# Зміст



**Система оцінювання**

| № | Тема | К-ть балів |
|---|---|---|
| 1. | *Захист принаймні одного завдання з роботи* | 1 |
| 2. | Завдання в тексті | 0,6* |
| 3. | Практичні завдання | 3* |
| 4. | *Здача звіту* | 0,4 |
| | **Всього** | **5** |

* – діє бонусна система

# Тема 1. Базовий синтаксис мови програмування Kotlin

Мова програмування Kotlin у значній мірі є безпечнішою, зручнішою та компактнішою версією мови програмування Java. Середовище розробки IntelliJ IDEA, яке буде використовуватись для демонстрації виконання програмного коду, дозволяє перетворювати Java-код при його копіюванні у файли первинного коду Kotlin (зазвичай з розширенням *.kt). Зокрема, версія коду з лістингу 1.2 була отримана шляхом автоматичного конвертування Java-коду з лістингу 1.1. Створення та збирання виконується аналогічно до Java-проєктів. Крім того, допускається збирання в межах одного проєкту як первинного коду мовою Java, так і Kotlin. У цілому, отриманий код повністю повторює функціональність Java-версії з лістингу 1.1, проте того ж результату можна добитись, залишивши *лише рядки 3-5* з лістингу 1.2.

***Лістинг 1.1. Найпростіша програма мовою Java***

```
1  public class Main {
2      public static void main(String[] args) {
3          System.out.println("Hello World!");
4      }
5  }
```

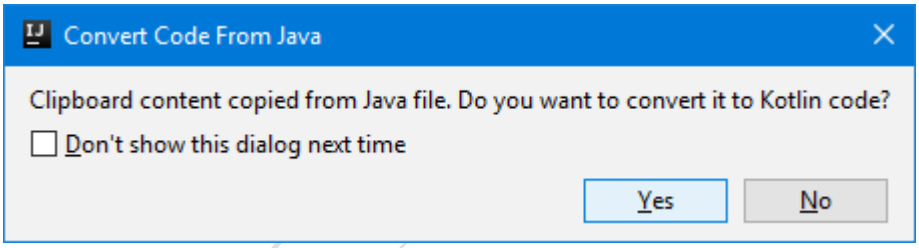


**Рис. 1.1. Діалогове вікно автоматичної конвертації Java-коду в Kotlin**

***Лістинг 1.2. Сконвертована версія коду мовою Kotlin***

```
1  object Main {
2      @JvmStatic
3      fun main(args: Array<String>) {
4          println("Hello World!")
5      }
6  }
```

Опишемо загальні відмінності, виявлені на даному етапі:

– необов'язковість класу в Kotlin-коді, на відміну від мови Java. У сконвертованій версії присутнє ключове слово object, яке в даному контексті є аналогом синглетного класу (допускає конструювання тільки одного об'єкта). Таким чином, замість опису класу та створення його екземпляру з точкою ходу в додаток пропонується компактно створити один об'єкт, який і буде мати цю точку входу. Як уже було зазначено, рядки 1, 2 та 6 взагалі можна опустити, оскільки тут не вимагається створення екземпляру класу;

– у мові Kotlin відсутнє ключове слово static, проте аналогічна поведінка досягається анотацією @JvmStatic. Це ще раз нагадує, що базовою віртуальною машиною для виконання байткоду залишається Java Virtual Machine. Тому можливо звернутись до функціональності, яка доступна в мові Java, проте в явному вигляді не представлена в мові Kotlin. Слід зауважити, що власне

глобальність в коді мовою Kotlin просто реалізується іншими мовними засобами залежно від контексту. У даному випадку для дослівної конвертації коду використали вище згадану анотацію;

– значно відрізняються сигнатури функцій та методів. По-перше, використовується зарезервоване слово fun, яке позначає функцію або метод. По-друге, тип функції визначається не перед її назвою, а в кінці. У лістингу 1.2 тип функції не зазначений, оскільки в Java-версії коду ним був void, проте за потреби повернення цілого числа сигнатура мала б наступний вигляд:

```
fun main(args: Array<String>):Int
```

По-третє, оголошення параметрів функції, як і змінних в цілому, передбачає дещо інший запис – назваЗмінної: типЗмінної. Саме тому тип функції записується після списку параметрів. Крім того, відсутній модифікатор доступу public, що пов'язано з його використанням у якості модифікатора доступу за умовчанням;

– для позначення масивів використовується спеціальний клас Array, на відміну від Java, в якій поширеним є запис із квадратними дужками;

– розробники Kotlin зробили обгортки над популярними методами Java, зокрема System.out.println(), що й відображено в лістингу 1.2. Також зауважте, що можна не ставити оператор ";" наприкінці інструкцій. Проте навіть поставивши його, компілятор просто попередить, що в тому місці код можна спростити.

| | |
|---|---|
| **Завдання** | *Зберіть власний програмний проєкт у середовищі IntelliJ IDEA та виведіть на екран свою коротку автобіографію.* |

### Змінні та типи даних в мові Kotlin

Згадуючи про синтаксис оголошення та ініціалізації змінних, слід відмітити появу ключових слів val і var (ключове слово var з'явилось у Java 10). Вони відносяться до так званого *виведення типу (type inference)* локальної змінної, що передбачає визначення компілятором типу змінної за типом значення, присвоєного цій змінній при ініціалізації. Зокрема, оголосити змінну, що зберігатиме ім'я можна кількома способами:

```
val name: String = "Stanislav"
val name = "Stanislav"
var name: String = "Stanislav"
var name = "Stanislav"
```

Серед запропонованих варіантів записи першої пари за своєю суттю однакові. Так само і для другої пари, оскільки компілятор автоматично виводить

тип String, що дозволяє не записувати його явно. Проте строга типізація мови Kotlin забезпечує неможливість присвоєння для цієї змінної в подальшому значення з несумісним типом. Зокрема, якщо надалі присвоїти змінній число 30, компілятор сповістить про помилку (рис. 1.2).

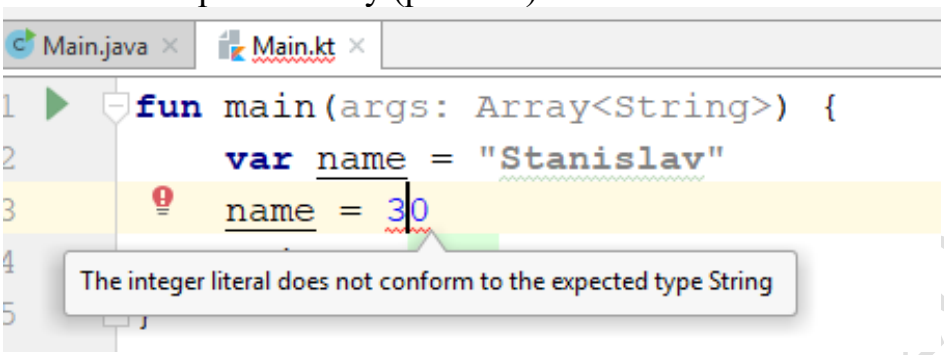


**Рис. 1.2. Помилка при присвоєнні значення несумісного типу**

Зауважте, що в коді використовувалось ключове слово var, яке допускає зміну значення ініціалізованої змінної в подальшому тексті програми. У свою чергу, ключове слово val використовується для позначення констант і є подібним до ключового слова final у мові програмування Java.

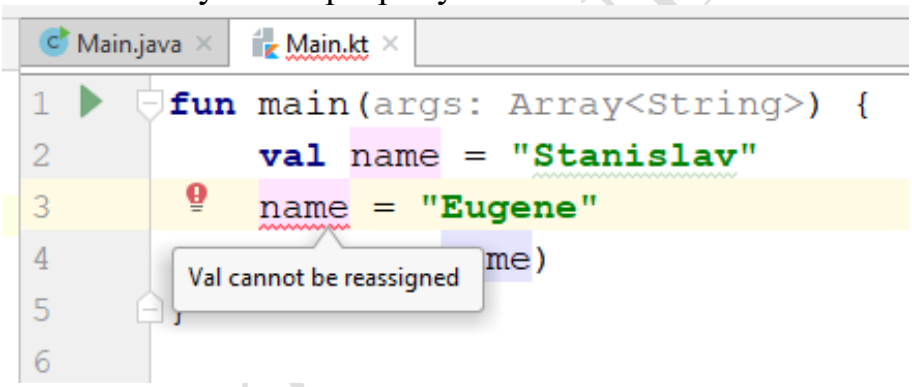


**Рис. 1.3. Помилка при спробі змінити значення змінної, оголошеної за допомогою ключового слова val**

Зазначені ключові слова описують поняття змінюваності (mutability) змінних. Застосовуючи ключове слово val, можна створити незмінювану (immutable) змінну. Рекомендується використовувати саме такий підхід у всіх випадках, коли це можливо. Проте якщо розробник передбачає зміну значення змінної під час виконання програми, слід використовувати ключове слово var. Ініціалізацію змінних можна відкласти, окремо виконуючи оголошення:

```
val name: String
name = "Stanislav"
```

Розглядаючи роботу зі змінними, потрібно згадати й типи даних, що присутні в мові програмування Kotlin. Їх перелік наведено в таблиці 1.1.

Суфікси, які уточнюють числовий тип, можуть записуватись як маленькими, так і великими літерами, тобто 14.16f = 14.16F. Примітивні типи даних, що були характерні для мови Java, відсутні в мові Kotlin. Також система

типів Kotlin допускає типи даних з підтримкою null-значення – так звані ***нулабельні типи (nullable types)***. У мові Java усі змінні непримітивних типів даних теж могли набувати такого значення, що часто призводило до появи виключення NullPointerException (NPE), як показано на рис. 1.4.

***Таблиця 1.1. Примітивні типи даних у мові Kotlin***

| Назва типу | Розмір у бітах | Приклад значень |
|---|---|---|
| Boolean | 1 | true, false |
| Byte | 8 | -124, 127, 0 |
| Char | 16 | 'c', '7', ':', '\uFA0B', \n' |
| Short | 16 | 32765, -32767, -791 |
| Int | 32 | 2147483645, -2147483645, 1_000_000 (знак підкреслення застосовується для зручності) |
| Long | 64 | 1234_5678_9000_0000L, -10000000000 |
| Float | 32 | 2.7182818284f (зберігається 2.7182817) |
| Double | 64 | 2.7182818284 |

```
1  public class Main {
2      static String name = null;
3
4      public static void main(String[] args) {
5          System.out.println(name.length());
6      }
7  }
"C:\Program Files\Java\jdk-9.0.4\bin\java.exe" "-javaagent:
Exception in thread "main" java.lang.NullPointerException
    at Main.main(Main.java:5)
```

**Рис. 1.4. Демонстрація NullPointerException у мові програмування Java**

**Завдання**

*Продемонструйте оголошення та ініціалізацію змінних, що мають типи з таблиці 1.1. Виведіть їх значення на екран. Для цього можете використовувати як оператор конкатенації +, так і шаблонні вирази, що починаються зі знаку $. Наприклад, вивести цілочисельну змінну k зі значенням 100 можна так:*

```
val k = 100
println("k = $k")
println("k = " + k)
```

Безпечність мови програмування Kotlin, зокрема, забезпечується спеціальним синтаксисом для явної підтримки нулабельності типу. За умовчанням, усі типи даних в Kotlin не є нулабельними. Якщо розробник передбачає можливість набуття такого значення змінною, потрібно дописати знак питання до назви типу. На цьому етапі виникає проблема: для нулабельного типу не підтримуються стандартні атрибути його ненулабельної версії, тобто length неможливо викликати, як це показано на рис. 1.5.

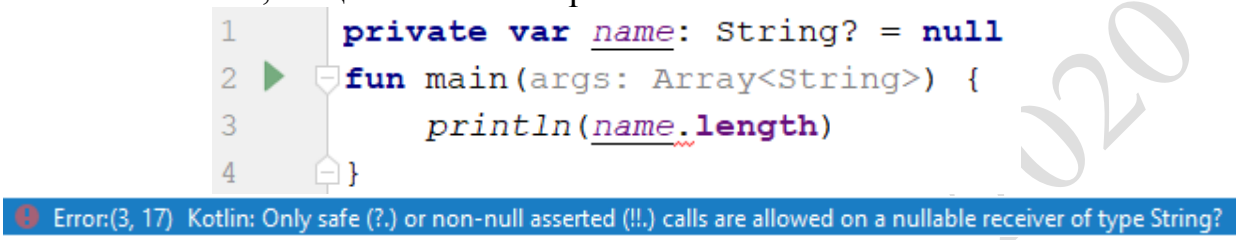


**Рис. 1.5. Неможливість застосування стандартних атрибутів до нулабельного типу**

Компілятор Kotlin пропонує два виходи з цієї ситуації:

– виконати безпечну перевірку на null, дописавши до назви змінної знак питання: name?.length. Тоді в консоль буде виведено значення null, помилки компіляції відсутні. Java-реалізація подібної функціональності вимагає застосування оператору if-else;

– придушити попередження про можливу небезпеку та, в даному випадку, отримати NullPointerException шляхом використання оператору “!!”: name!!.length. Такий оператор буде корисним, якщо розробник гарантує неможливість набуття змінною значення null протягом виконання програми. Навіть запис оператору як двох знаків оклику звертає увагу на небезпечність такого підходу.

Таким чином, можемо безпечно працювати з нулабельними типами, застосовуючи оператор безпечного доступу “?.”, в тому числі, для ланцюжкового виклику методів. Наприклад, текстове представлення довжини рядка можна зберігати так:

```
val length: String? = name?.length?.toString()
```

Додаткові можливості пропонує оператор “Елвіс” – “?:”. Його задача – замінити null-значення при його появі на певний нейтральний результат. Припустимо, довжина рядка null дорівнює нулю. Тоді можемо замінити null на 0:

```
val length = name?.length?:0
println("${name?.length?:0}")
```

Зверніть увагу, що вивід значень, отриманих з використанням операторів доступу вимагає після знаку $ огортати змінну фігурними дужками.

Кожний числовий тип підтримує зведення до іншого числового типу за допомогою методів з самоописовими назвами: toByte(), toShort(), toInt(), toLong(), toFloat(), toDouble(), toChar(). Явна типізація мови програмування Kotlin вимагає використання цих методів за потреби конвертації типів:

```
val si: Short = 100
val i: Int = si                 // помилка!!!
val i: Int = si.toInt()         // нормальна робота
```

Числові тпии підтримують стандартний набір математичних операцій. Особливими для мови Kotlin є побітові операції – спеціальних символів, як у Java, для них передбачено, проте використовуються функції, які викликаються в інфіксній формі: shl(біти) – знаковий побітовий зсув уліво, shr(біти) – знаковий побітовий зсув управо, ushr(біти) – беззнаковий побітовий зсув управо, and(біти) – побітовий “І”, or(біти) – побітове “АБО”, xor(біти) – побітове “виключне АБО“, inv() – інвертування бітів. Використання даних методів набуває подібного вигляду:

```
val number = (41 ushr 3) or 0xAA9
println(number.inv())
```

**Умовні оператори в мові програмування Kotlin**

У мові Java присутні наступні оператори мови програмування:

- умовний оператор if;
- оператор галуження switch;
- цикл while;
- цикл do-while;
- цикл for та його спеціальна версія for-each;
- оператор break;
- оператор continue;
- оператор return.

Більшість з них також доступні у мові Kotlin, проте основна відмінність полягає в тому, що тут вони є ***виразами***, а не операторами мови програмування. Це дозволяє впроваджувати цікаві синтаксичні конструкції.

Почнемо з умовного оператору if, який у своїй базовій формі не відрізняється від аналогічного у мові програмування Java. Проте представлення результату роботи оператору в формі виразу дає можливість, наприклад, присвоїти цей результат змінній. Розглянемо просту задачу щодо оцінювання студента з предмету. Студент вважається атестованим, якщо набирає 60 балів, неатестованим з правом перездачі, якщо отримує понад 35 балів, та змушений заново пройти курс у випадку підсумкового балу, нижчого за 35. Стандартна реалізація мовою Kotlin представлена в лістингу 1.3.

***Лістинг 1.3. Умовний оператор if***

```
fun main() {
    val mark: Int = 45
    val passed: String
    if (mark >= 60) {
        passed = "атестовано"
    }
    else if (mark >= 35) {
        passed = "не атестовано, з правом перездачі"
    }
    else {
        passed = "не атестовано, без права перездачі"
    }
    println("За результатами сесії студента $passed")
}
```

Проте існує можливість компактніше записати представлений код, оскільки кожна вітка повертає значення passed. Реалізація умовного оператору як виразу дозволяє перекписати код, як показано в лістингу 1.4. Разом з тим, у блоках оператору можна виконувати додаткові дії. Наприклад, додати бали після перездачі або записати нові бали після повторного проходження курсу. Слід зауважити, що необхідною умовою роботи подібного коду є те, що значення виразу повинно бути записано останньою інструкцією в блоці.

***Лістинг 1.4. Умовний оператор if як вираз***

```
fun main() {
    val mark: Int = 45
    val passed =
            if (mark >= 60) {
                "атестовано"
            }
            else if (mark >= 35) {
                val new_mark = 61
                "не атестовано, з правом перездачі.\n"+
                "Бали після перездачі - $new_mark"
            }
            else {
                val new_mark = 70
                "не атестовано, без права перездачі.\n"
                + "Бали після повторного проходження -
                $new_mark"
            }
    println("За результатами сесії з предмету студента
            $passed")
}
```

Робота з операторами мови програмування як виразами має свої особливості. Зокрема, дозволяється, щоб результат кожного блоку мав різний тип. Наприклад, один блок повертатиме рядок, інший – ціле число, ще один – дробове тощо. Проте загальний результат роботи операторубуде зводитись до типу, який виступатиме базовим (батьківським) для типів даних результатів окремих блоків.

Зокрема, для значень типів Int та String спільним базовим типом буде Any – аналог типу Object з мови програмування Java. Очевидно, що функціональність такого результату буде значно вужчою за функціональність породжених типів даних, тобто багато методів можуть бути недоступними. Крім того, слід не забувати і про нулабельність типів, яка розглядалась вище.

Якщо результат блоку порожній, у мові програмування Java пропонувався тип void для позначення типу даних при поверненні такого результату. Аналогом у мові Kotlin є тип Unit. Якщо видалити з блоків умовного оператору в лістингу 1.4 значення, які представляють результати (рядки 5, 9-10, 14-16), тоді вивід буде наступним: За результатами сесії з предмету студента kotlin.Unit.

Оператор галуження switch-case з мови програмування Java має прямий аналог у мові Kotlin – оператор when. Розглянемо приклад, у якому потрібно буде вивести, скільки днів у деякому місяці (лістинг 1.5). Якщо в Java-коді доводилось для кожного випадку писати case та не забувати оператори break, мова Kotlin пропонує компактнішу версію – оператор -> (стрілка). Також, на відміну від ключового слова default для мови Java застосовується ключове слово else. Його присутність необхідна задля коректної роботи всього оператору.

***Лістинг 1.5. Умовний оператор if***

```
1   fun main() {
2       val monthName = "Лютий"
3       val isLeapYear = true
4       val check =
5               when(monthName) {
6                   "Січень", "Березень", "Травень",
7                       "Липень", "Серпень", "Жовтень",
8                       "Грудень" -> {"31"}
9                   "Квітень", "Червень", "Вересень",
10                      "Листопад" -> {"30"}
11                  "Лютий" -> {if(isLeapYear) {"29"}
12                              else {"28"}
13                  }
14                  else -> {"Такого місяця немає!"}
15              }
16      println("Днів у місяці $monthName: $check")
17  }
```

**Оператори циклу та безумовного переходу в мові програмування Kotlin**

Оператори циклу частіше за все пов'язані з послідовним повторенням дій та роботою з колекціями даних. Тому почнемо розгляд з масивів як найпростіших статичних наборів даних фіксованої довжини. На початку теми було зазначено, що стандартний для Java синтаксис із квадратними дужками дещо видозмінений у мові Kotlin. Зокрема, впроваджено спеціальний тип Array, а для ініціалізації масивів передбачено метод arrayOf(). Слід зауважити, що методи з подібною назвою (тип_колекції + Of) характерні й для інших колекцій. Для доступу до елементів використовується такий же синтаксис, як і в мові Java. У лістингу 1.6

демонструється основна робота з масивами в мові Kotlin. Також слід зауважити, що Kotlin-масиви використовують дженерики (алмазний синтаксис), тому масив значень примітивного типу неможливо створити подібним способом. Для цього мова Kotlin пропонує спеціальні класи IntArray, DoubleArray та ін.

У рядках 8-10 показано аналог циклу for-each з мови програмування Java. Синтаксично вони відрізняються заміною оператора ":" на оператор in та відсутністю типу ітератора (тут – змінна i). У той же час підтримка циклу з лічильником теж працює на основі ітератора, що продемонстровано в рядках 11-13. Зверніть увагу, що діапазон значень для лічильника задається за допомогою оператора "..". Існує альтернативний оператор until, який має невключну верхню межу, а також колекція indices, яка містить набірдоступних індексів. Тоді рядок 11 можна переписати дещо простіше в одній з двох форм:

```
(1)    for(i in 0 until array2.size)
(2)    for(i in array2.indices)
```

***Лістинг 1.6. Базова робота з масивами***

```
1  fun main() {
2      val array1 = arrayOf(12, 7, -4, 21,615)
3      val array2: Array<String?> =
4              arrayOf("Масив", "рядків", "array2", null)
5      val array3 = arrayOfNulls<Int?>(5)
6      array3[0] = 3
7      array3[3] = 10
8      println("Третє значення у масивах: " +
9              "${array1[2]}, ${array2[2]}, ${array3[2]}")
10     for(i in array1) {
11         println(i)
12     }
13     for(i in 0 until array2.size) {
14         println(array2[i])
15     }
16     for(i in array1.size-1 downTo 0 step 2) {
17         println(array1[i])
18     }
19     for((index, i) in array3.withIndex()) {
20         array3[index] = i?.times(5)
21         println(array3[index])
22     }
23 }
```

Якщо лічильник потрібно спрямувати в напрямку зменшення (декременту), використовується оператор downTo, як показано в рядку 14. Також передбачена можливість задати ітераційний крок за допомогою оператору step, після якого через пробіл встановлюється значення кроку. Такий синтаксис можна застосовувати як при інкрементації, так і декрементації лічильника.

Часто лічильники та значення бажано використовувати разом. У рядках 17-19 показано код, який дозволяє замінити значення масиву array3 на п'ятикратно збільшені. Зверніть увагу, що в даному контексті маємо нулабельний тип цих значень, тому стандартне множення за допомогою оператора "*" неможливе. Слід використовувати спеціальний метод times(), який приймає відповідний множник.

Цикли while та do-while працюють у тій же манері, що й у мові програмування Java. Обмежимось демонстрацією роботи цикла while на прикладі знаходження суми введених користувачем чисел. Передбачаємо, що при введенні нуля слід припинити подальший ввід інформації та вивести суму чисел (лістинг 1.7). Зверніть також увагу на оператор break, який перериває роботу нескінченного циклу.

***Лістинг 1.7. Демонстрація роботи циклу while***

```
1   fun main() {
2       var sum = 0
3       while(true) {
4           val number = readLine()!!.toInt()
5           if (number == 0)
6               break
7           sum += number
8       }
9       println(sum)
10  }
```

Дещо відрізняється від Java-версії синтаксис використання маркованих операторів безумовного переходу (labeled break, labeled continue). Мітка для безумовного переходу записується у вигляді ідентифікатора з символом "@" наприкінці, як показано в лістингу 1.8.

***Лістинг 1.8. Маркований оператор break***

```
1   fun main() {
2       loop@ for (i in 1..100) {
3           for (j in 1..100) {
4               if (<умова>) break@loop
5           }
6       }
7   }
```

## Основи роботи з колекціями

Внутрішня реалізація колекцій повністю покладається на Java Collections Framework, основу для колекцій складають інтерфейси List (списки), Set (множини – колекції унікальних елементів) та Map (відображення, мепи – колекції на основі пар "ключ-значення", аналоги словників у інших мовах програмування). Основною відмінністю від Java Collections Framework стало явне розмежування незмінюваних (маніпулювання вмістом колекцій заборонене) та змінюваних (mutable) колекцій. На рис. 1.10 зображено базові інтерфейси колекцій у мові програмування Kotlin. Загалом інтерфейс Iterable пропонує ітератор, який використовується для перебору колекції. Інтерфейс Collection

доповнює цю функціональність можливостями зчитування даних, перевірки наявності елементу тощо. Саме в ньому визначені:

- size – кількість елементів у колекції;
- isEmpty() – перевірка колекції на порожність (значення true);
- contains(елемент) – перевірка наявності елементу (повертає true);
- containsAll(колекція) – перевірка наявності елементів однієї колекції в іншій.

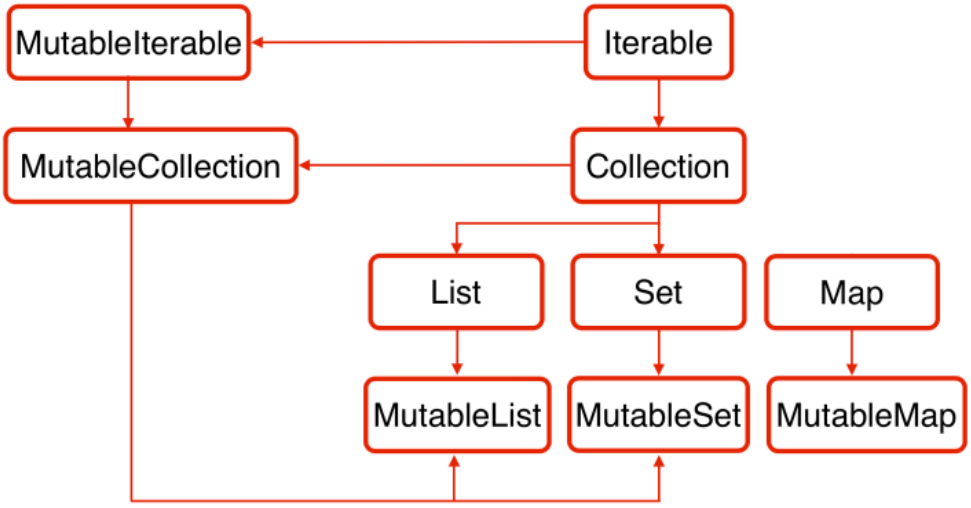


**Рис. 1.10. Інтерфейси колекцій в мові програмування Kotlin**

Як уже зазначалось, особливістю колекцій в мові Kotlin є їх незмінюваність за умовчанням. Тобто списки, утворені на базі типу List, не дозволяють змінювати набір своїх даних. Для підтримки маніпулювання вмістом колекцій вводяться спеціальні класи, які легко вирізнити за наявністю «Mutable» у назві. Тому замість типу List у більшості випадків будуть використовуватись змінні типу MutableList. Аналогічна ситуація з множинами (Set та MutableSet) і відображеннями (Map та MutableMap). На рис. 1.11 показано приклад ініціалізації стандартного та змінюваного списків, а також невдалу спробу зміни значення елемента незмінюваного списку. Масиви та списки в мові Kotlin забезпечують безпеку типів при компіляції, яка була недоступна для мови Java. Загальною рекомендацією є використання списків, а не масивів, у всіх випадках, за винятком частин коду, для яких критична швидкодія.

```
3  fun main() {
4      val list : List<String>  = listOf("один", "два", "три")
5      val mutableList : MutableList<String>  =
6              mutableListOf("1", "2", "3")
7      mutableList[0] = "0"
8      list[0] = "нуль"
```

No set method providing array access

Unresolved reference.
None of the following candidates is applicable because of receiver type mismatch:

- @InlineOnly **public inline operator fun** <K, V> MutableMap<Int, String>.set(key: Int, value: String): Unit *defined in* kotlin.collections
- @InlineOnly **public inline operator fun** kotlin.text.StringBuilder /* = java.lang.StringBuilder */.set(index: Int, value: Char): Unit *defined in* kotlin.text

**Рис. 1.11. Ініціалізація та зміна значень елементів списків**

Масиви та списки мають кілька суттєвих відмінностей:

– масиви є змінюваними, а списки за умовчанням – ні;

– масиви мають окремі оптимізовані версії для примітивних типів (IntArray, DoubleArray, CharArray та ін.), які напряму відображаються на Java-масиви примітивних значень. Списки такими оптимізаціями не володіють;

– масив виступає контейнером фіксованого розміру та реалізується за допомогою узагальненого класу. Списки описуються за допомогою інтерфейсів з різними реалізаціями (ArrayList<T>, LinkedList<T> тощо).

Списки підтримують поширені операції для отримання окремих елементів, продемонстровані в лістингу 1.9. Більшість назв методів є самоописовими. Зверніть увагу на умовне отримання елементів у рядках 14-18. Для задавання умови використовується спеціальний синтаксис лямбда-виразів, який детальніше описаний в наступній темі. Для перевірки наявності елемента в списку можливе використання оператора in або методів contains(), containsAll(). У рядку 21 відбувається одночасний пошук двох елементів у списку, причому порядок цих елементів може не зберігатись.

***Лістинг 1.9. Операції отримання елементів списку***

```
1  fun main() {
2      val numbers = listOf(1, 8, 4, 12, -6, 0, 21)
3      println(numbers.get(0))
4      println(numbers[0])          // виняток при get(7)
5      println(numbers.subList(2, 4))
6      println(numbers.first())
7      println(numbers.last())
8      println(numbers.elementAt(4))
9      println(numbers.random())
10     println(numbers.elementAtOrNull(7))
11     println(numbers.elementAtOrElse(7) {
12         index -> "Значення за індексом $index не визначене"
13     })
14     println("Перше число>3: ${ numbers.first { it > 3 } }")
15     println("Перше число>3 або null: ${ numbers.find {
16                                             it > 3 } }")
17     println("Останнє число>3 або null: ${ numbers.findLast {
18                                             it > 3 } }" )
19     println(3 in numbers)           // false
20     println(numbers.contains(3))    // false
21     println(numbers.containsAll(listOf(8, 1))) ])  // true
22     println(numbers.isEmpty())
23     println(numbers.isNotEmpty())
24 }
```

Також списки володіють вбудованими можливостями для пошуку елементів відносно їх позицій. Методи indexOf(), lastIndexOf(), indexOfFirst(), indexOfLast() повертають індекс відповідного елементу або -1 за умови його відсутності в списку. В їх основі лежить алгоритм лінійного пошуку, проте для відсортованих списків також доступний метод binarySearch() на базі бінарного пошуку. Крім

цього, слід не забувати про особливості порівняння Comparable-об'єктів: потрібно забезпечувати реалізацію інтерфейса Comparator. Впорядкування елементів змінюваного списку підтримується спеціальними функціями-розширеннями sort(), sortDescending(), sortBy(), shuffle(), reverse() та ін.

Множини (сети) є представленням у коді математичних множин. Вони мають набір власних, специфічних операцій (рис. 1.12): об'єднання (union), переріз (intersect) та різниця (subtract). Множина не допускає повторення елементів, а також за своєю природою невпорядкована. Лістинг 1.10 показує використання даних операцій. Також доступні можливості доповнення множин елементами та відповідне видалення. При цьому, при додаванні існуючого або видаленні неіснуючого елементів винятки не викидаються.

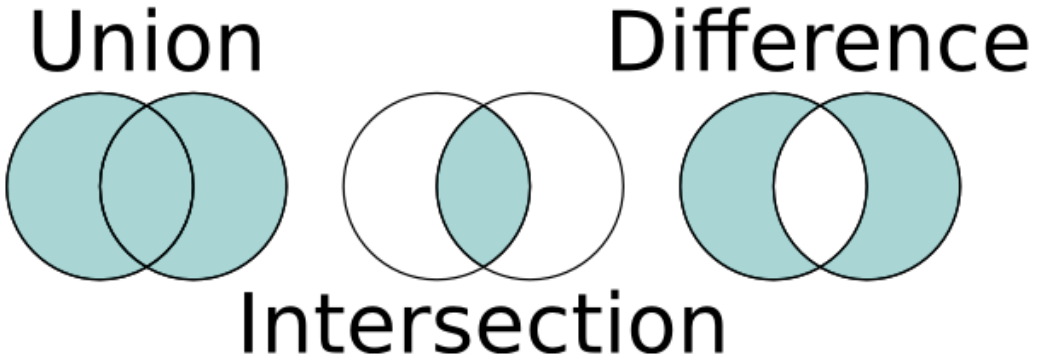


**Рис. 1.12. Операції з математичними множинами**

***Лістинг 1.10. Специфічні операції з множинами***

```
fun main() {
    val numbers1 = setOf("один", "два", "три")
    val numbers2 = setOf("чотири", "п'ять")

    println(numbers1 union numbers2)
    println(numbers2 union numbers1)
    println(numbers1)
    println(numbers1 intersect setOf("два", "один"))
    println(numbers2 subtract setOf("три", "чотири"))
    println(numbers1 subtract setOf("чотири", "три"))
    println(numbers1.minusElement("два"))
    println(numbers1.plusElement("один"))
}
```

У результаті запуску коду буде виведено:

*[один, два, три, чотири, п'ять]*
*[чотири, п'ять, один, два, три]*
*[один, два, три]*
*[один, два]*
*[п'ять]*
*[один, два]*
*[один, три]*
*[один, два, три]*

Ще одним типом колекцій у мові Kotlin є мепи (відображення, словники), які зберігають дані у вигляді пар "ключ-значення". Тому меп володіє власними операціями для заповнення та отримання даних. Наприклад, при ініціалізації мепу зазвичай використовується інфіксний запис (інфіксна функція to, рядок 2 лістингу 1.11), проте за використання хешмепу (відображення класу HashMap з мови Java) елементи переважно додаються за допомогою синтаксису, подібного до роботи з масивами (рядок 3).

***Лістинг 1.11. Операції з мепами***

```
1   fun main() {
2       val symbols = mapOf(97 to 'a', 98 to 'b', 99 to 'c')
3       val chars = HashMap<Int, Char>()
4       chars[100] = 'd'
5       chars[101] = 'e'
6       println("Keys:" + symbols.keys)
7       println("Values:" + symbols.values)
8       for ((k, v) in symbols.entries) {
9           println("$k = $v")
10      }
11      println("У мепі ${symbols.size} пар")
12      println("У мепі ${symbols.count()} пар")
13      println(symbols[98])
14      println(symbols.get(98))
15      println(symbols.getValue(98))
16      println(symbols.getOrDefault(98, -1))
17      println(symbols.getOrElse(98, {
18          -1
19      }))
20  }
```

Усі ключі мепу можна отримати за допомогою властивості keys, а значення – values. Лістинг 1.11 передбачає цілочисельні ключі, проте загалом їх тип може бути довільним. Відповідні пари (entry) доступні за умови використання властивості entries. У рядку 8 дана властивість застосовується лише для демонстрації, тут програмний код працюватиме і при простому зверненні до мепу symbols від оператора in. Кількість елементів мепу можна визначити як за допомогою методу count(), так і при зчитуванні значення властивості size. Використання методу більш універсальне, оскільки можливо задати умови відбору елементів та підрахувати їх кількість (синтаксис аналогічний для методу find() для списків). Для отримання окремих пар, ключів або значень також доречні стандартні методи get(), getValue(), getOrDefault(), getOrElse(). Середовище розробки рекомендує використовувати звернення в стилі рядка 13, а за умови постачання альтернативних значень (у даному коді -1) чи виразів обирати методи getOrDefault() або getOrElse() відповідно. Для перевірки наявності певних даних у мепі за аналогією застосовуються спеціальні методи containsKey() та containsValue().

## Основи роботи з функціями

Під функцією маємо на увазі іменований блок коду, в який може передаватись та з якого може повертатись інформація. Нагадаємо синтаксис стандартної функції:

```
fun назва_функції (параметри функції): тип_функції {…}
```

Для повернення даних з функції використовується оператор return, який повністю повторює свій Java-еквівалент. Проте трактування умовних операторів як виразів дозволяє дещо зручніше організовувати функції зсередини. Наприклад, у лістингу 1.12 розглянемо функцію, яка знаходить мінімум двох чисел, які їй передаються

***Лістинг 1.12. Функція для знаходження мінімуму двох чисел***

```
1   fun min(a:Int, b:Int):Int {
2       return  if(a < b) { a }
3               else { b }
4   }
5   fun main() {
6       println(min(8,5))
7   }
```

Якщо всередині функції виконується всього одна дія, код з лістингу 1.12 можна ще більше спростити:

```
fun min(a:Int, b:Int) = if(a < b) { a } else { b }
```

Тип функції (return type) в даному випадку можна не вказувати, а замість фігурних дужок та оператору return записати оператор "=". Однорядковий запис функцій також буде корисним при розгляді функціонального програмування.

Функції в мові Kotlin дозволяють передавати змінну кількість аргументів за допомогою ключового слова vararg. Крім того, новими можливостями стали ***іменовані аргументи (named argument)*** та ***аргументи за умовчанням (default argument)***. У цілому, подібна функціональність також присутня у мові програмування Python, для якої розробники мови Kotlin – компанія JetBrains – також розвивають середовище розробки PyCharm. У лістингу 1.13 показані дані можливості роботи з функціями. Перша функція, isum(), дозволяє додавати довільну кількість цілих чисел, оскільки має у своїй сигнатурі (рядок 1) ключове слово vararg. Аргументи передаються у функцію у вигляді колекції даних, з якою потім іде робота. Функція printFullName() потрібна для виведення ПІБ особи, проте не кожна людина має по батькові, тому такий параметр стає необов'язковим. У коді це демонструється за допомогою присвоєння порожнього рядка параметру secondName (рядок 10), що називають аргументом за умовчанням. Для більшої універсальності програми всі три параметри записані як аргументи за умовчанням. Також це значить, що доступна можливість взагалі не

передавати такі аргументи. Зокрема, у виклику функції printFullName() передаються тільки ім'я та прізвище.

***Лістинг 1.13. Можливості роботи з параметрами функцій***

```
1   fun isum(vararg numbers:Int):Int {
2       var res = 0
3       for (num in numbers) {
4           res += num
5       }
6       return res
7   }
8
9   fun printFullName(firstName: String = "",
10       lastName: String = "", secondName: String = "") {
11      if(firstName.isNotEmpty()) {
12          println("Ім'я: $firstName")
13      }
14      if(secondName.isNotEmpty()) {
15          println("По-батькові: $secondName")
16      }
17      if(lastName.isNotEmpty()) {
18          println("Прізвище: $lastName")
19      }
20  }
21
22  fun main() {
23      printFullName(lastName = "Марченко",
24                    firstName = "Станіслав")
25      println(isum(4, 6, -9, 0, 17))
26  }
```

Іменовані аргументи проявляються в рядках 23-24, де перед значення дописується назва параметру функції, якій це значення відповідає. Таким чином, можна передавати значення в довільному порядку. Програмний вивід для лістингу 1.13 наступний:

*Ім'я: Станіслав*
*Прізвище: Марченко*
*18*

Мова програмування Kotlin дозволяє вкладати функції одна в одну, тобто робити функції локальними. Вкладену функцію можливо викликати тільки з зовнішньої функції (лістинг 1.14), проте їй доступні дані з зовнішньої фукції. Зокрема, радіус не передається окремим параметром у відповідні локальні функції для знаходження довжини кола, площі кругу та об'єму кулі. Такий підхід робить код більш модульним та чистішим.

***Лістинг 1.14. Приклад використання локальних функцій***

```
1   fun printFeatures(radius: Double) {
2       fun circumference() = 2.0 * Math.PI * radius
3       fun area() = Math.PI * Math.pow(radius, 2.0)
4       fun volume() = 4.0 * Math.PI / 3 * Math.pow(radius, 3.0)
5       println(circumference())
6       println(area())
7       println(volume())
8   }
9
10  fun main() {
11      printFeatures(2.5)
12  }
```

**Завдання**

*Розгляньте реалізації рекурсивних функцій та викликів у мові програмування Kotlin. Напишіть рекурсивну функцію, яка буде виконувати піднесення числа x до степені n за відповідними правилами:*

$$x^n = \begin{cases} 1, & \text{якщо } n = 0 \\ \left(x^{\frac{n}{2}}\right)^2, & \text{якщо } n > 0,\ n - \text{парне} \\ x \cdot x^{n-1}, & \text{якщо } n > 0,\ n - \text{непарне} \\ \frac{1}{x^n}, & \text{якщо } n < 0 \end{cases}.$$

## Практичні завдання

1. ***0,3 бала*** Напишіть програму, яка за введеним користувачем текстом виведе його абревіатуру.
2. ***0,4 бала*** Знайдіть у мережі Інтернет рейтинг найсильніших алергенів та трансформуйте його в меп для своєї програми (алерген – рівень загрози у балах). Зімітуйте трьох людей з однією або кількома алергіями (кількість обирається випадковим чином) та підрахуйте відповідний алергенний рівень (суму балів). Виведіть на екран інформацію щодо кожної людини.
3. ***0,5 бала*** Напишіть програму, яка виконуватиме шифрування та дешифрування україномовного тексту за допомогою шифру Атбаш.
4. ***0,5 бала*** Напишіть програму, яка обчислить, скільки коштів потрібно витратити на пофарбування кімнати. Користувач повинен вводити лінійні розміри кімнати (довжину, ширину та висоту) в метрах. На основі цих розмірів слід обчислити необхідну площу для фарбування. Враховуйте, що кімната має два вікна з розміром 1.4×1.2 м та двері розміром 2.0×1.2 м. Додайте перевірку на коректність введених розмірів (одна стіна має вміщати обидва вікна, інша – двері, всі розміри мають бути невід'ємними).
   Вважаємо, що для фарбування 1м2 витрачається в середньому 0.12л фарби. Доступні три варіанти банок з фарбою:

- 0,5л – за 65.80 грн;
- 1л – за 125.20 грн;
- 2л – за 240 грн.

Визначте оптимальну кількість банок фарби кожної місткості та загальну суму витрат.

5. ***0,5 бала*** Напишіть функцію, яка обчислюватиме відстань Геммінга для двох слів. Забезпечте коректну роботу функції для слів різної довжини.
6. ***0,4 бала*** Напишіть функцію, яка буде підраховувати кількість очків, отриманих від кидання дротика в дартсі. Правила гри передбачають кілька зон, у які може потрапити дротик після кидка (рис. 1.11):
   - яблучко (50 очків);
   - зелене кільце навколо яблучка (25 очків);
   - зовнішнє вузьке кільце (подвоює очки відповідного сектора);
   - внутрішнє вузьке кільце (потроює очки відповідного сектора);
   - сектор (відповідна кількість очків).

   Задійте генератор випадкових чисел, щоб зімітувати кидки в різні зони мішені. Здійсніть три кидка, виведіть кількість очків за кожен кидок та сумарну кількість очків.

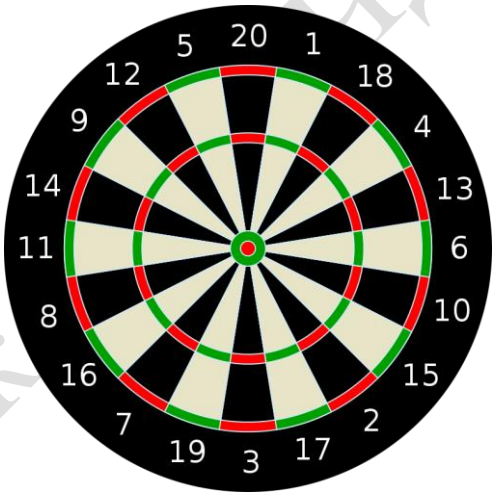


**Рис. 1.13. Мішень для гри в дартс**

7. ***0,4 бала*** *(Калькулятор тарифів Нової Пошти)* Компанія «Нова Пошта» пропонує таблицю з базовими тарифами на перевезення відправлень. Зверніть увагу, що вага відправлення встановлюється не шляхом зважування, а за формулою об'ємної ваги:

***Об'ємна вага (кг)*** = (Довжина(см) × Ширина(см) × Висота(см)) / 4 000

або об'єм вантажу (м³) × 250.

Також на вартість відправлення впливають тарифні зони (в межах міста, області та України (1-4)). Крім того, слід враховувати, чи замовляв клієнт адресний забір або доставку, а також оголошену вартість (якщо понад

200грн, до загальної вартості відправлення додається 0.5% від оголошеної вартості).

Напишіть програму, яка прийматиме на вхід *розміри пакунка* в сантиметрах, *початковий та кінцевий пункт або області* (доступні міста для відправки в межах міст), прапорець щодо адресної доставки та оголошену вартість. На вихід додаток повинен вивести об'ємну вагу відправлення, обрану тарифну зону та сформувати загальну вартість відправлення (вивести оплату за кожним параметром доставки та ітогову суму).