порядок елементів. Функція **reversed()** змінює порядок елементів на зворотний, а функція **shuffle()** перемішує елементи випадковим чином, наприклад:

```
1   val numbers = listOf(1, 2, 3, 4, 5, 6)
2   val reversed = numbers.reversed()
3   println(reversed)    // [6, 5, 4, 3, 2, 1]
4
5   val shuffled = numbers.shuffled()
6   println(shuffled)    // [4, 6, 3, 5, 1, 2]
```

7.12. Агрегатні операції

Колекції та послідовності підтримують низку агрегатних операцій, які повертають певне значення.

7.12.1. Мінімальне та максимальне значення

Функції **minOrNull()** і **maxOrNull()** повертають відповідно мінімальне та максимальне значення, а якщо колекція чи послідовність порожня, то повертається **null**. Причому для роботи цих функцій елементи колекції чи послідовності повинні реалізувати інтерфейс **Comparable**, а саме:

```
1   val numbers = listOf(4, 6, 3, 5, 1, 2)
2   val people = listOf("Olena", "Taras", "Stepan",
    "Kateryna", "Volodymyr")
3
4   println(numbers.minOrNull())    // 1
5   println(numbers.maxOrNull())    // 6
6
7   println(people.minOrNull())    // Volodymyr
8   println(people.maxOrNull())    // Taras
```

Функції **minByOrNull()** та **maxByOrNull()** як параметр приймають функцію селектора, яка дозволяє визначити критерій порівняння об'єктів, а саме;

```
1   fun main(){
2
3       val people = listOf( Person("Taras", 37),
    Person("Volodymyr",41), Person("Stepan", 25))
4       // мінімальний вік
5       val personWithMinAge = people.minByOrNull { it.age
    }
6       println(personWithMinAge)      // Stepan (25)
7       // максимальний вік
8       val personWithMaxAge = people.maxByOrNull { it.age
```

```
    }
9       println(personWithMaxAge)      // Volodymyr (41)
10  }
11  class Person(val name: String, val age: Int){
12      override fun toString(): String = "$name ($age)"
13  }
```

Вище наведені функції знаходили елемент із найменшим та найбільшим значенням властивості **age**. Але що якщо необхідно отримати не весь об'єкт, а лише самі значення мінімального та максимального віку? У цьому випадку можна використовувати функції **minOfOrNull()** та **maxOfOrNull()**, які також приймають функцію селектора, але повертають саме значення, на основі якого відбувається порівняння, наприклад:

```
1   fun main(){
2
3       val people = listOf( Person("Taras", 37),
    Person("Volodymyr",41), Person("Stepan", 25))
4       // мінімальний вік
5       val minAge = people.minOfOrNull { it.age }
6       println(minAge)      // 25
7       // максимальний вік
8       val maxAge = people.maxOfOrNull { it.age }
9       println(maxAge)      // 41
10  }
11  class Person(val name: String, val age: Int){
12      override fun toString(): String = "$name ($age)"
13  }
```

Ще пара функцій, а саме **minWithOrNull()** та **maxWithOrNull()** як параметр приймають компаратор реалізацію інтерфейсу **Comparator**, наприклад:

```
1   val colors = listOf("red", "green", "blue",
    "yellow")
2   val minColor = colors.minWithOrNull(compareBy
    {it.length})
3   val maxColor = colors.maxWithOrNull(compareBy
    {it.length})
4   println(minColor)    // red
5   println(maxColor)    // yellow
```

В якості критерію порівняння в цьому прикладі застосовується властивість рядків **length**, тобто рядки порівнюються за довжиною.

Розглянемо ще одну пару функцій: **minOfWithOrNull()** та **maxOfWithOrNull()**, які приймають реалізацію інтерфейсу **Comparator**, у якості першого параметру та селектор критерію для порівняння у якості другого параметру, а саме:

```
maxOfWithOrNull(comparator: Comparator<in R>, selector:
(T) -> R): R?
minOfWithOrNull(comparator: Comparator<in R>, selector:
(T) -> R): R?
```

Розглянемо наступний приклад застосування цих функцій:

```
fun main(){

    val people = listOf(
        Person("Taras", 37), Person("Kateryna",29),
        Person("Stepan", 25), Person("Lena", 33)
    )
    // найкоротше ім'я
    val minName =
people.minOfWithOrNull(compareBy{it.length}) {
it.name }
    println(minName)      // Lena
    // найдовше ім'я
    val maxName =
people.maxOfWithOrNull(compareBy{it.length}) {
it.name }
    println(maxName)      // Kateryna
}
class Person(val name: String, val age: Int){
    override fun toString(): String = "$name ($age)"
}
```

Як критерій порівняння в цьому прикладі застосовується властивість **name** об'єктів **Person**, тому що функція селектора визначена наступним чином: **{ it.name }**. Тут **it** – це поточний об'єкт **Person**.

Оскільки, як критерій порівняння застосовується властивість **name**, то до функції компаратора передається значення цієї властивості. І власне воно використовується для порівняння об'єктів **Person**, а саме: **compareBy{it.length}**. Тут **it** – це значення властивості **name**.

7.12.2. Середнє значення

Для отримання середнього значення застосовується функція **average()**, наприклад:

```
val numbers = listOf(4, 6, 3, 5, 1, 2)
val avg = numbers.average()
println(avg)    // 3.5
```

7.12.3. Сума значень

Для отримання суми числових значень застосовується функція **sum()**, наприклад:

```
1   val numbers = listOf(4, 6, 3, 5, 1, 2)
2   val sum = numbers.sum()
3   println(sum)     // 21
```

7.12.4. Кількість елементів

Для отримання кількості елементів колекції чи послідовності застосовується функція **count()**. Вона має два варіанти, а саме:

```
1   count(): Int
2   count(predicate: (T) -> Boolean): Int
```

Перша версія повертає кількість всіх елементів. Друга версія повертає кількість елементів, які відповідають умові предикату, що передається у функцію як параметр. Наведемо приклад застосування цих функцій:

```
1   val people = listOf("Taras", "Stepan", "Volodymyr",
    "Kateryna", "Olena")
2   // загальна кількість
3   val count1 = people.count()
4   println(count1)       // 5
5
6   // кількість рядків, у яких довжина дорівнює 5
7   val count2 = people.count{it.length == 5}
8   println(count2)       // 2
```

7.12.5. Зведення елементів

7.12.5.1. Функція reduce()

Функція **reduce()** зводить всі значення потоку до одного значення, а саме:

```
1   reduce(operation: (acc: S, T) -> S): S
```

Функція **reduce()** приймає функцію, яка має два параметри. Перший параметр при першому запуску представляє перший елемент, а при наступних запусках представляє результат функції над попередніми елементами. Другий параметр функції представляє наступний елемент.

Розглянемо приклад, коли є список чисел, а потрібно знайти суму всіх чисел, наприклад:

```
1   val numbers = listOf(1, 2, 3, 4, 5)
2   val reducedValue = numbers.reduce{ a, b -> a + b }
3   println(reducedValue)    // 15
```

В цьому прикладі при першому запуску у функції **reduce** параметр **a** дорівнює 1, а параметр **b** дорівнює 2. При другому запуску параметр **a** містить

результат попереднього виконання функції, тобто число 1 + 2 = 3, а параметр **b** дорівнює 3, тобто наступне число в потоці.

Розглянемо інший приклад із рядками:

```
val people = listOf("Taras", "Volodymyr", "Kateryna",
"Stepan", "Olena")
val reducedValue = people.reduce{ a, b -> "$a $b" }
println(reducedValue)   // Taras Volodymyr Kateryna
Stepan Olena
```

В цьому прикладі **reduce** з'єднує всі рядки в один.

7.12.5.2. Функція fold

Функція **fold** також зводить всі елементи потоку в один. Але на відміну від **reduce** в якості першого параметру приймає початкове значення, а саме:

```
fold(initial: R, operation: (acc: R, T) -> R): R
```

Розглянемо наступний приклад:

```
val people = listOf("Taras", "Volodymyr", "Kateryna",
"Stepan", "Olena")
val foldedValue = people.fold("People:", { a, b -> "$a
$b" })
println(foldedValue)   // People: Taras Volodymyr
Kateryna Stepan Olena
```

У цьому прикладі початковим значенням є рядок "**People:**", до якого потім додаються інші елементи списку.

7.13. Додавання, віднімання та об'єднання колекцій

7.13.1. Додавання

За допомогою функції **plus** можна складати колекції чи послідовності з іншими елементами або колекціями та послідовностями. Ця функція приймає як одиночний елемент, так і колекцію чи послідовність, а саме:

```
fun main(){
    val people = listOf("Taras", "Volodymyr", "Stepan")
    // додаємо один обєкт
    val result1 = people.plus("Olena")
    println(result1)  // [Taras, Volodymyr, Stepan,
Olena]

    val employees = listOf("Myhailo", "Kateryna")
    // додаємо колекцію об’єктів
    val result2 = people.plus(employees)
    println(result2)  // [Taras, Volodymyr, Stepan,
Myhailo, Kateryna]
```

```
11  }
```

Звернемо увагу, що початкова колекція, яка викликає функцію **plus()**, у наведеному прикладі це колекція **people**, не змінюється, а результатом об'єднання є нова колекція.

Замість функції **plus()** можна використовувати оператор +, наприклад:

```
1   fun main(){
2       val people = listOf("Taras", "Volodymyr", "Stepan")
3       val result1 = people + "Olena"
4       println(result1)  // [Taras, Volodymyr, Stepan,
    Olena]
5
6       val employees = listOf("Myhailo", "Kateryna")
7       val result2 = people + employees
8       println(result2)  // [Taras, Volodymyr, Stepan,
    Myhailo, Kateryna]
9   }
```

7.13.2. Віднімання

Аналогічним чином можна використовувати функцію **minus()** для віднімання або одиночного об'єкта, або колекції чи послідовності, наприклад:

```
1   fun main(){
2       val people = listOf("Taras", "Volodymyr", "Stepan",
    "Kateryna")
3       // віднімаємо один обєкт
4       val result1 = people.minus("Volodymyr")
5       println(result1)  // [Taras, Stepan, "Kateryna"]
6
7       val employees = listOf("Myhailo", "Kateryna")
8       // віднімаємо колекцію
9       val result2 = people.minus(employees)
10      println(result2)  // [Taras, Volodymyr, Stepan]
11  }
```

Початкова колекція, яка викликає функцію **minus()**, також не змінюється, і результатом віднімання є нова колекція.

Замість функції **minus()** можна використовувати оператор - , а саме:

```
1   val people = listOf("Taras", "Volodymyr", "Stepan",
    "Kateryna")
2   val result1 = people - "Volodymyr"
3   println(result1)  // [Taras, Stepan, "Kateryna"]
4
5   val employees = listOf("Myhailo", "Kateryna")
6   val result2 = people - employees
7   println(result2)  // [Taras, Volodymyr, Stepan]
```

7.13.3. Об'єднання

Для об'єднання двох різних колекцій чи послідовностей в одну застосовується функція **zip()**, а саме:

```
zip(other: Iterable<R>/Sequence<R>): List<Pair<T, R>>/Sequence<Pair<T, R>>
```

В якості параметра вона приймає іншу колекцію чи послідовність та повертає набір з об'єктів типу **Pair**.

Розглянемо приклад об'єднання двох списків в один:

```
val english = listOf("red", "blue", "green")
val ukrainian = listOf("червоний", "синій", "зелений")
val words = english.zip(ukrainian)

for(word in words)
    println("${word.first}: ${word.second}")
```

В цьому прикладі кожному елементу зі списку **english** зіставляється елемент відповідної позиції зі списку **ukrainian**. Результатом функції буде список об'єктів **Pair<String, String>**, де властивість **first** зберігає значення з поточної колекції, а властивість **second** зберігає значення з колекції, переданої як параметр. Результат роботи програми наведено на рисунку 7.10.

```
red: червоний
blue: синій
green: зелений
```

Рисунок 7.10. Результат роботи програми

Якщо в одній з колекцій менше елементів, ніж в іншій, то зіставляється стільки елементів, скільки міститься елементів у найменшій колекції.

Функція **zip()** має ще одну версію, яка як другий параметр отримує функцію перетворення, що застосовується до елементів обох колекцій чи послідовностей, а саме:

```
zip(other: Iterable<R>,transform: (a: T, b: R) -> V): List<V>
zip(other: Sequence<R>,transform: (a: T, b: R) -> V): Sequence<V>
```

Наведемо приклад використання цих функцій:

```
val english = listOf("red", "blue", "green")
val ukrainian = listOf("червоний", "синій", "зелений")
val words = english.zip(ukrainian){en, ua -> "$en - $ua"}
```

```
4
5   for(word in words) println(word)
```

Результат роботи програми наведено на рисунку 7.11.

```
red - червоний
blue - синій
green – зелений
```

Рисунок 7.11. Результат роботи програми

Також у мові Kotlin є зворотна функція **unzip**, яка дозволяє отримати дві колекції, а саме:

```
1   unzip(): Pair<List<T>, List<R>>
```

Наведемо приклад використання цієї функції:

```
1   val dictionary = listOf("red", "blue", "green")
2           .zip(listOf("червоний", "синій", "зелений"))
3   val words = dictionary.unzip()
4   println(words.first)     // [red, blue, green]
5   println(words.second)    // [червоний, синій, зелений]
```

7.14. Отримання частини елементів

7.14.1. Функція slice

Функція **slice()** повертає частину колекції, елементи якої розміщуються на певних індексах. Індекси передаються у функцію або у вигляді діапазону **IntRange** або колекції значень **Iterable<Int>**. Результатом функції є список **List<T>**, який містить елементи за вказаними індексами, а саме:

```
1   val people = listOf("Taras", "Volodymyr", "Stepan",
    "Kateryna", "Olena", "Myhailo")
2
3   println(people.slice(1..3))      // [Volodymyr, Stepan,
    Kateryna]
4   println(people.slice(0..5 step 2))  // [Taras, Stepan,
    Olena]
5   println(people.slice(listOf(1, 3, 5, 1)))    //
    [Volodymyr, Kateryna, Myhailo, Volodymyr]
```

Варто зазначити, що для послідовностей ця функція не доступна.

7.14.2. Функції Take та takeWhile

Функція **take()** вибирає з початку колекції чи послідовності певну кількість елементів, а саме:

```
take(n: Int): List<T>/Sequence<T>
```

Наприклад, виберемо 3 перших елементів:

```
val people = listOf("Taras", "Volodymyr", "Stepan",
"Kateryna", "Olena", "Myhailo")
println(people.take(3))     // ["Taras", "Volodymyr",
"Stepan"]
```

Для колекцій також доступна функція **takeLast()**, яка вибирає певну кількість елементів з кінця колекції, а саме:

```
val people = listOf("Taras", "Volodymyr", "Stepan",
"Kateryna", "Olena", "Myhailo")
// отримуємо три останніх елементи
println(people.takeLast(3))     // ["Kateryna",
"Olena", "Myhailo"]
```

Функція **takeWhile()** вибирає елементи з початку колекції чи послідовності, які відповідають умові предикату, а саме:

```
takeWhile(predicate: (T) -> Boolean):
List<T>       // для колекцій
takeWhile(predicate: (T) -> Boolean):
sequence<T>   // для послідовностей
```

Функція предикату отримує як параметр поточний елемент і повертає **true**, якщо елемент відповідає умові, наприклад:

```
val people = listOf("Taras", "Stepan", "Kateryna",
"Volodymyr", "Olena", "Myhailo")
val part = people.takeWhile{it.length == 3}
println(part)     // ["Taras", "Stepan"]
```

В наведеному прикладі умова предикату повертає **true**, якщо довжина рядка дорівнює 3: **{it.length == 3}**, тому функція **takeWhile** вибирає рядки доти, доки вони відповідають цій умові. Отже в цьому прикладі у результуючому списку виявляться перші два елементи: **["Taras", "Stepan"]**. Коли функція **takeWhile()** зустріне хоч один елемент, який не відповідає умові, вона завершить роботу. Хоча у списку **people** ще залишились елементи з довжиною у три символи, наприклад елемент "**Volodymyr**". Тому якщо у списку перший елемент не відповідає умові предикату, то функція поверне порожній список, а саме:

```
val people = listOf("Taras", "Stepan", "Kateryna",
"Volodymyr", "Olena", "Myhailo")
val part = people.takeWhile{it.length == 4}     //
вибираємо рядки з довжиною у 4 символи
println(part)     // [] - порожній список
```

Для колекцій також доступу функція **takeLastWhile()**, яка працює аналогічним чином, тільки вибирає елементи з кінця списку, наприклад:

```
val people = listOf("Taras", "Stepan", "Kateryna",
"Volodymyr", "Olena", "Myhailo")
```

```
2   val part = people.takeLastWhile{it.length !=3} //
    довжина НЕ дорівнює 3
3   println(part)      // ["Olena", "Myhailo"]
```

7.14.3. Функції drop та dropWhile

Функція **drop()** дозволяє пропустити певну кількість елементів у колекції чи послідовності, наприклад:

```
1   val people = listOf("Taras", "Stepan", "Kateryna",
    "Volodymyr", "Olena", "Myhailo")
2   val part = people.drop(3) // пропускаємо перші 3
    елементи
3   println(part)      // [Volodymyr, Olena, Myhailo]
```

Для колекцій також доступна функція **dropLast()**, яка пропускає певну кількість елементів з кінця колекції, наприклад:

```
1   val people = listOf("Taras", "Stepan", "Kateryna",
    "Volodymyr", "Olena", "Myhailo")
2   val part = people.dropLast(2) // пропускаємо останні 2
    елементи
3   println(part)      // [Taras, Stepan, Kateryna,
    Volodymyr]
```

Функція **dropWhile()** пропускає елементи з початку колекції чи послідовності, які відповідають умові предикату, наприклад:

```
1   dropWhile(predicate: (T) -> Boolean): List<T>        //
    для колекцій
2   dropWhile(predicate: (T) -> Boolean): Sequence<T>    //
    для послідовностей
```

Функція предикату отримує як параметр поточний елемент і повертає **true**, якщо елемент відповідає умові, наприклад:

```
1   val people = listOf("Taras", "Stepan", "Kateryna",
    "Volodymyr", "Olena", "Myhailo")
2   val part = people.dropWhile{it.length == 3}
3   println(part)      // [Kateryna, Volodymyr, Olena,
    Myhailo]
```

У цьому прикладі умова предикату повертає **true**, якщо довжина рядка дорівнює 3: **{it.length == 3}**, тому функція **dropWhile()** пропускає рядки доти, доки вони відповідають цій умові. Так, у наведеному прикладі в результуючому списку виявляться останні чотири елементи: **[Kateryna, Volodymyr, Olena, Myhailo]**. Коли функція **dropWhile()** зустріне хоч один елемент, який відповідає умові, вона завершить роботу. Хоча у списку **people** ще є елементи з довжиною у три символи, наприклад елемент "**Volodymyr**". Тому якщо в списку перший

елемент не відповідає умові предикату, то функція поверне список з усіма елементами.

Для колекцій також доступна функція **dropLastWhile()**, яка працює аналогічним чином, тільки пропускає елементи з кінця списку, наприклад:

```
val people = listOf("Taras", "Stepan", "Kateryna",
"Volodymyr", "Olena", "Myhailo")
val part = people.dropLastWhile{it.length !=3} //
довжина НЕ дорівнює 3
println(part)      // ["Taras", "Stepan", "Kateryna",
"Volodymyr"]
```

7.14.4. Поділ на частини

Функція **chuncked()** дозволяє розбити колекцію чи послідовність на списки певної довжини, наприклад:

```
chunked(size: Int): List<List<T>>        // для
колекцій
chunked(size: Int): Sequence<List<T>>  // для
послідовностей
```

Як параметр у функцію передається розмір списків, в які будуть розміщуватися елементи, наприклад:

```
val people = listOf("Taras", "Stepan", "Kateryna",
"Volodymyr", "Olena", "Myhailo")
val parts = people.chunked(3)
println(parts)      // [[Taras, Stepan, Kateryna],
[Volodymyr, Olena, Myhailo]]
```

У наведеному прикладі колекція розбивається на списки по 3 елементи, відповідно результуюча колекція міститиме два списки по три елементи.

Додаткова версія цієї функції як другий параметр приймає функцію перетворення елементів, наприклад:

```
chunked(size: Int, transform: (List<T>) -> R):
List<R>       // для колекцій
chunked(size: Int,transform: (List<T>) -> R):
Sequence<R> // для послідовностей
```

Тобто, спочатку колекція чи послідовність розбивається на кілька списків довжиною **size**, потім кожен список передається у функцію перетворення, де на його основі можна згенерувати нове значення, наприклад:

```
val people = listOf("Taras", "Stepan", "Kateryna",
"Volodymyr", "Olena", "Myhailo")
val parts = people.chunked(3){it.first()}
println(parts)      // [Taras, Volodymyr]
```

В цьому прикладі колекція розділяється на два списки завдовжки по 3 елементи. У функції перетворення з кожного списку вибирається та повертається перший елемент кожного списку. Відповідно результатом функції буде список із двох перших елементів.

7.15. Отримання окремих елементів

7.15.1. Отримання елемента за індексом

Для отримання елемента за індексом застосовується функція **elementAt()**, у яку передається індекс елемента, наприклад:

```
val people = listOf("Taras", "Stepan", "Kateryna",
"Volodymyr", "Olena")
// отримуємо елемент за індексом 1
println(people.elementAt(1))     // Stepan
```

Однак, якщо переданий індекс виходить за межі колекції чи послідовності, то програма згенерує помилку. У цьому випадку можна використовувати функцію **elementAtOrNull()**, яка повертає **null**, якщо індекс поза межами, наприклад:

```
val people = listOf("Taras", "Stepan", "Kateryna",
"Volodymyr", "Olena")
println(people.elementAtOrNull(1))     // Stepan
println(people.elementAtOrNull(6))     // null
```

Також можна використовувати функцію **elementAtOrElse()**, наприклад:

```
elementAtOrElse(index: Int, defaultValue: (Int) ->
T): T
```

Першим параметром цієї функції є індекс елемента, а другим параметром є функція, яка отримує індекс і повертає значення за замовчуванням, якщо індекс знаходиться поза межами колекції чи послідовності, наприклад:

```
val people = listOf("Taras", "Stepan", "Kateryna",
"Volodymyr", "Olena")

println(people.elementAtOrElse(1){"Не
визначено"})     // Stepan
println(people.elementAtOrElse(6){"Не
визначено"})     // Не визначено
println(people.elementAtOrElse(8){"Індекс $it поза
межами діапазону"})  // Індекс 8 поза межами діапазону
```

7.15.2. Отримання за умовою

Якщо потрібно отримати перший або останній елементи колекції чи послідовності, то можна використовувати функції **first()** і **last()**, наприклад:

```
val people = listOf("Taras", "Stepan", "Kateryna",
```

```
    "Volodymyr", "Olena")
2   println(people.first())      // Taras
3   println(people.last())      // Olena
```

У якості параметра ці функції можуть приймати функцію предикату, наприклад:

```
1   first(predicate: (T) -> Boolean): T
2   last(predicate: (T) -> Boolean): T
```

Ці функції повертають перший та останній елементи, які відповідають умові предикату, а саме:

```
1   val people = listOf("Taras", "Stepan", "Kateryna",
    "Volodymyr", "Olena")
2   // перший елемент з довжиною у 5 символів
3   println(people.first{it.length==5})      // Taras
4   // останній елемент з довжиною у 5 символи
5   println(people.last{it.length==5})      // Olena
```

Проте, якщо колекція порожня, або в колекції чи послідовності немає елементів, які відповідають умові, програма генерує помилку. У цьому випадку можна застосовувати функції **firstOrNull()** і **lastOrNull()**, які повертають **null**, якщо колекція порожня або в ній немає елементів, що відповідають умові, наприклад:

```
1   firstOrNull(): T?
2   firstOrNull(predicate: (T) -> Boolean): T?
3
4   lastOrNull(): T?
5   lastOrNull(predicate: (T) -> Boolean): T?
```

Наведемо приклад застосування цих функцій:

```
1   val people = listOf("Taras", "Stepan", "Kateryna",
    "Volodymyr", "Olena")
2
3   println(people.firstOrNull{it.length==5})      //
    Taras
4   println(people.firstOrNull{it.length==55})     // null
5
6   println(people.lastOrNull{it.length==5})      // Olena
7   println(people.lastOrNull{it.length==55})     // null
```

7.15.3. Вибір випадкового елемента

Для отримання випадкового елемента використовується функція **random()**, наприклад:

```
1   val people = listOf("Taras", "Stepan", "Kateryna",
    "Volodymyr", "Olena")
2   println(people.random())
```

Проте якщо колекція чи послідовність порожня, то ця функція генерує помилку. У цьому випадку можна використовувати функцію **randomOrNull()**, яка в такій ситуації повертає **null**, наприклад:

```
1   val people = listOf<String>()
2   println(people.randomOrNull())      // null
```

Резюме

В наведеному розділі розглядаються колекції та послідовності в мові Kotlin. Колекції поділяються на змінювані (**mutable**) і незмінювані (**immutable**), що впливає на можливість редагування їх елементів. Основні типи колекцій – це **List** (список), **Set** (множина унікальних значень) та **Map** (пари ключ-значення). Також описані послідовності (**sequences**), які відрізняються від колекцій тим, що обробляють дані поступово, без створення проміжних структур. Наведено багато операцій для роботи з колекціями та послідовностями, зокрема фільтрацію, сортування, перетворення та групування даних. Відмінність між **Iterable** та **Sequence** пояснюється через підхід до обробки даних: послідовності ефективніші для великих наборів даних, оскільки обробляють кожен елемент окремо.

Ознайомлення із цим розділом дозволить отримати знання: про колекції та послідовності; операції над ними; відмінностями між послідовностями та колекціями; механізмами фільтрації, трансформації та агрегації, що є передумовою для вивчення матеріалу, викладеного в наступних розділах, а також без чого не можливо створення мобільних, десктопних та веб-застосунків мовою Kotlin.

Контрольні запитання і завдання

1. Які основні типи колекцій існують у Kotlin?
2. Чим відрізняються змінювані (**mutable**) та незмінювані (**immutable**) колекції?
3. Як у Kotlin створити незмінюваний список (**List**) із початковими значеннями?
4. Які основні методи роботи зі списками (**List**) у Kotlin?
5. Чим відрізняється **Set** від **List**?
6. Як створити змінювану множину (**MutableSet**) у Kotlin?
7. Що таке **Map** у Kotlin, і як працює ця структура даних?
8. Як створити змінюваний словник (**MutableMap**) у Kotlin?
9. Чим **Map** відрізняється від **List** і **Set**?
10. Які основні операції можна виконувати над колекціями?

11. Що таке послідовності (**Sequence**) у Kotlin, і в чому їх перевага?
12. Як створити послідовність у Kotlin? Назвіть три способи.
13. У чому відмінність між **Iterable** та **Sequence**?
14. Як працює метод **filter()**, і що він робить?
15. Як можна відфільтрувати **null** значення у колекції?
16. Яка різниця між **map()** та **flatMap()**?
17. Як працює метод **groupBy()**, і для чого він використовується?
18. Як можна знайти перший елемент, що відповідає певній умові?
19. Які є способи сортування колекцій у Kotlin?
20. Чому використання **Sequence** може бути ефективнішим за **List** для великих наборів даних?
21. Створіть незмінюваний список чисел **listOf(3, 7, 12, 5, 9)**. Виведіть перший та останній елементи списку, а також кількість його елементів.
22. Створіть змінюваний список імен **mutableListOf("Анна", "Олег", "Марія")**. Додайте до нього ім'я «Тарас», видаліть «Олег» і виведіть оновлений список.
23. Створіть множину **setOf(2, 4, 6, 8, 4, 2)**. Виведіть її на екран. Чи є в множині повторювані значення?
24. Створіть колекцію **Map**, який міститиме назви країн та їхні столиці. Наприклад, **mapOf("Україна" to "Київ", "Франція" to "Париж")**. Виведіть столицю Франції.
25. Дано список **listOf(5, 12, 3, 18, 7, 21)**. Використовуючи метод **filter()**, створіть новий список, що містить лише парні числа.
26. Є список рядків **listOf("яблуко", "груша", "вишня")**. Перетворіть усі елементи у верхній регістр (**uppercase()**).
27. Створіть послідовність із чисел від 1 до 100. Відфільтруйте лише ті, що діляться на 5, і виведіть перші 5 таких чисел.
28. Дано список імен **listOf("Ганна", "Артем", "Олег", "Оксана", "Марія")**. Використайте **groupBy()**, щоб згрупувати імена за першою літерою.
29. Є список **listOf(2, 3, 4, 5)**. Використайте **reduce()**, щоб знайти добуток усіх чисел у списку.
30. Створіть змінюваний словник **mutableMapOf("a" to 1, "b" to 2)**. Додайте пару **"c" to 3**, змініть значення **"b"** на 10 і виведіть оновлений словник.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Який метод використовується для створення незмінюваного списку в Kotlin?

a) mutableListOf()

b) listOf()

c) arrayListOf()

d) setOf()

Правильна відповідь: b).

Яке твердження правильне щодо Sequence у Kotlin?

a) Sequence виконує всі операції одразу над усією колекцією

b) Sequence обробляє елементи по одному і не створює проміжні колекції

c) Sequence працює лише з унікальними значеннями

d) Sequence підтримує лише числові значення

Правильна відповідь: b).

Закриті запитання з кількома правильними відповідями

Які з наведених методів дозволяють додати елемент у змінювану колекцію?

a) add(element)

b) addAll(elements)

c) get(index)

d) contains(element)

Правильна відповідь: a). b).

Завдання на відповідність

Співвіднесіть тип колекції з її особливістю:

a) List → 1. Дозволяє дублікати, впорядкована

b) Set → 2. Не дозволяє дублікати, невпорядкована

c) Map → 3. Використовує пари ключ-значення

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

Вставити пропущені слова

У Kotlin для створення змінюваного набору (Set) використовується функція _____.

Правильна відповідь: mutableSetOf()

Завдання з кодом

Що виведе цей код?

```
val numbers = listOf(1, 2, 3, 4, 5)
println(numbers.elementAtOrNull(6))
```

a) IndexOutOfBoundsException

b) 6

c) null

d) 5

Правильна відповідь: c).

Який результат виведе наступний код?

```
val numbers = listOf(1, 2, 3, 4, 5)
val result = numbers.map { it * 2 }
println(result)
```

a) [1, 2, 3, 4, 5]

b) [2, 3, 4, 5, 6]

c) [2, 4, 6, 8, 10]

d) [2, 4, 6, 8]

Правильна відповідь: c).

Що поверне цей код?

```
val numbers = listOf(1, 2, 3, 4, 5)
val filtered = numbers.filter { it % 2 == 0 }
println(filtered)
```

a) [2, 4]

b) [1, 3, 5]

c) [1, 2, 3, 4, 5]

d) null

Правильна відповідь: a).

Визначте результат коду

```
val numbers = listOf(3, 6, 9)
val sum = numbers.sum()
```

```
println(sum)
```

a) 3

b) 6

c) 18

d) 9

Правильна відповідь: c).

Завдання з вибором істинності

Ключі у Мар можуть повторюватися.

a) Так

b) Ні

Правильна відповідь: b).

8. КОРУТИНИ

8.1. Введення в корутини

Останнім часом підтримка асинхронності та паралельних обчислень стала невід'ємною рисою багатьох мов програмування. І мова Kotlin не є винятком. В цьому розділі розглянемо навіщо потрібні асинхронність та паралельні обчислення.

Насамперед, паралельні обчислення дозволяють виконувати кілька завдань одночасно, а асинхронність дозволяє не блокувати основний хід перебігу програми під час виконання завдання, яке займає тривалий час. Наприклад, необхідно створити графічний додаток для робочого столу або мобільного пристрою. І необхідно з натисканням на кнопку надсилати запит до Інтернет-ресурсу. Однак такий запит може зайняти досить багато часу. І щоб програма не зависала на період відправки запиту, подібні запити до Інтернет-ресурсів слід надсилати асинхронно. При асинхронних запитах користувач не чекає, поки прийде відповідь від Інтернет-ресурсу, а продовжує роботу з додатком, і при отриманні відповіді отримає відповідне повідомлення.

У мові Kotlin підтримка асинхронності та паралельних обчислень втілена як (**coroutine**). По суті корутин представляє блок коду, який може виконуватися паралельно з рештою коду. Базова функціональність, що пов'язана з корутинами, зосереджена у бібліотеці **kotlinx.coroutines**.

В наступній частині розглянемо оголошення та застосування корутин на деяких прикладах.

8.1.1. Додавання бібліотеки kotlinx.coroutines

Насамперед варто відзначити, що функціональність корутин, а саме бібліотека **kotlinx.coroutines,** за замовчуванням не включена до проєкту. Отже її необхідно додати. Якщо створюється проєкт консольної програми в **IntelliJ IDEA**, то можна додати відповідну бібліотеку до проєкту. Для цього в меню «**File**» необхідно звернутись до пункту «**Project Structure..**», як показано на рисунку 8.1.

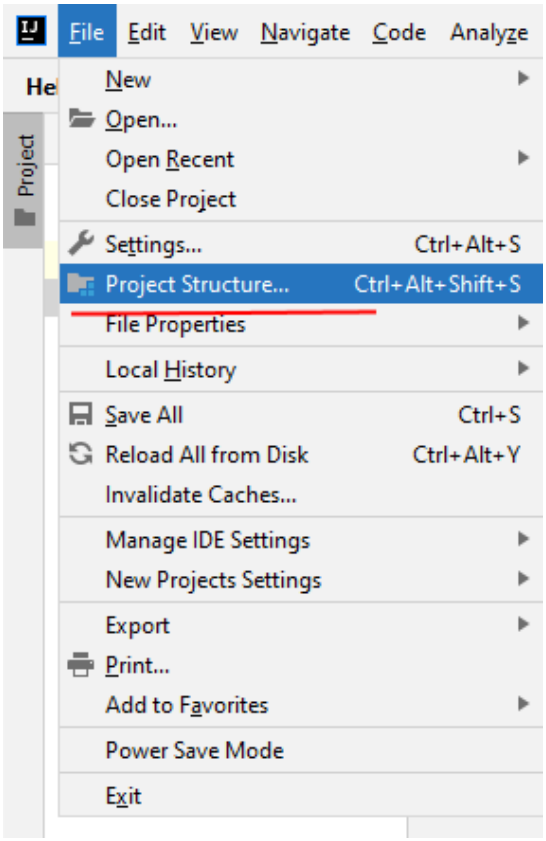

Рисунок 8.1. Вікно додавання нової бібліотеки

Далі на вкладці «**Project Settings**» необхідно перейти до пункту «**Libraries**». У центральному полі відображаються бібліотеки, які вже додані до проєкту. А для додавання нової бібліотеки необхідно натиснути на знак плюс «+», як показано на рисунку 8.2.

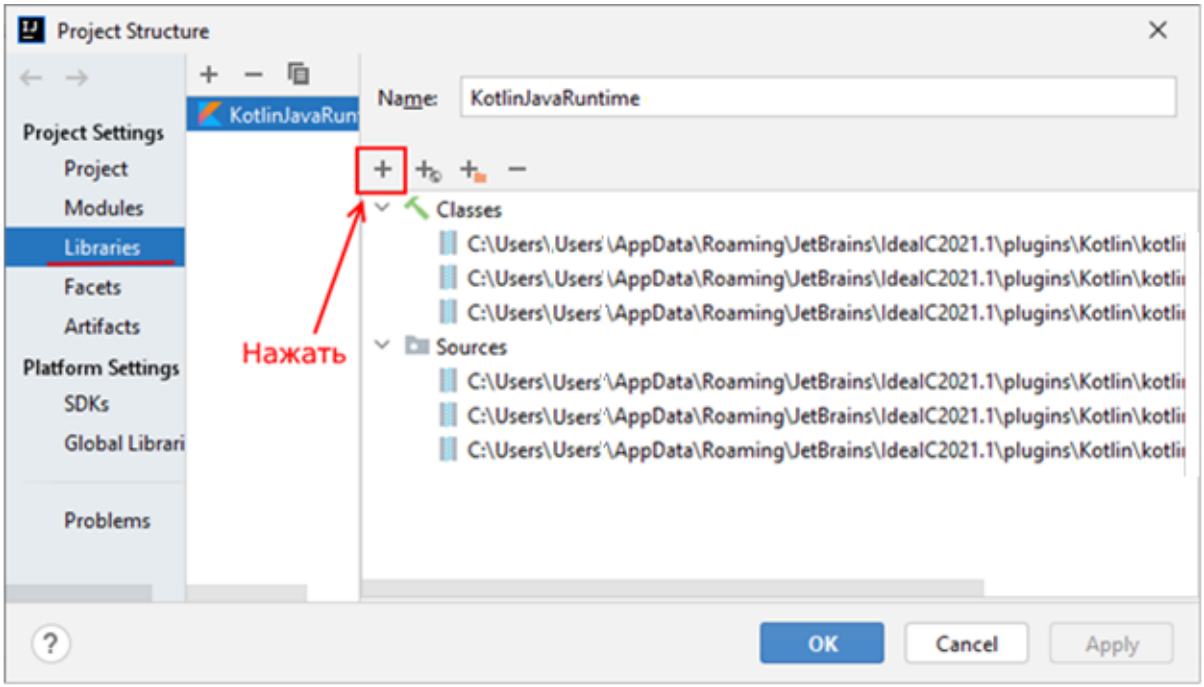

Рисунок 8.2. Вікно вибору нової бібліотеки

Після цього відкриється тека **lib**, яка містить бібліотеки Kotlin, які встановлюються за замовчуванням. Виберемо серед них бібліотеку **kotlinx-coroutines-core-jvm.jar** та натискаємо на кнопку «**OK**» для її додавання, як показано на рисунку 8.3.

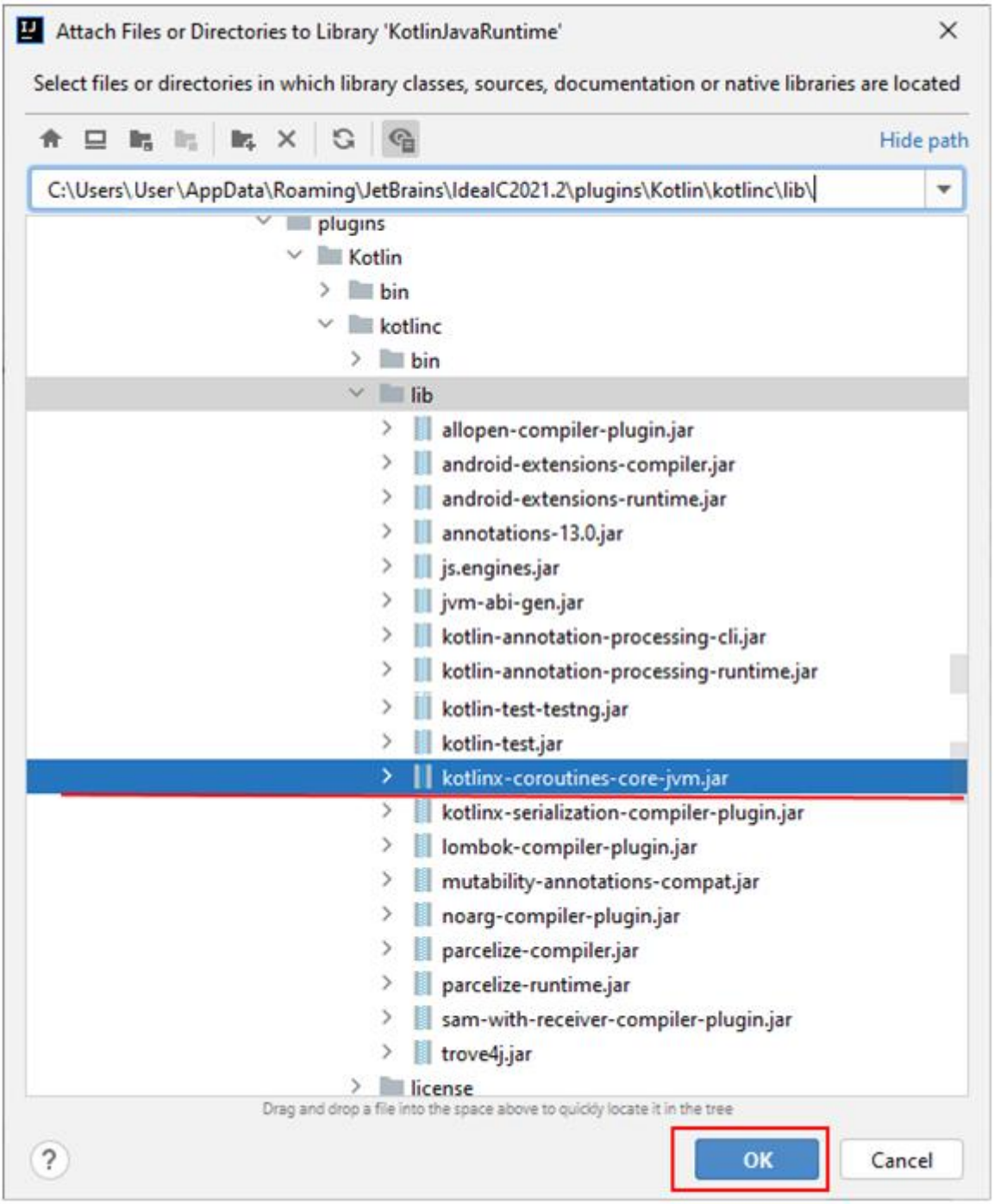

Рисунок 8.3. Вікно завершення додавання нової бібліотеки

Після додавання до проєкту у вузлі **External Libraries / KotlinJavaRuntime** буде відображено додану бібліотеку, як показано на рисунку 8.4.

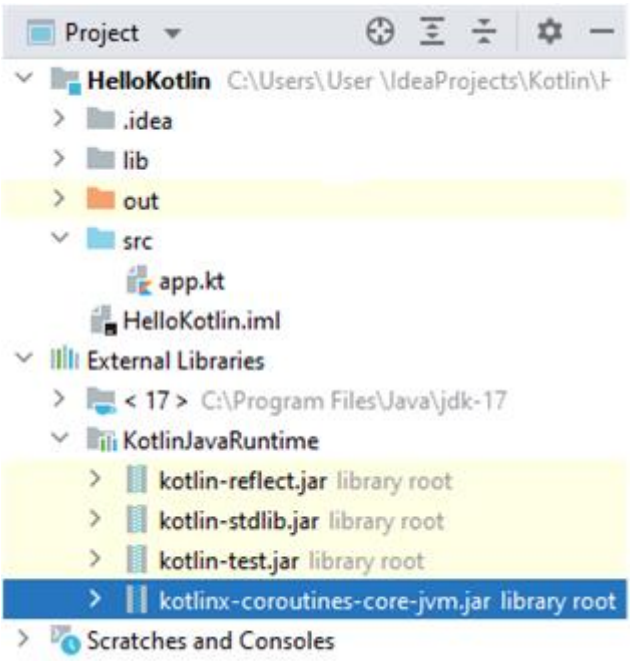

Рисунок 8.4. Відображення вдалого додавання бібліотеки

В інших типах проєктів для підключення **kotlinx.coroutines** можна використовувати **Gradle** або інші способи. Так, якщо проєкт використовує **Gradle**, то у файл **gradle** додається залежність, наприклад:

```
1   dependencies {
2       implementation( "org.jetbrains.kotlinx:kotlinx-
    coroutines-core:1.5.2" )
3   }
```

8.1.2. Оголошення suspend-функції

Для кращого розуміння призначення корутин, спочатку розглянемо приклад, який не використовує корутини, а саме:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main(){
4       for (i in 0..5) {
5           delay(400L)
6           println(i)
7       }
8
9       println( "Привіт Coroutines" )
10  }
```

В цьому програмному коді у функції **main** перебирається послідовність від 0 до 5 і виводиться поточний елемент послідовності на консоль. Для імітації тривалої роботи всередині циклу викликається спеціальна функція **delay()** пакета **kotlinx.coroutines**. У цю функцію передається кількість мілісекунд, на яку

виконується затримка. Значення, що передається, повинно мати тип **Long**. Тобто в наведеному прикладі функція здійснюватиме затримку 400 мілісекунд перед виведенням на консоль поточного елемента послідовності.

Далі, після виконання роботи циклу виводиться на консоль рядок «**Hello Coroutines**».

Аби стало можливим використання всередині функції **main** функції **delay()**, функції **main** передує модифікатором **suspend**. Модифікатор **suspend** визначає функцію, яка може призупиняти виконання та відновлювати його через деякий період часу.

Сама по собі функція **delay()** теж є подібною функцією, яка визначена з модифікатором **suspend**. Варто зазначити, що будь-яка функція з модифікатором **suspend** може викликатися тільки з іншої функції, яка теж має модифікатор **suspend**, або з корутини.

Якщо запустити цю програму на виконання, то побачимо результат, який наведено на рисунку 8.5.

Рисунок 8.5. Результат роботи програми

На цьому прикладі видно, що рядок «**Привіт Coroutines**» виводиться після виконання всього циклу. Але замість циклу могла би бути більш змістовна та й більш тривала робота, наприклад: доступ до Інтернет-ресурсу чи віддаленої бази даних; деякі операції з файлами; тощо. У таких випадках всі операції, зазначені після них, будуть очікувати завершення всієї довготривалої роботи, саме так, як в наведеному прикладі операція друку рядка «**Привіт Coroutines**» на консоль, чекає закінчення всього циклу.

8.1.3. Оголошення корутини

В наступному прикладі додамо імітатор довготривалої роботи, тобто цикл у корутину:

```
1   import kotlinx.coroutines.*
2
```

```
3   suspend fun main() = coroutineScope{
4       launch {
5               for (i in 0..5) {
6               delay(400L)
7               println(i)
8           }
9       }
10
11      println( "Привіт Coroutines" )
12  }
```

Насамперед при оголошенні і виконанні корутини для неї необхідно визначити контекст, оскільки корутина може викликатися лише у контексті корутини (**coroutine scope**). Для цього застосовується функція **coroutineScope()**, яка створює контекст корутини. Крім того, ця функція очікує на виконання всіх визначених усередині неї корутин. Варто зазначити, що функція **coroutineScope()** може бути використана тільки у функції з модифікатором, якою є функція **main**.

Сама корутина оголошується і запускається за допомогою будівельника корутин функції **launch**. Ця корутин функція створює корутину у вигляді блоку коду, який в наведеному вище прикладі мав наступний вигляд:

```
1   {
2       for (i in 0..5) {
3           delay(400L)
4           println(i)
5       }
6   }
```

та запускає цю корутину паралельно з рештою коду. Тобто ця корутина виконується незалежно від іншого коду, що визначено у функції **main**.

Як наслідок, під час виконання програми отримується дещо інший результат роботи програми, а саме такий як наведено на рисунку 8.6.

```
Привіт Coroutines
0
1
2
3
4
5
```

Рисунок 8.6. Результат роботи програми

Тепер рядок «**Привіт Coroutines**» не чекає, доки завершиться цикл, а виконується паралельно з ним.

8.1.4. Винесення коду корутин в окрему функцію

У попередньому прикладі код корутини розташовувався у функції **main**. Але можна визначити його у вигляді окремої функції та викликати в корутині цю функцію. Розглянемо наступний приклад:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main()= coroutineScope{
4       launch{ doWork() }
5
6       println( "Привіт Coroutines" )
7   }
8   suspend fun doWork(){
9       for (i in 0..5) {
10          println(i)
11          delay(400L)
12      }
13  }
```

У цьому прикладі основний код корутини винесено у функцію **doWork()**. Оскільки в цій функції застосовується функція **delay()**, то **doWork()** визначена з модифікатором **suspend**. Сама корутина створюється також за допомогою функції **launch()**, яка викликає функцію **doWork()**.

8.1.5. Корутини та потоки

У деяких мовах програмування є такі структури, які дозволяють використовувати потоки. Проте між корутинами та потоками немає прямої відповідності. Корутіна не прив'язана до конкретного потоку. Вона може призупинити виконання в одному потоці, а відновити виконання в іншому.

Коли корутина зупиняє виконання, наприклад, як у наведеному вище прикладі при виклику затримки за допомогою функції **delay()**, ця корутина звільняє потік, в якому вона виконувалася, і зберігається в пам'яті. А звільнений потік може бути задіяний для виконання інших завдань. Коли ж завершується запущене завдання (наприклад, виконання функції **delay()**), то корутин відновлює свою роботу в одному із вільних потоків.

8.2. Область корутини

Корутина може виконуватися лише у певній області корутини (**coroutine scope**). Область корутин представляє простір, в якому діють корутини, де є певний життєвий цикл і де вони управляють життєвим циклом створюваних у ній корутин.

Для створення області корутину у мові Kotlin може використовуватися низка функцій, які створюють об'єкт інтерфейсу **CoroutineScope**. Однією з функцій є **coroutineScope**. Вона може застосовуватися до будь-якої функції, наприклад:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main(){
4
5       doWork()
6
7       println( "Привіт Coroutines" )
8   }
9   suspend fun doWork()= coroutineScope{
10      launch {
11          for (i in 0..5) {
12              println(i)
13              delay(400L)
14          }
15      }
16  }
```

8.2.1. Запуск кількох корутин

Розглянемо, яким чином можна запускати в одній функції кілька корутин, де вони виконуватимуться одночасно, а саме:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main()= coroutineScope{
4
5       launch {
6           for (i in 0..5) {
7               delay(400L)
8               println(i)
9           }
10      }
11      launch {
12          for (i in 6..10) {
13              delay(400L)
```

```
14              println(i)
15          }
16      }
17
18      println( "Привіт Coroutines" )
19  }
```

Функція **coroutineScope()**, яка створює область корутин, чекатиме завершення всіх визначених у цій області корутин. Тобто функція **main** завершить виконання тоді, коли будуть завершені обидві корутини.

В наведеному прикладі буде отримано результат, який наведено на рисунку 8.7. При цьому варто зауважити, що результат є недетермінованим, тобто при кожному запуску програми на виконання результат може змінюватись.

```
Привіт Coroutines
6
0
7
1
8
2
9
3
10
4
5
```

Рисунок 8.7. Результат роботи програми

8.2.2. Вкладені корутини

Одна корутин може містити інші корутини. Розглянемо наступний приклад:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       launch {
6           println( "Outer coroutine" )
7           launch {
8               println( "Inner coroutine" )
9               delay(400L)
10          }
11      }
```

```
12
13      println( "End of Main" )
14  }
```

Подібним чином зовнішні корутини визначають область для вкладених корутин та керують їх життєвим циклом.

8.3. Функція launch та об'єкт Job

Для створення корутини необхідний будівельник корутини. Одним з будівельників корутини у пакеті **kotlinx.coroutines** є функція **launch**. Раніше вже розглядалось те, як створювати корутини за допомогою **launch**. Тепер розглянемо деякі аспекти більш докладно.

Насамперед, **launch()**, як правило, застосовується, коли відсутня необхідність у поверненні результату з корутини і коли є необхідність виконання одночасно з іншим кодом.

8.3.1. Об'єкт Job

Будівельник корутин **launch** повертає об'єкт **Job**, за допомогою якого можна керувати запущеною корутиною, а саме:

```
1   Val job: Job = launch {
2       println( "Some coroutin" )
3       delay(400L)
4   }
```

Наприклад, його метод **join()** дозволяє чекати доди, доки корутина не завершиться. Розглянемо наступний приклад:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       launch {
6           for (i in 1..5) {
7               println(i)
8               delay(400L)
9           }
10      }
11
12      println( "Start" )
13      println( "End" )
14  }
```

Результат роботи програми, яка наведена вище, наведено на рисунку 8.8.

```
Start
End
1
2
3
4
5
```

Рисунок 8.8. Результат роботи програми

Тепер явним чином застосуємо інтерфейс **Job**, а саме:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       val job = launch {
6           for (i in 1..5) {
7               println(i)
8               delay(400L)
9           }
10      }
11
12      println( "Start" )
13      job.join() // очікуємо завершення корутини
14      println( "End" )
15  }
```

В цьому прикладі, корутина також запускається за допомогою **launch**, проте завдяки методу **join()** отриманого об'єкта **Job** функція зупинить виконання і чекатиме завершення корутини і лише після її завершення продовжить роботу. Відповідно в цьому разі результат буде іншим, а саме як наведено на рисунку 8.9.

```
1
2
3
4
5
End
```

Рисунок 8.9. Результат роботи програми

8.3.2. Відкладене виконання

За замовчуванням будівельник корутин **launch** створює і відразу ж запускає корутину. Однак мова Kotlin також дозволяє застосовувати техніку відкладеного запуску корутини (**lazy-запуск**), при якому корутина запускається при виклику методу **start()** об'єкта **Job**.

Для створення відкладеного запуску у функцію **launch()** передається значення **start = CoroutineStart.LAZY**.

Щоб побачити різницю, спочатку розглянемо корутину зі стандартним виконанням, а саме:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       // Корутина створена та запущена
6       launch( ) {
7           delay(200L)
8           println("Coroutine has started")
9       }
10
11      delay(1000L)
12      println("Other actions in main method")
13  }
```

Щоб дозволити корутині встигнути виконатися раніше інших операцій у коді програми, в методі **main**, після визначення корутини встановлено затримку у 1 секунду. Результат роботи програми наведено на рисунку 8.10.

```
Coroutine has started
Інші actions in main method
```

Рисунок 8.10. Результат роботи програми

Тепер розглянемо застосування відкладеного виконання, а саме:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       // Корутина створена, але не запущена
6       val job = launch(start = CoroutineStart.LAZY) {
7           delay(200L)
8           println("Coroutine has started")
9       }
10
```

```
11      delay(1000L)
12      job.start() // запускаємо корутину
13      println("Other actions in main method")
14  }
```

Тепер корутина тільки створюється за допомогою функції **launch**, але безпосередньо вона запускається тільки при виклику методу **job.start()**, відповідно буде отримаємо інший результат роботи програми, а саме як наведено на рисунку 8.11.

```
Other actions in main method
Coroutine has started
```

Рисунок 8.11. Результат роботи програми

8.4. Функції async, await та Deferred

Поряд з будівельником корутин **launch** в пакеті **kotlinx.coroutines** є ще один будівельник корутини, а саме функція **async**. Ця функція використовується, коли необхідно отримати якийсь результат від корутини.

Функція **Async** запускає окрему корутину, яка виконується паралельно з іншими корутинами. Окрім того, вона повертає об'єкт **Deferred**, який очікує на отримання результату корутини. Інтерфейс **Deferred** успадкований від інтерфейсу **Job**, тому йому також доступний увесь функціонал, визначений для інтерфейсу **Job**.

Для отримання результату з об'єкта **Deferred** використовується функція **await()**. Розглянемо наступний приклад:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       val message:     Deferred<String>     =     async{
    getMessage()}
6       println( "message: ${message.await()}" )
7       println( "Program has finished" )
8   }
9   suspend fun getMessage() : String{
10      delay(500L) // імітація тривалої роботи
11      return "Hello"
12  }
```

В наведеному прикладі для імітації тривалої роботи визначено функцію **getMessage()**, яка повертає рядок.

За допомогою функції **async** запускається корутина, яка виконує цю функцію, а саме:

```
1   async{ getMessage()}
```

Оскільки функція **getMessage()** повертає об'єкт типу **String**, то об'єкт, що повертається корутиною, представляє тип **Deferred<String>**. Адже об'єкт **Deferred** типізується тим типом функції, що повертається, тобто в наведеному прикладі типом String, наприклад.

```
1   val message: Deferred<String> = async{ getMessage()}
```

Далі об'єкт **Deferred** для отримання результату функції **getMessage()** викликає метод **await()**. Він очікує, доки не буде отримано результат. Результат роботи програми наведено на рисунку 8.12.

```
message: Hello
Program has finished
```

Рисунок 8.12. Результат роботи програми

Оскільки функція **getMessage()** повертає об'єкт типу **String**, то метод **await()** в цьому разі також повертатиме рядок, який можна, наприклад, присвоїти іншій змінній, наприклад:

```
1   val text: String = message.await()
```

При цьому можна за допомогою **async** запустити кілька корутин, які будуть виконуватись паралельно, наприклад:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       val numDeferred1 = async{ sum(1, 2)}
6       val numDeferred2 = async{ sum(3, 4)}
7       val numDeferred3 = async{ sum(5, 6)}
8       val num1 = numDeferred1.await()
9       val num2 = numDeferred2.await()
10      val num3 = numDeferred3.await()
11
12      println( "number1:  $num1 number2:  $num2 number3:
    $num3" )
13  }
14  suspend fun sum(a: Int, b: Int) : Int{
15      delay(500L) // імітація тривалої роботи
16      return a + b
17  }
```

В наведеному вище прикладі, запускається три корутини, кожна з яких виконує функцію **sum()**. Ця функція складає два числа і повертає їх суму як об'єкт

Int. Тому корутини повертають об'єкт **Deferred<Int>**. Відповідно виклик методу **await()** цього об'єкта поверне об'єкт **Int**, тобто суму двох чисел. При цьому, всі три корутині будуть запущені одночасно. Наприклад, на скріншоті відладчика корутин, який наведено на рисунку 8.13, видно, що дві корутини вже працюють (або перебувають у стані **Running**), і ще одна корутина тільки но створена і чекає на запуск (стан **Created**).

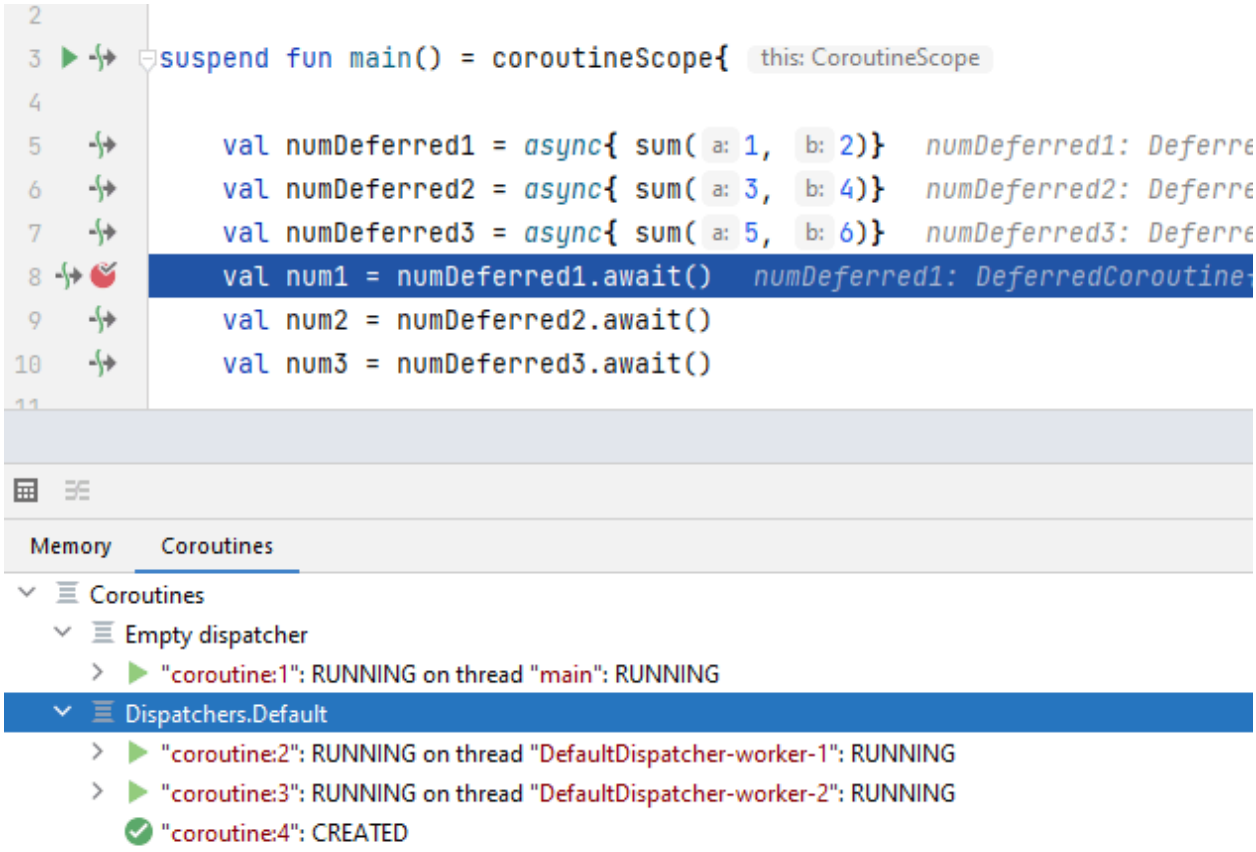

Рисунок 8.13. Вигляд вікна відладчика корутин

8.4.1. Відкладений запуск

За замовчуванням будівельник корутин **async** створює і відразу ж запускає корутину. Але як і при створенні корутин за допомогою **launch** для **async-корутин** можна застосовувати техніку відкладеного запуску. Тільки в цьому випадку корутина запускається не тільки при виклику методу **start** об'єкта **Deferred**, який успадковується від інтерфейсу **Job**, але також і за допомогою методу **await()** при зверненні до результату корутини, наприклад:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       // Корутина створена, але не запущена
```

```
6       val sum   =   async(start   =   CoroutineStart.LAZY){
    sum(1, 2)}
7
8       delay(1000L)
9       println( "Actions after the coroutine creation" )
10      println( "sum:   ${sum.await()}" ) //   запуск   та
    виконання корутини
11  }
12  fun sum(a: Int, b: Int) : Int{
13      println( "Coroutine has started" )
14      return a + b
15  }
```

Результа роботи програми наведено на рисунку 8.14.

```
Actions after the coroutine creation
Coroutine has started
sum: 3
```

Рисунок 8.14. Результат роботи програми

Якщо необхідно, щоб корутина ще до методу **await()** почала виконуватися, то можна застосувати метод **start()**, наприклад:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       // Корутина створена, але не запущена
6       val sum   =   async(start   =   CoroutineStart.LAZY){
    sum(1, 2)}
7
8       delay(1000L)
9       println( "Actions after the coroutine creation" )
10      sum.start() // запуск корутини
11      println( "sum:   ${sum.await()}" ) //   отримуємо
    результат
12  }
13  fun sum(a: Int, b: Int) : Int{
14      println( "Coroutine has started" )
15      return a + b
16  }
```

8.5. Диспетчер корутини

Контекст корутини включає у себе такий елемент як диспетчер корутини. Диспетчер корутини визначає, який потік або які потоки будуть використовуватися для виконання корутини.

Всі будівельники корутини, зокрема, функції **launch** і **async** як необов'язковий параметр приймають об'єкт типу **CoroutineContext**, який може використовуватися для визначення диспетчера створюваної корутини.

Коли функція **launch** викликається без параметрів, вона отримує контекст, у якому створюється і запускається. Розглянемо приклад використання методу **Thread.currentThread()**, який надає **JDK**, щоб отримати дані потоку корутини:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       launch {
6           println("Корутина виконується на потоці: ${Thread.currentThread().name}")
7       }
8       println("Функція main виконується на потоці: ${Thread.currentThread().name}")
9   }
```

В наведеному вище фрагменті програмного коду використовуючи змінну **Thread.currentThread().Name**, можна отримати назву потоку. Результа роботи програми наведено на рисунку 8.15.

```
Функція main виконується на потоці: main
Корутина виконується на потоці: DefaultDispatcher-worker-1
```

Рисунок 8.15. Результат роботи програми

Отже бачимо, що функція **main** виконується на потоці під назвою **"main"**, який власне і відведений для виконання цієї функції, а корутина виконується на іншому потоці під назвою **DefaultDispatcher-worker-1**. Якщо звернутися до відладчика корутин, то можна побачити диспетчер, який використовується корутиною, як наведено на рисунку 8.16.

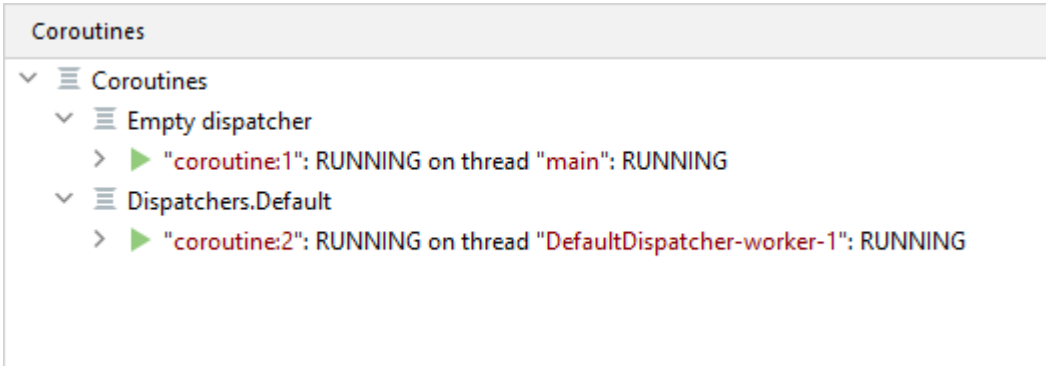

Рисунок 8.16. Вікно перегляду диспетчера, який використовується корутиною

Тут видно, що корутина, яка виконується в потоці **"DefaultDispatcher-worker-1"**, застосовує диспетчер **Dispatcher.Default**.

В наведеному прикладі контекст корутини у функції **main** створюється за допомогою функції **coroutineScope**, яка встановлює диспетчер за замовчуванням типу **Dispatcher.Default** для згенерованих корутин. Тому корутина, яка визначена у прикладі вище, бере на себе цей контекст разом із диспетчером цього типу.

Розглянемо, що це означає. Для цього розглянемо доступні типи диспетчерів:

- **Dispatchers.Default**: застосовується за замовчуванням, якщо тип диспетчера не вказано явно. Цей тип використовує загальний пул фонових потоків, що розділяються, і підходить для обчислень, які не працюють з операціями введення-виведення (операціями з файлами, базами даних, мережею) і які вимагають інтенсивного споживання ресурсів центрального процесора;
- **Dispatchers.IO**: використовує загальний пул потоків, створюваних у міру необхідності, і призначений для виконання операцій введення-виводу (наприклад, операції з файлами або запитами мережі);
- **Dispatchers.Main**: застосовується у графічних програмах, наприклад, у програмах Android або JavaFX;
- **Dispatchers.Unconfined**: корутина не закріплена чітко за певним потоком або пулом потоків. Вона запускається в поточному потоці до першого призупинення. Після відновлення роботи корутина продовжує роботу в одному з потоків, який суворо не фіксований. Розробники мови Kotlin у звичайній ситуації не рекомендують використовувати цей тип;
- **newSingleThreadContext** та **newFixedThreadPoolContext**: дозволяють вручну задати потік або пул для виконання корутини.

Також можна для корутини самостійно створити власний диспетчер, передавши у функцію **launch** (а також **async**) відповідне значення, як наведено у прикладі нижче:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       launch(Dispatchers.Default) {// явно визначаємо
    диспетчер Dispatcher.Default
6           println("Корутина виконується на потоці:
    ${Thread.currentThread().name}")
7       }
8       println("Функція main виконується на потоці:
    ${Thread.currentThread().name}")
9   }
```

8.5.1. Диспетчер типу Dispatchers.Unconfined

Диспетчер типу **Dispatchers.Unconfined** запускає корутину в поточному потоці, до першого призупинення. Після відновлення корутина продовжує роботу в одному з потоків, який суворо не фіксований. Подібний тип диспетчера підходить для корутин, яким не потрібно інтенсивно споживати час CPU або працювати із загальними даними, на зразок об'єктів інтерфейсу користувача. Розглянемо приклад застосування цього типу диспетчера:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       launch(Dispatchers.Unconfined) {
6           println("Потік корутини (до зупинки):
    ${Thread.currentThread().name}")
7           delay(500L)
8           println("Потік корутини (після зупинки):
    ${Thread.currentThread().name}")
9       }
10
11      println("Потік функції main:
    ${Thread.currentThread().name}")
12  }
```

Результат роботи програми наведено на рисунку 8.17.

```
Потік корутини (до зупинки): main
Потік функції main: main
Потік корутини (після зупинки): kotlinx.coroutines.DefaultExecutor
```

Рисунок 8.17. Результат роботи програми

8.5.2. Диспетчер типу newSingleThreadContext

Диспетчер типу **newSingleThreadContext** вручну запускає потік із зазначеним ім'ям, а саме:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       launch(newSingleThreadContext("Custaras Thread")) {
            println("Потік корутини:
6   ${Thread.currentThread().name}")
7       }
8
9       println("Потік функції main:
    ${Thread.currentThread().name}")
10  }
```

В наведеному програмному коді для виконання корутини запускатиметься потік з ім'ям **"Custaras Thread"**. Результат роботи програми наведено на рисунку 8.18.

```
Потік функції main: main
Потік корутини: Custaras Thread
```

Рисунок 8.18. Результат роботи програми

У той самий час виділений потік є досить затратним ресурсом. І в реальному додатку подібний потік слід або звільняти за допомогою функції **close()**, якщо він більше не потрібен, або зберігати у глобальній змінній та використовувати його повторно для подібних завдань протягом роботи програми.

8.6. Скасування виконання корутин

Під час роботи програми може скластися необхідність скасувати виконання корутини. Наприклад, у мобільному додатку запущена корутина для завантаження даних із деякого Інтернет-ресурсу, але користувач вирішив перейти до іншої сторінки програми, і йому більше не потрібні ці дані. У цьому випадку,

щоб даремно не витрачати ресурси мобільного пристрою, можна передбачити скасування виконання корутини.

Для скасування виконання корутини в об'єкті **Job** може бути застосовано метод **cancel()**, наприклад:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       val downloader: Job = launch {
6           println( "Починаємо завантаження файлів" )
7           for (i in 1..5) {
8               println( "Завантажений файл $i" )
9               delay(500L)
10          }
11      }
12      delay(800L) // встановимо затримку, щоб кілька
    файлів завантажилися
13      println("Набридло чекати, поки всі файли
    завантажаться. Перерву я завантаження...")
14      downloader.cancel() // скасовуємо корутину
15      downloader.join() //     очікуємо     завершення
    корутини
16      println("Робота програми завершена")
17  }
```

У цьому прикладі визначено корутину, яка імітує завантаження файлів. У циклі пробігаємося від 1 до 5 та умовно завантажується п'ять файлів.

Далі виклик методу **downloader.cancel()** сигналізує корутину, що треба перервати виконання. Потім за допомогою методу **join()** очікується завершення корутини, яка перервана. Результат роботи програми наведено на рисунку 8.19.

```
Починаємо завантаження файлів
Завантажений файл 1
Завантажений файл 2
Набридло чекати, поки всі файли завантажаться. Перерву я завантаження...
Робота програми завершена
```

Рисунок 8.19. Результат роботи програми

Також замість двох методів **cancel()** та **join()** можна використовувати один збірний метод **cancelAndJoin()**, наприклад:

```
1   import kotlinx.coroutines.*
2
```

```
3   suspend fun main() = coroutineScope{
4
5       val downloader: Job = launch {
6           println("Починаємо завантаження файлів")
7           for (i in 1..5) {
8               println("Завантажений файл $i")
9               delay(500L)
10          }
11      }
12      delay(800L)
13      println("Набридло чекати, поки всі файли
    завантажаться. Перерву я завантаження...")
14      downloader.cancelAndJoin() // скасовуємо корутину і
    очікуємо на її завершення
15      println( "Робота програми завершена")
16  }
```

8.6.1. Обробка винятку CancellationException

Всі **suspend-функції** в пакеті **kotlinx.coroutines** є такими, що можуть бути перервані (**cancellable**). Це означає, що вони перевіряють, чи перервана корутина. І якщо її виконання перервано, вони генерують виняток типу **CancellationException**. В самій корутині можна перехопити цей виняток, щоб обробити скасування корутини. Розглянемо наступний приклад:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       val downloader: Job = launch {
6           try {
7               println("Починаємо завантаження файлів")
8               for (i in 1..5) {
9                   println("Завантажений файл $i")
10                  delay(500L)
11              }
12          }
13          catch (e: CancellationException) {
                println("Завантаження файлів перервано")
14          }
15          finally {
16              println("Завантаження завершено")
17          }
18      }
19      delay(800L)
        println("Набридло чекати. Перерву я
```

```
20 завантаження...")
21     downloader.cancelAndJoin() // скасовуємо корутину
22 і очікуємо на її завершення
23     println("Робота програми завершена")
24 }
```

В цьому фрагменті програмного коду код виконання корутини обернений у конструкцію **try**. Якщо корутина буде перервана ззовні, то за допомогою блоку **catch** та перехоплення винятку **CancellationException** можна обробити скасування корутини.

Якщо ж необхідно виконати деякі завершальні дії, наприклад, звільнити ресурси, що використовуються в корутині, наприклад закрити файли, різні підключення до зовнішніх ресурсів, то це можна зробити у блоці **finally**. В наведеному прикладі в цьому блоці виводиться діагностичне повідомлення.

У результаті при виклику методу **downloader.cancel()** буде здійснено скасування корутини. Буде згенеровано виняток, і в корутині у блоці **catch** стане можливим його обробка. Результат роботи програми наведено на рисунку 8.20.

```
Починаємо завантаження файлів
Завантажений файл 1
Завантажений файл 2
Набридло чекати. Перерву я завантаження...
Завантаження файлів перервано
Завантаження завершено
Робота програми завершена
```

Рисунок 8.20. Результат роботи програми

8.6.2. Скасування виконання async-корутини

Подібним чином можна скасовувати виконання корутин, створюваних за допомогою функції **async()**. У цьому випадку зазвичай виклик методу **await()** міститься у блоці **try**, наприклад:

```
1  import kotlinx.coroutines.*
2
3  suspend fun main() = coroutineScope{
4
5      // створюємо та запускаємо корутину
6      val message = async {
7          getMessage()
8      }
9      // скасування корутини
10     message.cancelAndJoin()
```

```
11
12      try {
13          // очікуємо на отримання результату
14          println( "message: ${message.await()}" )
15      }
16      catch (e:CancellationException){
17          println( "Coroutine has been canceled" )
18      }
19      println( "Program has finished" )
20  }
21
22  suspend fun getMessage() : String{
23      delay(500L)
24      return "Hello"
25  }
```

Результат роботи програми наведено на рисунку 8.21.

```
Coroutine has been canceled
Program has finished
```

Рисунок 8.21. Результат роботи програми

8.7. Канали

Канали дозволяють передавати потоки даних. У мові Kotlin канали представлені інтерфейсом **Channel**, у якого можна відзначити два основні методи:

- **abstract suspend fun send(element: E): Unit** – Відправляє об'єкт **element** у канал;
- **abstract suspend fun receive(): E** – Отримує дані з каналу.

Розглянемо приклад визначивши найпростіший канал, через який передаватимемо числа типу **Int**:

```
1   import kotlinx.coroutines.*
2   import kotlinx.coroutines.channels.Channel
3
4   suspend fun main() = coroutineScope{
5
6       val channel = Channel<Int>()
7       launch {
8           for (n in 1..5) {
9               // надсилаємо дані через канал
10              channel.send(n)
11          }
12      }
13
```

```
14      // отримуємо дані з каналу
15      repeat(5) {
16          val number = channel.receive()
17          println(number)
18      }
19      println("End")
20  }
```

Основна функціональність каналів зосереджена в пакеті **kotlinx.coroutines.channels**, відповідно з нього і необхідно імпортувати тип **Channel**, а саме:

```
1   import kotlinx.coroutines.channels.Channel
```

У програмі визначимо змінну, яка представлятиме канал, а саме:

```
1   val channel = Channel<Int>()
```

Оскільки через канал будуть передаватися значення типу **Int**, відповідно об'єкт **Channel** типізований типом **Int**.

Потім за допомогою функції **launch** створюється корутина, в якій в циклі передаємо у канал числа від 1 до 5:

```
1   launch {
2       for (n in 1..5) {
3           // надсилаємо дані через канал
4           channel.send(n)
5       }
6   }
```

В цьому прикладі у метод **send()** передається значення, яке необхідно відправити через канал. Особливістю цього є те, що можна його запустити тільки в корутині.

Для отримання даних з каналу за допомогою функції **repeat()** визначається функція, яка буде виконуватись 5 разів, оскільки передається в канал п'ять чисел, а саме:

```
1   repeat(5) {
2       val number    =    channel.receive() //    отримуємо
    значення з каналу
3       println(number)
4   }
```

Метод **receive()** повертає об'єкт, що отримується з каналу.

Результат роботи програми наведено на рисунку 8.22.

```
1
2
3
4
5
End
```

Рисунок 8.22. Результат роботи програми

Розглянемо інший приклад, а саме відправку через канал рядків:

```
1   import kotlinx.coroutines.*
2   import kotlinx.coroutines.channels.Channel
3
4   suspend fun main() = coroutineScope{
5       val channel = Channel<String>()
6       launch {
7           val users                                          =
8   listOf( "Taras" , "Volodymyr" , "Stepan" )
            for (user in users) {
9               println( "Sending $user" )
10              channel.send(user)
11          }
12      }
13
14      repeat(3) {
15          val user = channel.receive()
16          println("Received: $user")
17      }
18      println("End")
19  }
```

В цьому прикладі в канал передається три рядки, відповідно функція **repeat()** тричі запускає отримання даних із каналу. Результат роботи програми наведено на рисунку 8.23.

```
Sending Taras
Received: Taras
Sending Volodymyr
Received: Volodymyr
Sending Stepan
Received: Stepan
End
```

Рисунок 8.23. Результат роботи програми

8.7.1. Закриття каналу

Щоб вказати, що в каналі більше немає даних, його можна закрити методом **close()**. Якщо для отримання даних з каналу застосовується цикл **for**, то, отримавши сигнал про закриття каналу, цей цикл отримає всі раніше надіслані об'єкти до закриття і завершить виконання, як у прикладі нище:

```
1   import kotlinx.coroutines.*
2   import kotlinx.coroutines.channels.Channel
3
4   suspend fun main() = coroutineScope{
5
6       val channel = Channel<String>()
7       launch {
8           val users                                    =
    listOf("Taras" , "Volodymyr" , "Stepan")
9           for (user in users) {
10              channel.send(user) // Відправляємо дані до
    каналу
11          }
12          channel.close() // Закриття каналу
13      }
14
15      for (user in channel) { // Отримуємо дані з каналу
16          println(user)
17      }
18      println("End")
19  }
```

8.7.2. Патерн producer-consumer

Розглянутий вище приклад насправді є поширеним способом передачі даних від однієї корутини до іншої. І щоб спростити написання такого коду, мова Kotlin надає низку додаткових функцій. Так, функція **produce()** є будівельником корутини, який створює корутину, в яку передаються дані каналу. Наприклад, за допомогою функції **produce()** можна визначити нову функцію-корутину, яка надсилатиме певні дані, наприклад:

```
1   fun CoroutineScope.getUsers(): ReceiveChannel<String>
    = produce{
2       val users                                        =
    listOf("Taras" , "Volodymyr" , "Stepan")
3       for (user in users) {
4           send(user)
5       }
6   }
```

В цьому прикладі визначається функція **getUsers()**. Причому вона визначається як функція інтерфейсу **CoroutineScope**. Функція повинна повертати об'єкт **ReceiveChannel**, типізований тим типом даних, які передаються. В наведеному прикладі передається значення типу **String**.

Функція **getUsers()** є корутиною, що створюється будівельником корутин **produce**. У корутині знову ж таки здійснюється прохід за списком рядків і за допомогою функції **send** дані передаються в канал.

Для отримання даних з каналу може застосовуватися метод **consumeEach()** об'єкта **ReceiveChannel**, який по суті замінює цикл **for**. Він приймає функцію, в яку як параметр передається об'єкт, що отримується з каналу, наприклад:

```
1   val users = getUsers()
2   users.consumeEach { user -> println(user) }
```

Повний код програми матиме наступний вигляд:

```
1   import kotlinx.coroutines.*
2   import kotlinx.coroutines.channels.*
3
4   suspend fun main() = coroutineScope{
5
6       val users = getUsers()
7       users.consumeEach { user -> println(user) }
8       println( "End" )
9   }
10
11  fun CoroutineScope.getUsers(): ReceiveChannel<String>
    = produce{
12      val users                                          =
    listOf( "Taras" , "Volodymyr" , "Stepan" )
13      for (user in users) {
14          send(user)
15      }
16  }
```

Результат роботи програми наведено на рисунку 8.24.

```
Taras
Volodymyr
Stepan
End
```

Рисунок 8.24. Результат роботи програми

Резюме

В наведеному розділі розглянуто корутини в мові Kotlin, їх використання для асинхронного та паралельного програмування. Описано підключення бібліотеки **kotlinx.coroutines**, створення suspend-функцій і запуск корутин за допомогою **launch** та **async**. Розглянуто роботу з потоками, область корутин (coroutine scope) і управління їхнім життєвим циклом. Пояснено роль об'єкта Job для контролю виконання корутин, включаючи їхню зупинку та відкладене виконання. Також детально розглянуто диспетчери корутин (**Dispatchers.Default**, **Dispatchers.IO** тощо) і способи зміни потоків виконання. Наприкінці розглянуто скасування корутин, обробку винятків **CancellationException** та використання каналів для передачі даних між корутинами.

Ознайомлення із цим розділом дозволить отримати знання: про корутини, які забезпечують організацію асинхронності та паралельних обчислень, що стало невід'ємною рисою багатьох мов програмування; їх призначенням, оголошенням, використанням та запуском, без чого не можливо створення мобільних, десктопних та веб-застосунків мовою Kotlin.

Контрольні запитання і завдання

1. Що таке корутини, і для чого вони використовуються в Kotlin?
2. У чому різниця між асинхронним програмуванням і багатопотоковістю?
3. Яка бібліотека використовується для роботи з корутинами в Kotlin?
4. Як підключити бібліотеку **kotlinx.coroutines** до проєкту?
5. Чому suspend-функції можуть викликатися тільки з корутин або інших suspend-функцій?
6. Яка функція використовується для створення корутин у Kotlin?
7. У чому різниця між функціями **launch** і **async**?
8. Що таке **coroutineScope** і для чого він використовується?
9. Як створити та запустити корутину у функції **main**?
10. Як можна передати параметри у корутину?
11. Що таке **Job**, і як він використовується для управління корутинами?
12. Як зупинити виконання корутини?
13. Які є способи відкладеного запуску корутин?
14. Як працює метод **join()**, і для чого він потрібен?
15. Що відбувається, коли викликається метод **cancel()** у корутині?
16. Які є основні типи диспетчерів корутин у Kotlin?
17. У чому відмінність між **Dispatchers.Default** і **Dispatchers.IO**?
18. Як змусити корутину виконуватись у певному потоці?

19. Що таке канали (**Channel**) у Kotlin, і як вони використовуються?
20. Як працює патерн **producer-consumer** у корутинах?
21. Додайте бібліотеку **kotlinx.coroutines** до проєкту та переконайся, що вона працює.
22. Напишіть програму, в якій корутина виводить 5 чисел із затримкою 300 мс між ними.
23. Створіть suspend-функцію, яка імітує завантаження даних (наприклад, повертає рядок «Дані отримані» після 2-секундної затримки).
24. Напишіть програму, в якій дві корутини виконуються одночасно, одна виводить парні числа, а інша — непарні.
25. Запустіть корутину, яка друкує числа від 1 до 10, і через 1 секунду примусово зупиніть її за допомогою **cancel()**.
26. Створіть дві async-корутини, які обчислюють суму двох чисел. Використайте **await()**, щоб отримати результат.
27. Запустіть три корутини з різними диспетчерами (**Dispatchers.Default**, **Dispatchers.IO**, **Dispatchers.Unconfined**) і виведіть, в яких потоках вони працюють.
28. Напишіть програму, яка створює канал і передає через нього 5 повідомлень між корутинами.
29. Реалізуйте патерн **producer-consumer**, де одна корутина генерує числа, а інша отримує їх і друкує на екран.
30. Створіть async-корутину, яка виконує тривалу задачу (наприклад, обчислення факторіала великого числа). Через певний час скасуйте її виконання.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Яка бібліотека використовується для роботи з корутинами в Kotlin?

a) kotlin.coroutines
b) kotlinx.coroutines
c) kotlin.concurrent
d) kotlinx.threads

Правильна відповідь: b).

Що відбудеться, якщо викликати job.cancel() для корутини?

a) Корутину буде негайно зупинений без завершення коду
b) Корутина завершить свою роботу перед скасуванням

c) Корутину буде скасовано, але вона може виконати блок finally

d) Код корутини продовжить виконуватись без змін

Правильна відповідь: c).

Яке твердження про async є правильним?

a) async використовується лише для запуску корутин без повернення результату

b) async завжди виконується у головному потоці

c) async повертає об'єкт Deferred, з якого можна отримати результат

d) async не можна використовувати з await()

Правильна відповідь: c).

Що таке suspend-функція в Kotlin?

a) Це функція, яка завжди виконується в окремому потоці

b) Це функція, яка може бути викликана лише всередині корутин

c) Це функція, яка автоматично виконується асинхронно

d) Це функція, яка не підтримує delay()

Правильна відповідь: b).

Закриті запитання з кількома правильними відповідями

Які з наведених операторів можна використовувати для створення корутин?

a) launch

b) async

c) thread

d) runBlocking

Правильна відповідь: a), b), d).

Завдання на відповідність

Співвіднесіть диспетчер корутин із його призначенням:

a) Dispatchers.Default → 1. Використовується для обчислень

b) Dispatchers.IO → 2. Використовується для I/O-операцій

c) Dispatchers.Main → 3. Використовується у графічних інтерфейсах

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

Вставити пропущені слова

Заповніть пропуски у коді, щоб корутина працювала правильно:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    _____________ {  // Створення корутини
        delay(1000L)
        println("Hello, Coroutines!")
    }
}
```

Правильна відповідь: launch

Відкриті запитання

Який буде результат виведення на консоль цього коду? Поясніть чому.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    val job = launch {
        for (i in 1..3) {
            delay(500L)
            println(i)
        }
    }
    delay(1000L)
    job.cancel()
}
```

Правильна відповідь:

1

2

Третє число не буде виведене, бо корутину буде скасовано після 1000 мс.

У якому порядку буде виведено текст на консоль у цьому коді?

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(300L)
```

```
        println("A")
    }
    println("B")
}
```

Правильна відповідь:

B

A

Оскільки launch запускається асинхронно, а println("B") виконується негайно.

Чому цей код не скомпілюється?

```
import kotlinx.coroutines.*

fun main() {
    delay(1000L)
    println("Hello, Coroutines!")
}
```

Правильна відповідь: **delay()** можна використовувати тільки всередині suspend-функції або корутини.

9. АСИНХРОННІ ПОТОКИ

9.1. Введення в асинхронні потоки

Корутини дозволяють повертати одиничні значення. І тому можна, наприклад, створювати корутину за допомогою будівельника **async**. Але мова Kotlin також дозволяє створювати асинхронні потоки (**Asynchronous Flow**), які повертають набір об'єктів.

В принципі для отримання набору об'єктів можна у корутині повертати колекцію елементів, наприклад, список **List**, на кшталт наступного:

```
1   import kotlinx.coroutines.*
2
3   suspend fun main() = coroutineScope{
4
5       launch {
6           getUsers().forEach { user -> println(user) }
7       }
8   }
9
10  suspend fun getUsers(): List<String> {
11      delay(1000L) // імітація тривалої роботи
12      return listOf( "Taras" , "Volodymyr" , "Stepan" )
13  }
```

Проте проблема таких колекцій у тому, що вони миттєво повертають усі об'єкти. Наприклад, якщо у списку очікується 1000 об'єктів, то, відповідно, поки функція **getUsers()** не поверне список із 1000 об'єктів (наприклад, отримуючи їх з бази даних або із зовнішнього Інтернет-ресурсу), буде не можлива маніпуляція об'єктами з цього списку.

Цю проблему в мові Kotlin якраз і дозволяють вирішити асинхронні потоки. Змінимо приклад вище із застосуванням асинхронних потоків, а саме:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4       getUsers().collect { user -> println(user) }
5   }
6
7   fun getUsers(): Flow<String> = flow {
8       val database                                    =
    listOf( "Taras" , "Volodymyr" , "Stepan" ) //   умовна
    база даних
9       var i = 1;
10      for (item in database) {
```

```
11          delay(400L) // імітація тривалої роботи
12          println( "Emit $i item" )
13          emit(item) // емітуємо значення
14          i++
15      }
16  }
```

Для створення асинхронного потоку даних використовується інтерфейс **Flow**. Тобто, по суті, асинхронний потік – це об'єкт **Flow**. Він типізується типом тих даних, які мають передаватись у потоці. В наведеному прикладі передаються рядки, тому **Flow** типізується типом **String**, а саме:

```
1   fun getUsers(): Flow<>
```

При цьому, при визначенні функції потоку (в наведеному прикладі функції **getUsers**) необов'язково використовувати модифікатор **suspend**.

Для створення об'єкта **Flow** використовується спеціальна функція **flow()**, а саме:

```
1   fun getUsers(): Flow<String> = flow {
2
3       // Створення асинхронного потоку у функції flow
4   }
```

У самій функції в цьому прикладі імітується отримання об'єктів з умовної бази даних, якою тут для простоти служить перелік **List**. Цикл проходиться цим списком і відправляє в потік поточний об'єкт за допомогою функції **emit()**, а сема:

```
1   emit(item) // передаємо значення потік
```

Це є ключовим моментом. Завдяки цьому зовнішній код зможе отримати передане через **emit()** в потік значення і використовувати його.

Для індикації номера об'єкта, що відправляється, в цьому прикладі додано змінну-лічильник **i**, яка збільшується при переході до іншого об'єкта списку. Виведення номера об'єкта, що надсилається, дозволяє побачити, що отримання зовнішнім кодом об'єктів зі списку відбувається в міру його передачі в потік за допомогою функції **emit()**, а не коли будуть відправлені всі об'єкти зі списку.

У зовнішньому коді функції **main** викликається функція-потік **getUsers()**. Для управління об'єктами з потоку для інтерфейсу **Flow** визначено низку функцій, однією з яких є функція **collect()**. Як параметр вона приймає функцію, яку передає переданий об'єкт з потоку. Так, в наведеному прикладі це просто функція виведення на консоль, а саме:

```
1   getUsers().collect { user -> println(user) }
```

Результат роботи програми наведено на рисунку 9.1.

```
Emit 1 item
Taras
Emit 2 item
Volodymyr
Emit 3 item
Stepan
```

Рисунок 9.1. Результат роботи програми

Таким чином, програма не чекає, коли функція **getUsers** поверне всі рядки. А отримує рядки в міру їхнього відправлення в потік через функцію **emit()**.

Розглянемо інший приклад, де створимо та використаємо асинхронний потік чисел, а саме:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4       getNumbers().collect { number -> println(number) }
5   }
6
7   fun getNumbers(): Flow<Int> = flow{
8       for (item in 1..5) {
9           emit(item * item)
10      }
11  }
```

В цьому прикладі майже все те ж саме. Функція **getNumbers()** є асинхронним потоком об'єктів **Int**. В якості об'єктів тут до потоку додаються квадрати чисел від 1 до 5. Результат роботи програми наведено на рисунку 9.2.

```
1
4
9
16
25
```

Рисунок 9.2. Результат роботи програми

9.1.1. Запуск Flow

Варто зазначити, що асинхронний потік не запускається, поки не буде застосована термінальна операція над отриманими даними, наприклад, **collect()**:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
```

```
4
5       val numberFlow = getNumbers() // потік створений,
    але не запущений
6       println( "numberFlow has created" )
7       println( "launch collect function" )
8       numberFlow.collect { number -> println(number)
    } // запуск потоку
9   }
10
11  fun getNumbers() = flow{
12      println( "numberFlow has started" )
13      for (item in 1..5) {
14          emit(item * item)
15      }
16  }
```

Результат роботи програми наведено на рисунку 9.3.

```
numberFlow has created
launch collect function
numberFlow has started
1
4
9
16
25
```

Рисунок 9.3. Результат роботи програми

9.2. Створення асинхронного потоку

Для створення асинхронного потоку можна використовувати різні методи. Розглянемо три основні способи.

9.2.1. Функція flow

Функція-будівельник потоку **flow()** дозволяє встановити логіку передачі об'єктів в потік. Вона може застосовуватися як до окремої функції, так і сама по собі. Розглянемо приклад, створення потоку на основі функції:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val numberFlow = getNumbers()
6       numberFlow.collect{n -> println(n)}
7   }
```

```
8
9   fun getNumbers() = flow{
10      for (item in 1..5) {
11          emit(item * item)
12      }
13  }
```

В цьому прикладі будівельник **flow** створює потік на основі функції **getNumbers()**, передаючи у потік квадрати значень від 1 до 5. Результат роботи програми наведено на рисунку 9.4.

```
1
4
9
16
25
```

Рисунок 9.4. Результат роботи програми

Визначати потік як окрему функцію, як у прикладі вище, необов'язково. Наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val userFlow = flow {
6           val usersList                                   =
7   listOf( "Taras" , "Volodymyr" , "Stepan" )
            for (item in usersList){
8               emit(item)
9           }
10      }
11      userFlow.collect({user -> println(user)})
12  }
```

В цьому коді визначена змінна **userFlow**, яка має тип **Flow<String>** і яка представляє потік, створюваний будівельником **flow**. У цьому випадку **flow** передає об'єкти в потік зі списку рядків. Результат роботи програми наведено на рисунку 9.5.

```
Taras
Volodymyr
Stepan
```

Рисунок 9.5. Результат роботи програми

9.2.2. Функція flowOf

Спеціальна функція-будівельник **flowOf()** створює потік із набору переданих у функцію значень, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val numberFlow : Flow<Int> = flowOf(1, 2, 3, 5,
    8)
6       numberFlow.collect{n -> println(n)}
7   }
```

В наведеному прикладі до функції будівельника асинхронного потоку **flowOf()** передається 5 значень типу **Int**, тому потік, що створюється, має тип **Flow<Int>**. Усі передані значення автоматично передаватимуться в потік. А отримати їх можна також, як і у загальному випадку, наприклад, через функцію **collect()**. Результат роботи програми наведено на рисунку 9.6.

```
1
2
3
5
8
```

Рисунок 9.6. Результат роботи програми

Розглянемо аналогічний приклад із рядками:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val userFlow =
    flowOf( "Taras" , "Stepan" , "Volodymyr" )
6       userFlow.collect({user -> println(user)})
7   }
```

9.2.3. Метод asFlow

Стандартні колекції та послідовності в мові Kotlin мають метод розширення **asFlow()**, який дозволяє перетворити колекцію або послідовність на потік. Розглянемо наступний приклад:

```
1   import kotlinx.coroutines.flow.*
2
```

```
3   suspend fun main(){
4
5       // Перетворення послідовності в потік
6       val numberFlow : Flow<Int> = (1..5).asFlow()
7       numberFlow.collect{n -> println(n)}
8
9       // Перетворення колекції List<String> на потік
10      val userFlow                                    =
    listOf( "Taras" , "Stepan" , "Volodymyr" ).asFlow()
11      userFlow.collect({user -> println(user)})
12  }
```

9.3. Операції з потоками

Для роботи з потоками у мові Kotlin для інтерфейсу **Flow** визначено низку функцій. Залежно від того, чи вони повертають конкретне значення або оброблений потік ці функції діляться на два види: термінальні функції та проміжні функції. Розглянемо основні з них.

9.3.1. Термінальні функції потоків

Термінальні функції потоків (**terminal operators**) представляють **suspend**-функції, які дозволяють безпосередньо отримувати об'єкти з потоку або повертають якесь кінцеве значення. Наведено основні з них:

- **collect()** : отримує з потоку передані значення;
- **toList()** : перетворює потік значень на колекцію **List**;
- **toSet()** : перетворює потік значень на колекцію **Set**;
- **first() / firstOrNull()** : отримує перший об'єкт із потоку;
- **last() / lastOrNull()** : отримує останній об'єкт із потоку;
- **single() / singleOrNull()** : очікує отримання одного об'єкта з потоку;
- **count()** : отримує кількість елементів у потоці;
- **reduce()** : отримує результат певної операції над елементами потоку;
- **fold()** : отримує результат певної операції над елементами потоку, на відміну функції **reduce()** приймає початкове значення.

9.3.2. Проміжні функції

Проміжні функції (**Intermediate operator**) приймають потік та повертають оброблений потік. Наведено основні з них:

- **combine()** : об'єднує два потоки в один після застосування до їх елементів функції перетворення;

- **drop()** : виключає з початку потоку певну кількість значень та повертає отриманий потік;
- **filter()** : фільтрує потік, залишаючи ті елементи, які відповідають умові;
- **filterNot()** : фільтрує потік, залишаючи ті елементи, які не відповідають умові;
- **filterNotNull()** : фільтрує потік, видаляючи всі елементи, які дорівнюють **null**;
- **map()** : застосовує до елементів потоку функцію перетворення;
- **onEach()** : застосовує до елементів потоку певну функцію перед тим, як вони будуть передані в потік, що повертається;
- **take()** : вибирає з потоку певну кількість елементів;
- **transform()** : застосовує до елементів потоку функцію перетворення;
- **zip()** : із двох потоків створює один, застосовуючи до їх елементів функцію перетворення.

9.4. Функції count, take та drop. Кількість елементів у потоці

9.4.1. Функція count

Оператор **count** отримує кількість об'єктів у потоці, наприклад:

```
1  import kotlinx.coroutines.flow.*
2
3  suspend fun main(){
4
5      val userFlow                                          =
   listOf( "Taras" , "Volodymyr" , "Stepan" ).asFlow()
6      println( "Count:  ${userFlow.count()}" ) //  Count:
   3
7  }
```

Також можна у функцію **count()** передати умову у вигляді функції, яка повертає об'єкт **Boolean**, тобто **true** або **false**. Тоді функція **count()** поверне кількість елементів, які відповідають цій умові, наприклад:

```
1  import kotlinx.coroutines.flow.*
2
3  suspend fun main(){
4
5      val userFlow                                              =
   listOf( "Taras" , "Volodymyr" , "Kateryna" , "Stepan"
   , "Olena" ).asFlow()
6      val count    =    userFlow.count{    username    ->
   username.length > 5 }
7      println( "Count: $count" ) // Count: 2
```

```
8   }
```

У наведеному прикладі в якості умови передається функція **username -> username.length > 5**. Її параметр – це об'єкт потоку. Тут вимагається враховувати рядок, тільки якщо його довжина більше 5 символів. У результаті в наведеному списку тільки два об'єкти відповідають цій умові, тому змінна **count** дорівнюватиме 2.

9.4.2. Функція take

Оператор функції **take** обмежує кількість елементів у потоці. Як параметр функція **take** приймає кількість елементів з початку потоку, які треба залишити в потоці, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val userFlow                                          =
6   listOf( "Taras" , "Volodymyr" , "Kateryna" , "Stepan" ,
    "Olena").asFlow()
7       userFlow.take(3).collect{user -> println(user)}
    }
```

У наведеному вище прикладі залишається перші три елементи, а результат роботи програми наведено на рисунку 9.7.

```
Taras
Volodymyr
Kateryna
```

Рисунок 9.7. Результат роботи програми

9.4.3. Функція drop

Оператор функції **drop** видаляє з потоку певну кількість елементів. Як параметр функція **drop** приймає кількість елементів з початку потоку, які треба прибрати з потоку, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val userFlow                                          =
    listOf( "Taras" , "Volodymyr" , "Kateryna" , "Stepan" ,
6   "Olena").asFlow()
        userFlow.drop(3).collect{user -> println(user)}
```

```
7   }
```

У прикладі наведеному вище прибирається перші три елементи, у результаті в потоці залишиться два останні елементи, а результат роботи програми наведено на рисунку 9.8.

```
Stepan
Olena
```

Рисунок 9.8. Результат роботи програми

9.5. Функції first, last, single

9.5.1. Функції first/firstOrNull

Метод **first()** отримує перший об'єкт списку, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val userFlow                                        =
    listOf( "Taras" , "Volodymyr" , "Kateryna" , "Stepan" ,
    "Olena").asFlow()
6       val firstUser = userFlow.first()
7       println( "First User: $firstUser" ) // First User:
    Taras
8   }
```

Також метод **first()** може як параметр приймати функцію-умову, яка повертає об'єкт **Boolean**. Тоді **first()** повертає перший елемент потоку, який відповідає цій умові, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val userFlow                                        =
    listOf( "Taras" , "Volodymyr" , "Kateryna" , "Stepan"
    , "Olena" ).asFlow()
6       val firstUser = userFlow.first{ name-> name.length
    > 8}
7       println( "First User: $firstUser" ) // First User:
    Volodymyr
8   }
```

В цьому прикладі отримується перший елемент, довжина якого більше ніж 8 символів. У потоці це рядок "**Volodymyr**".

Однак може скластися ситуація, коли умові не відповідає жоден із елементів потоку, наприклад:

```
1 val userFlow                                          =
  listOf( "Taras" , "Volodymyr" , "Stepan" ).asFlow()
2 val firstUser = userFlow.first{ name-> name.length
  > 9}
```

Або потік виявився порожній, як у наступному прикладі:

```
1   val userFlow = listOf<String>().asFlow()
2   val firstUser = userFlow.first()
```

В обох випадках під час роботи програми згенерується виняток. У цьому випадку, звичайно, можна обробляти виняток за допомогою **try..catch**. Однак, в якості альтернативи мова Kotlin надає метод **firstOrNull()**, який повертає **null** якщо потік порожній або жоден з його елементів не відповідають умові, наприклад:

```
1   val userFlow = listOf<String>().asFlow()
2   val firstUser1 = userFlow.firstOrNull()
3   val firstUser2   =   userFlow.firstOrNull{   name->
    name.length > 9}
```

В цьому разі можна перевірити отримане значення на **null**, а саме:

```
1   if (firstUser1 == null )
2       println( "User not found" )
3   else
4       println( "First User: $firstUser1" )
```

9.5.2. Функції last / lastOrNull

Функція **last()** вибирає з потоку останній елемент, тільки на відміну від функції **first()**, яка принаймні для версії мови Kotlin 1.5.1 не приймає умову, може також приймати умову, наприклад

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val userFlow = listOf("Taras",
    "Volodymyr" , "Olena" , "Stepan" ).asFlow()
6       val lastUser = userFlow.last()
7       println( "Last  User:  $lastUser" ) //  Last  User:
    Stepan
8   }
```

Якщо є ймовірність, що потік буде порожнім, можна використовувати функцію **lastOrNull()** , яка повертає **null** у разі відсутності елементів, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
```

```
3   suspend fun main(){
4
5       val userFlow = listOf("Taras",
    "Volodymyr" , "Olena" , "Stepan" ).asFlow()
6       val lastUser = userFlow.lastOrNull()
7       if (lastUser!= null )      println( "Last      User:
    $lastUser" )
8   }
```

9.5.3. Функції single / singleOrNull

Функція **single()** повертає єдиний елемент потоку, якщо потік містить **лише один** елемент. Якщо потік не містить елементів, генерується виняток **NoSuchElementException**, а якщо в потоці більше одного елемента, то генерується виняток **IllegalStateException**, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val userFlow = listOf( "Taras" ).asFlow()
6       try {
7           val singleUser = userFlow.single()
8           println( "Single User: $singleUser" )
9       }
10      catch (e:Exception) { println(e.message) }
11  }
```

В якості альтернативи можна використовувати метод **singleOrNull()**, який повертає **null** якщо потік порожній або якщо в потоці більше одного елемента, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val userFlow =
    listOf( "Taras" , "Volodymyr" ).asFlow()
6       val singleUser = userFlow.singleOrNull()
7       if (singleUser!= null )
8           println( "Single User: $singleUser" )
9       else
10          println( "Not found" )
11  }
```

9.6. Перетворення даних. Функції map та transform

9.6.1. Функція map

Функція **map()** перетворює дані потоку. Як параметр функція **map()** приймає функцію перетворення. Функція перетворення приймає як єдиний параметр об'єкт із потоку і повертає перетворені дані, наприклад:

```
inline fun <T, R> Flow<T>.map(
    crossinline transform: suspend (value: T) -> R
): Flow<R> (source)
```

Розглянемо наступний приклад:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val peopleFlow = listOf(
        Person( "Taras" , 37),
        Person( "Stepan" , 41),
        Person( "Volodymyr" , 21)
    ).asFlow()

    peopleFlow.map{ person -> person.name }
        .collect { personName -> println(personName) }
}

data class Person( val name: String, val age: Int)
```

В цьому прикладі визначається потік даних **peopleFlow**, який містить об'єкти типу **Person**. Далі до цього потоку застосовується функція **map()**, у яку передається наступна функція перетворення:

```
person -> person.name
```

Тобто функція перетворення приймає як параметр **person** об'єкт типу **Person** і повертає значення його поля **name**, тобто рядок. Таким чином, функція **map()** із потоку об'єктів **Person** створює потік об'єктів **String**.

Далі до отриманого потоку об'єктів **String** застосовується функція **collect()**, в якій отримується кожен рядок з потоку та виводиться на консоль, а саме:

```
collect { personName -> println(personName) }
```

Результат роботи програми наведено на рисунку 9.9.

```
Taras
Stepan
Volodymyr
```

Рисунок 9.9. Результат роботи програми

Розглянемо інший приклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val peopleFlow = listOf(
6           Person( "Taras" , 37),
7           Person( "Vadym" , 5),
8           Person( "Stepan" , 14),
9           Person( "Volodymyr" , 21),
10      ).asFlow()
11
12      peopleFlow.map{ person -> object {
13          val name=person.name
14          val isAdult = person.age > 17
15      }}.collect    {    user    ->    println( "name:
    ${user.name} adult: ${user.isAdult} " )}
16  }
17
18  data class Person( val name: String, val age: Int)
```

У цьому прикладі функція **map()** перетворює потік даних типу **Person** на потік об'єктів анонімного типу, який визначає два поля: **name** (ім'я) та **isAdult** (чи є користувачеві більше за 17 років). Відповідно у функції **collect** отримуються об'єкти цього анонімного типу та виводяться їх дані на консоль. Результат роботи програми наведено на рисунку 9.10.

```
name: Taras adult: true
name: Vadym adult: false
name: Stepan adult: false
name: Volodymyr adult: true
```

Рисунок 9.10. Результат роботи програми

9.6.2. Функція transform

Функція **transform()** також дозволяє виконувати перетворення об'єктів у потоці. На відміну від **map()**, вона дозволяє використовувати функцію **emit()**, щоб передавати в потік довільні об'єкти, наприклад:

```
1   inline fun <T, R> Flow<T>.transform(
2       crossinline        transform:        suspend
    FlowCollector<R>.(value: T) -> Unit
3   ): Flow<R> (source)
```

Функція **transform** приймає функцію, яка отримує як параметр об'єкт потоку, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val peopleFlow = listOf(
6           Person( "Taras" , 37),
7           Person( "Vadym" , 5),
8           Person( "Stepan" , 14),
9           Person( "Volodymyr" , 21),
10      ).asFlow()
11
12      peopleFlow.transform{ person ->
13          if (person.age > 17) {
14              emit(person.name)
15          }
16      }.collect { personName -> println(personName)}
17  }
18
19  data class Person( val name: String, v a l age: Int)
```

В наведеному прикладі у функції **transform** отримується об'єкт **Person** з потоку. Якщо поле **age** цього об'єкта більше 17 то він передається в новий потік через функцію **emit()**. Результат роботи програми наведено на рисунку 9.11.

```
Taras
Volodymyr
```

Рисунок 9.11. Результат роботи програми

Причому при обробці одного об'єкта можна кілька разів викликати функцію **emit()**, передаючи таким чином кілька об'єктів в потоці, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val numbersFlow = listOf(2, 3, 4).asFlow()
6       numbersFlow.transform{ n ->
7           emit(n)
8           emit(n * n)
9       }.collect { n -> println(n)}
10  }
```

В цьому прикладі перетворюється потік чисел. Причому у вихідний потік передається саме число та його квадрат. Результат роботи програми наведено на рисунку 9.12.

Рисунок 9.12. Результат роботи програми

9.7. Фільтрування даних

9.7.1. Функція filter

Функція **filter** виконує фільтрацію об'єктів у потоці. Як параметр вона приймає функцію-умову, яка отримує об'єкт потоку і повертає **true**, якщо об'єкт відповідає критерію фільтрацію, та **false**, якщо не відповідає, наприклад:

```
inline fun <T> Flow<T>.filter(
    crossinline predicate: suspend (T) -> Boolean
): Flow<T> (source)
```

Розглянемо наступний приклад:

```
import kotlinx.coroutines.flow.*

suspend fun main(){
    val peopleFlow = listOf(
        Person( "Taras" , 37),
        Person( "Vadym" , 5),
        Person( "Stepan" , 14),
        Person( "Volodymyr" , 21),
    ).asFlow()

    peopleFlow.filter{ person -> person.age > 17}
        .collect { person -> println( "name:
${person.name} age: ${person.age} " )}
}

data class Person( val name: String, val age: Int)
```

В цьому прикладі для фільтрації використовується умова **person.age > 17**. Якщо вік користувача більше 17, він з’явитеся у вихідному потоці. Результат роботи програми наведено на рисунку 9.13.

```
name: Taras age: 37
name: Volodymyr age: 21
```

Рисунок 9.13. Результат роботи програми

9.7.2. Функція takeWhile

Мова Kotlin надає ще низку операцій фільтрації для різних ситуацій. Наприклад, оператор **takeWhile** вибирає з потоку елементи доти, доки буде виконуватись певна умова, а саме:

```
fun <T> Flow<T>.takeWhile(
    predicate: suspend (T) -> Boolean
): Flow<T>
```

Розглянемо наступний приклад:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val peopleFlow = listOf(
        Person( "Taras" , 37),
        Person( "Olena" , 32),
        Person( "Vadym" , 5),
        Person( "Stepan" , 14),
        Person( "Volodymyr" , 25),
    ).asFlow()

    peopleFlow.takeWhile{ person -> person.age > 17}
        .collect { person -> println( "name: ${person.name} age: ${person.age} " )}
}

data class Person( val name: String, val age: Int)
```

В наведеному прикладі оператор **takeWhile** вибиратиме в потік, що повертається, всі об'єкти **Person**, у яких значення **age** більше 17. Результат роботи програми наведено на рисунку 9.14.

```
name: Taras age: 37
name: Olena age: 32
```

Рисунок 9.14. Результат роботи програми

З результату роботи програми видно, що незважаючи на те, що в потоці ще є об'єкти **Person**, які відповідають умові, але зіткнувшись з першим об'єктом, який

не відповідає умові, **takeWhile** перестає вибирати об'єкти в потік, що повертається.

9.7.3. Функція dropWhile

Функція **dropWhile** є протилежною за своєю дією до функції **takeWhile**. Функція **dropWhile** видаляє з потоку елементи, доки вони не почнуть відповідати певній умові, наприклад:

```
fun <T> Flow<T>.dropWhile(
    predicate: suspend (T) -> Boolean
): Flow<T>
```

Розглянемо попередній приклад, тільки замість **takeWhile** використаємо функцію **dropWhile**, наприклад:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val peopleFlow = listOf(
        Person( "Taras" , 37),
        Person( "Olena" , 32),
        Person( "Vadym" , 5),
        Person( "Stepan" , 14),
        Person( "Volodymyr" , 25),
    ).asFlow()

    peopleFlow.dropWhile{ person -> person.age > 17}
        .collect { person -> println( "name: ${person.name} age: ${person.age} " )}
}

data class Person( val name: String, val age: Int)
```

У наведеному прикладі оператор **dropWhile** пропустить перші два об'єкти **Person**, які відповідають умові **person.age > 17**. Третій елемент потоку не відповідає цій умові, тому починаючи з третього елемента всі інші елементи потраплять у потік, що повертається, навіть якщо вони знову ж таки відповідають цій умові. Результат роботи програми наведено на рисунку 9.15.

```
name: Vadym age: 5
name: Stepan age: 14
name: Volodymyr age: 25
```

Рисунок 9.15. Результат роботи програми

9.8. Зведення даних. Функції reduce та fold

9.8.1. Функція reduce

Функція **reduce** зводить всі значення потоку до одного значення, наприклад:

```
suspend fun <S, T : S> Flow<T>.reduce(
    operation: suspend (accumulator: S, value: T) -> S
): S (source)
```

Функція **reduce** приймає функцію, яка має два параметри. Перший параметр при першому запуску представляє перший об'єкт потоку, а при наступних запусках – результат функції над попередніми об'єктами. А другий параметр функції є наступним об'єктом.

Розглянемо приклад, де є потік чисел і необхідно знайти суму всіх чисел у потоці:

```
import kotlinx.coroutines.flow.*

fun main(){

    val numberFlow = listOf(1, 2, 3, 4, 5)
    val reducedValue = numberFlow.reduce{ a, b -> a + b }
    println (reducedValue) // 15
}
```

В цьому прикладі при першому запуску у функції **reduce** параметр **a** дорівнює 1, а параметр **b** дорівнює 2.

При другому запуску параметр **a** містить результат попереднього виконання функції, якщо число 1 + 2 = 3, а параметр **b** дорівнює 3, яке є наступним числом в потоці.

Розглянемо інший приклад з рядками:

```
import kotlinx.coroutines.flow.*

fun main(){

    val userFlow = listOf( "Taras" , "Volodymyr" , "Kateryna" , "Stepan" , "Olena" ).asFlow()
    val reducedValue = userFlow.reduce{ a, b -> a + " " + b }
    println(reducedValue) // Taras Volodymyr Kateryna Stepan Olena
}
```

В цьому прикладі **reduce** об'єднує всі рядки в один.

9.8.2. Функція fold

Функція **fold** також зводить всі елементи потоку в один. Але на відміну від оператора **reduce** функція **fold** як перший параметр набуває певного початкового значення, наприклад:

```
inline suspend fun <T, R> Flow<T>.fold(
    initial: R,
    crossinline operation: suspend (acc: R, value: T) -> R
): R (source)
```

Розглянемо наступний приклад:

```
import kotlinx.coroutines.flow.*

suspend f u n main(){

    val userFlow = listOf( "Taras" , "Volodymyr" , "Kateryna" , "Stepan" , "Olena" ).asFlow()
    val foldedValue = userFlow.fold("Users:", { a, b -> a + " " + b })
    println(foldedValue) // Users: Taras Volodymyr Kateryna Stepan Olena
}
```

У цьому прикладі початковим значенням є рядок "**Users:**", до якого потім додаються інші елементи потоку даних.

9.9. Об'єднання потоків

Функція **zip** дозволяє об'єднати два потоки даних, а саме:

```
fun <T1, T2, R> Flow<T1>.zip(
    інші: Flow<T2>,
    transform: suspend (T1, T2) -> R
): Flow<R> (source)
```

Функція **zip** приймає два параметри. Перший параметр – це потік даних, з яким потрібно об’єднатися. Другим параметром є фактична функція злиття. Він приймає в якості параметрів відповідні елементи обох потоків і повертає результат їх об’єднання.

Розглянемо приклад об’єднання двох потоків:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val english =
```

```
    listOf( "red" , "yellow" , "blue" ).asFlow()
6       val ukrainian                                   =
    listOf( "червоний" , "жовтий" , "синій" ).asFlow()
7       english.zip(ukr) { a, b -> "$a: $b" }
8           .collect { word -> println(word) }
9   }
```

В цьому прикладі оператор **zip** послідовно перебирає елементи з обох потоків **english** та **ukrainian**. Перший елемент потоку, який викликається оператором **zip**, передається параметру **a**, а другий елемент потоку передається параметру **b**. Функція об'єднання з'єднує обидва елементи в один рядок. І кожний такий рядок з двох елементів передається в якості елемента для новоствореного потоку. Результат роботи програми наведено на рисунку 9.16.

```
red: червоний
yellow: жовтий
blue: синій
```

Рисунок 9.16. Результат роботи програми

Хоча в прикладі вище функція **zip** об'єднувала потоки одного типу, а саме **String**, і повертала потік того ж типу, але насправді потоки, що об'єднуються, можуть представляти різні дані, а потік, що повертається - ще один тип даних, наприклад:

```
1   import kotlinx.coroutines.flow.*
2
3   suspend fun main(){
4
5       val names                                       =
    listOf( "Taras" , "Volodymyr" , "Stepan" ).asFlow()
6       val ages = listOf(37, 41, 25).asFlow()
7       names.zip (вік) { name, age -> Person(name, age) }
8           .collect    {    person    ->    println( "Name:
    ${person.name} Age: ${person.age}" ) }
9   }
10
11  data class Person( val name: String, val age: Int)
```

У цьому прикладі функція **zip**, яка викликається на потоці типу **String**, поєднує його елементи з елементами потоку типу **Int**. Причому результатом такого поєднання стає потік об'єктів типу **Person**. Результат роботи програми наведено на рисунку 9.17.

```
Name: Taras Age: 37
Name: Volodymyr Age: 41
Name: Stepan Age: 25
```

Рисунок 9.17. Результат роботи програми

Резюме

В наведеному розділі розглядається концепція асинхронних потоків (**Flow**) у мові Kotlin, які дозволяють передавати дані поступово, а не всі одразу, як у списках. Показано, як створювати потоки за допомогою функцій **flow()**, **flowOf()** та **asFlow()**, а також як використовувати їх у корутинах. Розглянуто термінальні (**collect()**, **toList()**, **count()**) і проміжні (**map()**, **filter()**, **take()**, **drop()**) оператори для роботи з потоками. Наведено способи запуску потоку, особливості обробки його елементів і функції перетворення (**map**, **transform**). Окремо розглянуто функції зведення даних (**reduce**, **fold**) і методи об'єднання потоків (**zip**). Загалом, матеріал дає розуміння, як ефективно використовувати асинхронні потоки в Kotlin.

Ознайомлення із цим розділом дозволить отримати знання: про асинхронні потоки: їх створення та операції над ними, які забезпечують організацію асинхронності та паралельних обчислень, що дозволяє завершити курс вивчення мови Kotlin на цьому етапі і перейти до вдосконалення вмінь і навичок зі створення мобільних, десктопних та веб-застосунків мовою Kotlin.

Контрольні запитання і завдання

1. Що таке асинхронний потік (**Flow**) у Kotlin?
2. Чим асинхронний потік відрізняється від звичайного списку (**List**)?
3. Як створити потік за допомогою функції **flow()**?
4. Чи потрібно використовувати **suspend** для функцій, що повертають **Flow**?
5. Як отримати дані з потоку?
6. Яка функція використовується для збору (отримання) значень з потоку?
7. Що відбудеться, якщо викликати функцію **collect()** кілька разів?
8. Які основні термінальні оператори потоку ви знаєте?
9. Чим **first()** відрізняється від **firstOrNull()**?
10. Яку функцію слід використовувати, щоб отримати останній елемент потоку?
11. Як перетворити **List** у **Flow**?
12. Чим **flowOf()** відрізняється від **flow()**?
13. Як можна фільтрувати дані у потоці?

14. Чим функція **map()** відрізняється від **transform()**?
15. Що робить **функція take(n)**?
16. Як працює функція **drop(n)**?
17. Чим **reduce()** відрізняється від **fold()**?
18. Як поєднати два потоки в один?
19. Що відбувається, якщо **Flow** не запустити через термінальну операцію?
20. Яке значення поверне **count()**, якщо потік порожній?
21. Напишіть функцію, яка створює потік чисел від 1 до 5 і виводить їх у консоль.
22. Додайте в попереднє завдання затримку **delay(500L)** перед передачею кожного числа в потік.
23. Напишіть потік, який передає імена користувачів, а потім застосуйте **map()**, щоб перетворити всі імена на великі літери.
24. Створіть потік чисел від 1 до 10 і залиште в ньому лише парні числа, використовуючи **filter()**.
25. Напишіть функцію, яка створює потік випадкових чисел і обмежує його перші 3 значення за допомогою **take(3)**.
26. Створіть два потоки: один з іменами, інший з віком. Використовуючи **zip()**, об'єднайте їх у потік об'єктів **Person(name, age)**.
27. Створіть потік чисел від 1 до 10 і знайдіть їхню суму, використовуючи **reduce()**.
28. Напишіть потік об’єктів **Person(name, age)**, використовуючи **transform()**, передайте у вихідний потік лише тих, кому більше 18 років.
29. Створіть потік, але перед його запуском виведіть повідомлення «Створення потоку...», а потім виконайте **collect()**, щоб побачити порядок виконання коду.
30. Є список **listOf("Apple", "Banana", "Cherry")**. Перетворіть його в **Flow** і виведіть кожен елемент у консоль.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Яка основна перевага використання Flow у порівнянні з List?

a) Flow дозволяє отримувати дані поступово, а не всі одразу. ✓

b) Flow працює швидше, ніж List.

c) Flow не потребує корутин.

d Flow автоматично кешує всі отримані значення.

Правильна відповідь: a).

Виберіть правильний оператор для отримання 3 перших значень з потоку:

a) filter(3)

b) drop(3)

c) take(3)

d) map(3)

Правильна відповідь: c).

Який з операторів використовується для перетворення колекції в потік?

a) flowOf()

b) zip()

c) collect()

d) asFlow()

Правильна відповідь: d).

Закриті запитання з кількома правильними відповідями

Які методи є термінальними операторами для Flow? (Виберіть всі правильні варіанти)

a) collect()

b) toList()

c) map()

d) count()

Правильна відповідь: a), b), d).

Завдання на відповідність

Співставте функції Flow з їх описом:

a) map() → 1. Перетворює дані у потоці

b) filter() → 2. Вибирає лише елементи, що відповідають умові

c) zip() → 3. Об'єднує два потоки в один

d) collect() → 4. Отримує значення з потоку

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

d) → 4.

Вставити пропущені слова

Доповніть код так, щоб він створював потік і збирав дані:

```
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

fun getNumbers(): Flow<Int> = flow {
    for (i in 1..3) {
        emit(i)
    }
}

fun main() = runBlocking {
    getNumbers()._____________ { number -> println(number) }
}
```

Правильна відповідь: collect

Завдання з вибором істинності

Метод firstOrNull() поверне виняток, якщо потік порожній.

a) Так

b) Ні

Правильна відповідь: b). Метод firstOrNull() поверне null, а first() — виняток.

Відкриті запитання

Який порядок виведення на екран буде у наступному коді?

```
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.runBlocking

fun getNumbers(): Flow<Int> = flow {
    println("Generating numbers")
    emit(1)
    emit(2)
    emit(3)
}

fun main() = runBlocking {
```

```
        println("Start")
        getNumbers().collect { println(it) }
        println("End")
    }
```

Правильний порядок виводу:

Start

Generating numbers

1

2

3

End

Яким буде результат виконання наступного коду?

```
    import kotlinx.coroutines.flow.*
    import kotlinx.coroutines.runBlocking

    fun main() = runBlocking {
        val numbers = flowOf(1, 2, 3, 4, 5)
        val sum = numbers.reduce { acc, value -> acc +
value }
        println(sum)
    }
```

Правильна відповідь: 15.

Що поверне наступний код?

```
    import kotlinx.coroutines.flow.*
    import kotlinx.coroutines.runBlocking

    fun main() = runBlocking {
        val users = listOf("Anna", "Oleh", "Maria",
"Ihor").asFlow()
        val firstUser = users.firstOrNull { it.length > 6
}
        println(firstUser)
    }
```

Правильна відповідь: null, оскільки жодне ім'я не довше 6 символів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Dawn Griffiths, David Griffiths. Head First Kotlin. «O'Reilly Media», 2019. – 482 p. ISBN 978-1491996690.
2. Dmitry Jemerov, Svetlana Isakova. Kotlin in Action. «Manning», 2017. – 360 p. ISBN 978-1617293290.
3. John Horton, Android Programming with Kotlin for Beginners: Build Android apps starting from zero programming experience with the new Kotlin programming language. «Packt Publishing», 2019. – 698 p. ISBN 978-1789615401.
4. Ken Kousen, Kotlin Cookbook: A Problem-Focused Approach. «O'Reilly Media», 2019. – 251 p. ISBN 978-1492046677.
5. Duncan McGregor, Nat Pryce, Java to Kotlin: A Refactoring Guidebook, «O'Reilly Media», 2021. – 422 p. ISBN 978-1492082279.
6. Nate Ebel, Mastering Kotlin: Learn advanced Kotlin programming techniques to build apps for Android, iOS, and the web, «Packt Publishing», 2019. - 434 p. ISBN 978-1838555726.
7. Kotlin Programming Language [Електронний ресурс]. - Режим доступу: https://kotlinlang.org/. - Назва з екрана.
8. GitHub - JetBrains_kotlin The Kotlin Programming Language [Електронний ресурс]. - Режим доступу: https://github.com/JetBrains/kotlin/. - Назва з екрана.
9. Java Downloads Oracle [Електронний ресурс]. - Режим доступу: http://www.oracle.com/technetwork/java/javase/downloads/index.html. - Назва з екрана.
10. Release Kotlin 1.9.23 JetBrains_kotlin GitHub [Електронний ресурс]. - Режим доступу: https://github.com/JetBrains/kotlin/releases/latest/. - Назва з екрана.
11. Download IntelliJ IDEA – The Leading Java and Kotlin IDE [Електронний ресурс]. - Режим доступу: https://www.jetbrains.com/idea/download/. - Назва з екрана.
12. Kotlin Docs Kotlin Documentation [Електронний ресурс]. - Режим доступу: https://kotlinlang.org/docs/home.html. - Назва з екрана.
13. Kotlin.collections - Kotlin Programming Language [Електронний ресурс]. - Режим доступу: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html. - Назва з екрана.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ І ЕЛЕКТРОННИХ РЕСУРСІВ

1. Android Mobile App Developer Tools – Android Developers/ [Електронний ресурс]. - Режим доступу: https://developer.android.com/. - Назва з екрана.
2. Declaring Member Variables. The Java™ Tutorials Learning the Java Language Classes and Objects [Електронний ресурс]. - Режим доступу: https://docs.oracle.com/javase/tutorial/java/javaOO/variables.html. - Назва з екрана.
3. Creating Objects. The Java™ Tutorials Learning the Java Language Classes and Objects [Електронний ресурс]. - Режим доступу: https://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html. - Назва з екрана.
4. What's New in JDK 8 [Електронний ресурс]. - Режим доступу: http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html. - Назва з екрана.
5. Шульженко, К. О. Інструментальні засоби взаємодії в мережах транспортних засобів : магістерська дис. : 121 Інженерія програмного забезпечення / Шульженко Кирило Олександрович. – Київ, 2024. - 99 с. [Електронний ресурс]. - Режим доступу: https://ela.kpi.ua/handle/123456789/64408. - Назва з екрана.
6. Токар М. Є. Інструментарій для реалізації процедур машинного навчання на мобільних пристроях : дипломна робота бакалавра : 122 Комп'ютерні науки / Токар Михайло Євгенович. - Київ, 2022. - 102 с. [Електронний ресурс]. - Режим доступу: https://ela.kpi.ua/handle/123456789/53857. - Назва з екрана.
7. Гурин, Я. Ю. Мобільний додаток кабінету аспіранта кафедри : дипломна робота бакалавра : 121 Інженерія програмного забезпечення / Гурин Ярослав Юрійович. - Київ, 2023. - 65 с. [Електронний ресурс]. - Режим доступу: https://ela.kpi.ua/handle/123456789/59775. - Назва з екрана.
8. Antonio Leiva. Kotlin for Android Developers: Learn Kotlin the easy way while developing an Android App. CreateSpace Independent Publishing Platform, 2016. 240 p.
9. Ashok Kumar. S. Mastering Firebase for Android Development. Packt Publishing, 2018. 394 p. ISBN 978-1788624718.
10. Neil Smyth. Android Studio 3.0 Development Essentials. CreateSpace Independent Publishing Platform, 2017. 726 p. ISBN 978-1977540096/

11. Pierre-Yves. Saumont. The Joy of Kotlin. Manning, 2019. 480 p. ISBN 978-1617295362/
12. Reto Meier. Professional Android. Wrox, 2018. 928 p. ISBN 978-1118949528
13. Thomas Kunneth. Android UI Development with Jetpack Compose. Packt Publishing, 2022. 248 p. ISBN 978-1837634255/
14. Kotlin documentation: веб-сайт. URL: [Електронний ресурс]. - Режим доступу: https://kotlinlang.org/docs/home.html. - Назва з екрана.
15. Android documentation: веб-сайт. URL: [Електронний ресурс]. - Режим доступу: https://developer.android.com/guide. - Назва з екрана.
16. Firebase documentation: веб-сайт. URL: [Електронний ресурс]. - Режим доступу: https://firebase.google.com/docs. - Назва з екрана.