Програмування мобільних пристроїв

Основи мови програмування Kotlin

Слайди до лекцій (1 змістовий модуль)

Вступ - основні мобільні платформи

https://developer.android.com/ https://developer.apple.com/

Вступ - класифікація мобільних застосунків

Native

Hybrid

Web

Вступ - засоби програмування

Android Software Development Kit (SDK)

https://developer.android.com/

iOS Software Development Kit (SDK)

https://developer.apple.com/

Java

Kotlin

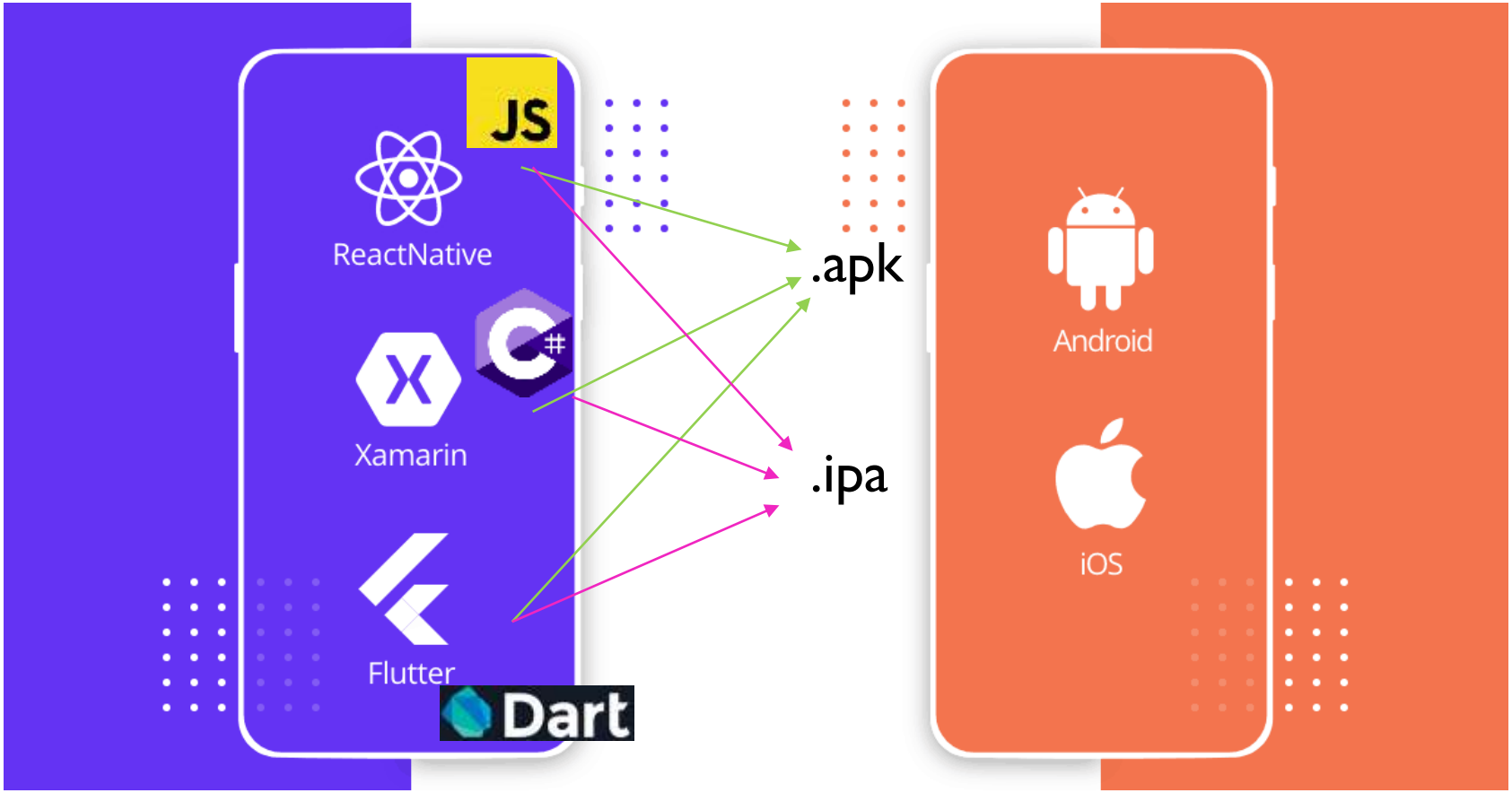
Android Studio

Objective - C

Swift

Xcode

Вступ - крос-платформні засоби програмування



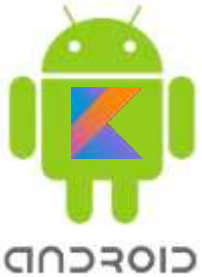
Kotlin - загальна характеристика

- Мова програмування Kotlin (якій надали ім'я як назву острову у Балтійському морі) розроблена у 2011 р компанією JetBrains

- Починаючи з Android Studio 3.0 у якості альтернативи Java Google додали мову програмування Kotlin

- У травні 2019 р на конференції Google I/O 2019 компанія оголосила Kotlin мовою програмування, якій віддається перевага при розробці Android-застосунків

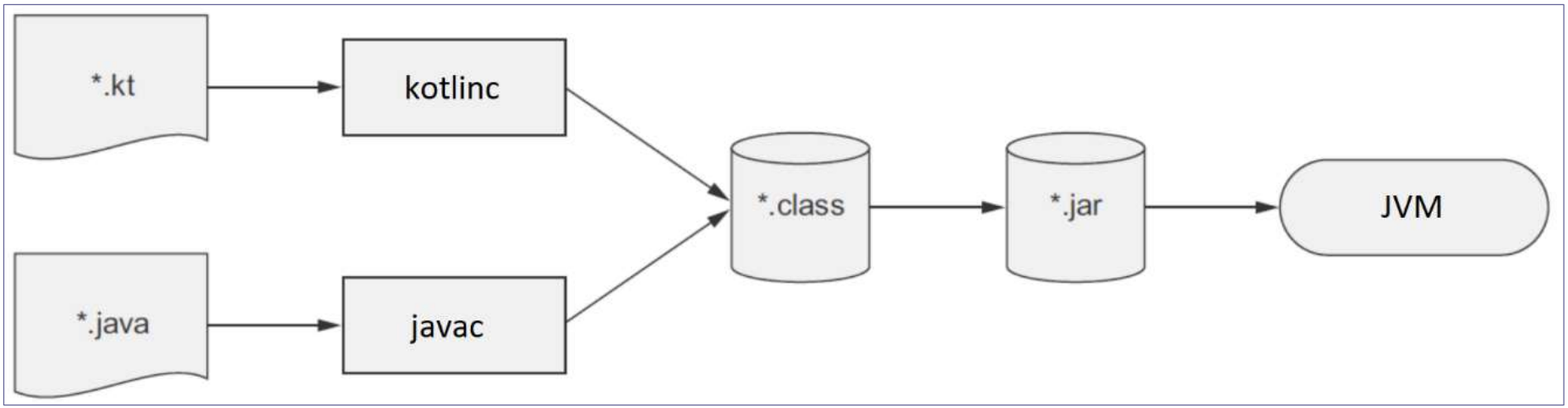
https://kotlinlang.org



Kotlin - переваги

- лаконічний та інтуїтивно зрозумілий синтаксис;
- підтримка безпечної роботи з посиланнями зі значенням null;
- виведення типів та автоматичне приведення типів при їх перевірці;
- малі накладні видатки при виконанні застосунків;
- підтримка, разом з об'єктно-орієнтованою, функціональної парадигми програмування;
- сумісність з кодом Java (традиційної мови програмування Android-застосунків);
- багатий вибір бібліотек/фреймворків (разом з бібліотеками/ фреймворками Java) та інструментів програмування;
- можливість вільного використання мови та відкритий вихідний код її інструментів (ліцензія Apache 2).
- Kotlin, як і Java, є *статично типизованою* та *компільованою* мовою програмування

Kotlin - компіляція та виконання програм



https://github.com/JetBrains/kotlin/releases

- Розпакувати завантажений архів до, наприклад, c:\Program Files\Kotlin
- Додати до PATH c:\Program Files\Kotlin\bin
- Перевірити у командному рядку: kotlinc -version

Kotlin - компіляція та виконання програм

```
package firstprogram

fun main() {
    println("Hello World!")
}
```

kotlinc <.kt файл> -include-runtime -d <ім'я jar-файлу>
java -jar <ім'я jar-файлу>

Відмінності від Java

Дослідження архіву

Kotlin - створення проєкту в IDE

Kotlin - базові типи даних

Назва типу Kotlin	Назва типу Java	Розмір, біт	Діапазон
Byte	byte	8	-128 ... 127
UByte	-	8	0 ... 255[1]
Short	short	16	-32768 ... 32767
UShort	-	16	0 ... 65535
Int	int	32	-2,147,483,648 (-2^{31}) ... 2,147,483,647 ($2^{31} - 1$)
UInt	-	32	0 ... 4,294,967,295 ($2^{32} - 1$)
Long	long	64	-9,223,372,036,854,775,808 (-2^{63}) ... 9,223,372,036,854,775,807 ($2^{63} - 1$)
ULong	-	64	0 ... 18,446,744,073,709,551,615 ($2^{64} - 1$)
Float	float	32	~±1.4e-45 ... ~±3.4e38
Double	double	64	~±4.9e-324 ... ~±1.8e308
Char	char	8-32	Unicode characters
String	-	-	Послідовність символів
Boolean	boolean	-	true or false (for Kotlin can not be null)
Array<T>	-	-	Клас, об'єкти якого представляють масиви

Kotlin має виключно посилальні типи даних

Kotlin - спеціальні типи даних

Any - кореневий клас ієрархії класів Kotlin - kotlin.Any.
Будь-який клас є нащадком Any (аналог класу Object Java).

Unit - тип, що має лише одне значення: об'єкт Unit - kotlin.Unit
Цей тип відповідає типу void в Java.

Nothing - Nothing не має екземплярів - kotlin.Nothing
Використовується для представлення «значення,
яке ніколи не існує»: наприклад, функція з типом повернення
Nothing ніколи не повертає значення (завжди викидає виняток).

Kotlin має виключно посилальні типи даних

Kotlin - оголошення змінних та констант

- Оголошення змінних Kotlin починається з ключового слова:
  var – для змінних, які можуть змінювати значення

```
var firstVar: Int
firstVar = 5
```

  val – для змінних, які не можуть змінювати значення
  (такі змінні при оголошенні повинні бути ініціалізовані)
  (аналог final-змінних Java)

```
val firstVal: Int = 7
```

- Тип змінної можна вказати після імені змінної,
  або не вказувати, якщо компілятор зможе вивести (infer) тип з наданого змінній значення:

```
var secondVar: Int = 7
val secondVal = 3123456L
```

Kotlin - особливості числових типів даних

- Класи числових типів успадковують абстрактний клас kotlin.Number з абстрактними функціями конвертації до усіх числових типів.
- Класи числових типів імплементують інтерфейс kotlin.Comparable<in T> з абстрактною функцією:
  fun compareTo(other: T): Int
  Наприклад:
  public class Int private constructor() : Number(), Comparable<Int> {...}
- При визначенні змінних типів Long, Float рекомендується вказувати суфікс L, F після значення літерала, у такому разі тип в оголошенні змінної буде виведений.
- При визначенні змінних беззнакових типів суфікс U після значення літерала - обов'язковий.

See BasicTypesDefinition

Kotlin - оголошення змінних та констант

- При виведенні значень змінних до консолі Kotlin використовує технологію *рядкових шаблонів* (*string templates*):
  println("firstVar= **$firstVar**, secondVal= **$secondVal**")
- Ця технологія дозволяє використовувати у якості шаблона не тільки ім'я змінної, а й вирази й функції, які повертають значення:
  println("Hello, **${if (args.size > 0)** args[0] else "someone"}!")

- Оголошення констант Kotlin виконується ключовими словами:
  const val *MY_CONST* = 5
- Константи при оголошенні повинні бути ініціалізовані
- Оголошення констант допускається тільки на верхньому рівні (рівні файлу/класу Kotlin)

See VarDefinition

Kotlin - тип даних Array<T>

- Тип Array<T> визначає масиви, клас цього типу має конструктор, функції get() та set(), які викликаються перевизначеним оператором [], цілочисельну властивість size та функцію, що повертає ітератор.
- Файл kotlin.Library містить функції для створення масивів різних типів - об'єктів внутрішніх класів файлу kotlin.Arrays , наприклад:
  val languages = *arrayOf*("Java", "Kotlin", "Clojure") - повертає Array<String>
  val arrInt = *intArrayOf*(1, 2, 3, 4, 5) - повертає IntArray
  val arrDbl = *doubleArrayOf*(1.5, 3.25, 5.15) - повертає DoubleArray
  ...
- Файл kotlin.collections._Arrays містить велику кількість перевантажених функцій для роботи з масивами різних типів

See ArrayRangeDefinition

Kotlin - робота з nullable типами

- При ініціалізації змінної Kotlin за замовчуванням встановлювати null заборонено:
  val specVar: Int = null
- Якщо необхідно зберігати null у змінних Kotlin, додається знак питання після типу при їх оголошенні:
  val specVar: Int? = null
- На відміну від інших мов, Kotlin слідкує за значеннями, які можуть дорівнювати null, щоб ви не намагалися виконувати з ними неприпустимі операції.
- Kotlin дозволяє працювати з nullable змінними без генерування NullPointerException:
  println(specVar) //null

See NullSafety

Kotlin - робота з nullable типами

- Для доступу до поля або виклику методу nullable змінної необхідна попередня перевірка на null (інакше компілятор показує помилку):

```
var b: String? = "hello"
b = null
//   println(b.length)
if (b != null) {
    println(b.length)   //null
}
```

- Для скорочення коду та уникнення виклику зі встановленої у null змінної після перевірки, Kotlin надає *оператор безпечного виклику функцій* (*Safe Call Operator)* ?.

```
println(b?.length)  //null - No NullPointerException!
println(b?. plus("world"))    //null - No NullPointerException!
```

See NullSafety

Kotlin - nullable параметри функцій

- У разі використання nullable змінної як параметра функції, компілятор покаже помилку:

```
val firstNumber = 10
val secondNumber: Int? = null
/*Type mismatch. Required: Int Found: Int?*/
// println(firstNumber.times(secondNumber))
```

- Для роботи цього коду необхідно або виконати попередню перевірку параметра на null,
або викликати метод за допомогою функції ***let*** :


```
secondNumber?.let {
    println("Multiplication result: ${firstNumber.times(it)}")
                                                    //not called
}
```

See NullSafety

Kotlin - ініціалізація nullable змінної

- Код ініціалізації nullable змінної часто не є компактним:

```
val myStr: String? = null
val notNullStr1 = if (myStr != null) {
    myStr
} else {
    "String is null"
}
println(notNullStr1)    //String is null
```

- Kotlin містить Елвіс-оператор (Elvis Operator) **?:**, який дозволяє зробити такий код значно компактнішим:

```
val notNullStr2 = myStr ?: "String is null"
println(notNullStr2)    //String is null
```

See NullSafety

Kotlin - not-null-ствердження

- Оператор not-null-ствердження (not-null assertion) **!!.** видаляє усі обмеження для nullable типу, перетворюючи його у схожий на Java тип (можна перевіряти умови виникнення проблем):

```
var b: String? = "hello"
b = null
//    val c : Char? = b!!.get(1)     //NullPointerException
```

- Kotlin дозволяє працювати з виключеннями способами, доступними Java, але додає можливість використовувати try-catch(-finally) блоки як вирази

```
val result = try {
    b?.toInt()
} catch (e: RuntimeException) {
    null
}
println("Try-catch result: $result")    //null
```

See NullSafety

Kotlin - перевірка та приведення типів

- Перевірка типу об'єкту при виконанні програми Kotlin виконується операторами **is** та **!is** (аналогічно оператору instanceof Java), при цьому виконується *розумне приведення типу (smart cast).*
- У разі, якщо об'єкт оголошений на рівні класу/файлу і є змінним, розумне приведення типу не може бути виконаним і компілятор показує помилку.
- У такому випадку можливе *явне приведення типу*:
  оператором **as** - *небезпечне (unsafe)* **-** у разі неможливості приведення генерується ClassCastException;
  або
  оператором **as?** - *безпечне (safe)* **-** у разі неможливості приведення повертається значення null для nullable об'єкта, який приводиться

See TypeChecksAndCasts

Kotlin - базові оператори

- Оскільки базові типи даних є класами, в них містяться методи, що реалізують різні операції, наприклад, **plus(other: Int)**, **minus(other: Int)**, **times(other: Int)** тощо, але вони перевантажені і визначені як звичайні оператори **+** , **-**, ***** тощо, які і рекомендується використовувати.
- Арифметичні оператори та логічні оператори повністю співпадають з такими у Java.
- Оператори перевірки на рівність **==** та **!=** перевіряють на рівність за вмістом, перевірка на рівність за посиланням виконується операторами **===** та **!==** (для базових типів обидва такі оператори працюють однаково).

See VarDefinition

Kotlin - базові оператори

- Цілочисельні бітові оператори у Kotlin не перевантажені у звичайні оператори Java (&, |, ^, ~, <<, >>, >>>) і повинні використовуватися як функції, що можуть викликатися з базових числових об'єктів або їх реалізації в інфіксній формі, наприклад:

```
val bitVar1 = 0b0111_1111
val bitVar2 = bitVar1.inv()
```

- Kotlin додає оператори визначення діапазону та перевірки, чи належить число до діапазону:

```
val myRange = 1..5    //implicit IntRange type variable
println(3 in myRange)     //true
println(7 in myRange)     //false
```

See BasicOperators

See ArrayRangeDefinition

Kotlin - оператори та вирази розгалуження

- Оператор розгалуження **if** у Kotlin може бути як *оператором* передачі управління, так і *виразом* – повертати значення подібно тернарному оператору Java:

```
val a = 5
val s = if (a % 2 == 0) "a is even" else "a is odd"
```

- Замість оператора **switch** Java, у Kotlin існує оператор (який може бути виразом) **when** (аналогічно покращеному switch JDK 13+):

```
val a = 5
val grade = when (a) {
    1, 2 -> "Bad"
    3 -> "Sufficient"
    4 -> "Good"
    5 -> "Excellent"
    else -> "No such mark" }
```

під when можуть бути практично любі типи, включаючи булеві значення та числа з плаваючою комою

See ControlFlowOperators

Kotlin - оператори організації циклів

- Цикл **for-in** Kotlin працює аналогічно покращеному циклу **for** (**for-each**) Java і може виконуватись для будь-яких типів, що містять ітератор (найчастіше – колекції, масиви, діапазони чисел):

```
val colors = listOf("Red", "Green", "Blue")
for (color in colors) {
    println(color)
}
```

- Застосувавши цикл **for** для рядка, можна отримати послідовний доступ до символів цього рядка:

```
for (index in "Hello"){
    print("$index ")
}
println()
```

See ControlFlowOperators

Kotlin - оператори організації циклів

- При організації циклу **for-in** можна використовувати індекси:
- 1-спосіб - використання функції

```
public val <T> Array<out T>.indices: IntRange
```

файлу kotlin.collections._Arrays:

```
val intArr = Array(10) { (1..100).random() }
for (i in intArr.indices) {
    if (i % 2 == 0)
        print("${intArr[i]} ")
}
println()
```

Генерація масиву з 10 випадкових цілих чисел від 1 до 100 включно

See ControlFlowOperators

Kotlin - оператори організації циклів

- 2-спосіб - використання функції

```
public fun <T> Array<out T>.withIndex():
                                    Iterable<IndexedValue<T>>
```

файлу kotlin.collections._Arrays:

```
for ((i, value) in intArr.withIndex()) {
    if (i % 2 == 0)
        print("intArr[$i]=$value ")
}
println()
```

See ControlFlowOperators

Kotlin - оператори організації циклів

- У циклі for-in можна застосовувати функції **downTo**, **until** та **step**:

```
for (index in 100 downTo 90 step 2) {
    print("$index ")    //100 98 96 94 92 90
}
println()

for (index in 1 until 10 step 2) {
    print("$index ")    //1 3 5 7 9
}
println()
```

See ControlFlowOperators

Kotlin - оператори організації циклів

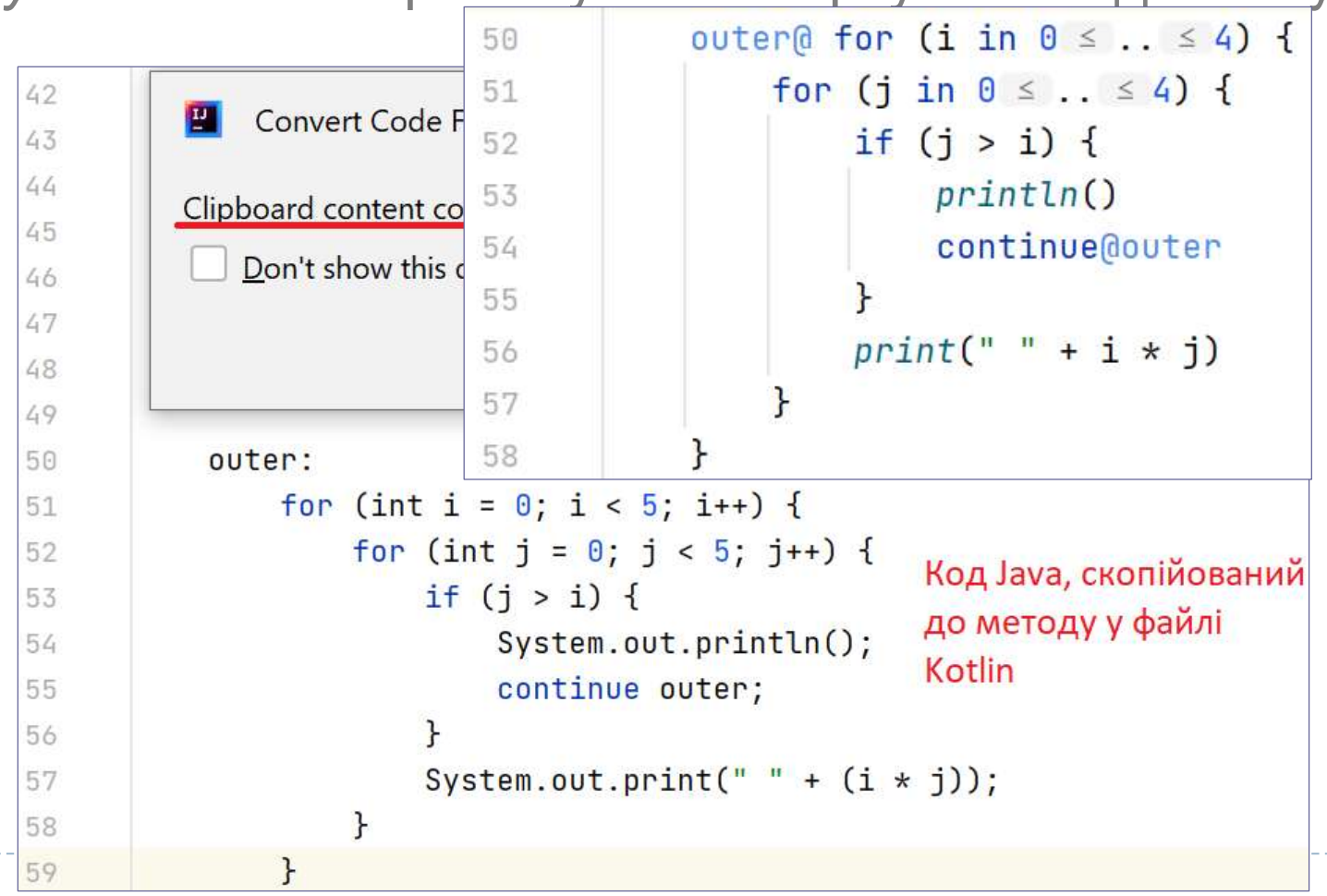
- Цикли while та do-while працюють так, як і у Java.
- Аналогічно Java працюють у циклах і оператори break та continue, включаючи використання міток. Єдиною відміною Kotlin є синтаксис міток: мітка повинна починатися з символу @, наприклад:

```
outer@ for (i in 0..4) {
    for (j in 0..4) {
        if (j > i) {
            println()
            continue@outer
        }
        print(" " + i * j)
    }
}
println() }
```

See ControlFlowOperators

Kotlin - конвертація коду Java

- При копіюванні кода Java до проєкту Kotlin через *Буфер Обміну* IntellJ IDEA пропонує сконвертувати код Java у код Kotlin.



Також наявна команда *Code–Convert Java File to Kotlin File*

Kotlin - функції та лямбда-вирази

- Повний синтаксис оголошення функцій Kotlin виглядає так:

```
fun ім'я_функції(ім'я_параметра1: тип_параметра1, ...,
                 ім'я_параметраN: тип_параметраN):
                 тип_результату {
    тіло-блок функції
}
```

- У Kotlin більшість операторів передачі управління (окрім циклів) є виразами, тобто вони повертають деяке значення. У разі, якщо тіло функції є таким виразом, оголошення функції-виразу може бути у формі:

```
fun ім'я_функції(ім'я_параметра1: тип_параметра1, ...,
                 ім'я_параметраN: тип_параметраN):
                 тип_результату = тіло-вираз функції
```

- У разі, коли компілятор зможе вивести з тіла-виразу тип_результату, він може не зазначатися. See FuncDefinition

Kotlin - функції та лямбда-вирази

- Kotlin надає можливість вказати *значення за замовчуванням для параметрів* при оголошенні функції (дозволяє уникати перевантажених методів).
- Існує можливість викликати таку функцію, передаючи їй не усі аргументи (для інших будуть братися значення за замовчуванням).
- Якщо при виклику функцій необхідно пропустити аргумент(и) (беруться їх значення за замовчуванням), то ті аргументи, які зазначаються після пропущених вказуються з ім'ям параметра – *поіменовані аргументи.*
- У разі, якщо функція Kotlin явно не повертає ніякого значення (аналог void у Java), вона все рівно повертає тип **Unit**, який має єдине значення – **Unit** (у такому випадку зазначення типу_результату є опціональним)

See FuncDefinition

Kotlin - функції та лямбда-вирази

- Функції Kotlin підтримують довільне число параметрів, для таких параметрів додаються модифікатори **varargs**).
- Також Kotlin надає оператор *spread* (*****), що дозволяє до varargs аргументів додати значення масиву (списку, діапазону) такого ж типу.
- Котлін підтримує виклики функції *у інфіксній нотації* – не зазначається оператор крапка, як доступ з об'єкта до функції та круглі дужки для параметрів, тобто виклик

```
myObj.someMeth(3)
```

у інфіксній формі буде:

```
myObj someMeth 3
```

перевагами використання функцій у інфіксній формі є можливість приблизити їх виклики до речень людської мови.

See FuncDefinition

See InfixFunClass

Kotlin - функції та лямбда-вирази

- Аналогічно Java, Kotlin підтримує функції з узагальненнями (generic), але оскільки тип, що повертається функцією оголошується в кінці, то параметри типу зазначаються у кутових дужках безпосередньо перед іменем функції:

```
package basics
/*Generic function*/
fun <T : Number> getSum(a: T, b: T): Number {
    return a.toDouble() + b.toDouble()
}

fun main() {
    /*Call generic function*/
println(getSum(3, 5))       //8.0
println(getSum(3.5, 5.5))   //9.0  }
```

See FuncDefinition

Kotlin - типи функцій

- ***функції-члени*** (***member functions***) – оголошуються всередині класів (аналогічно методам Java) та об'єктів;
- ***функції верхнього рівня*** (***top-level functions***) – функції у пакеті Kotlin, які визначаються поза будь-яким класом, об’єктом або інтерфейсом (зазвичай, у файлі Kotlin). Такі функції викликаються безпосередньо, без необхідності створювати будь-який об’єкт або зазначати будь-який клас (необхідно зазначати пакет, якщо функція оголошена у файлі з іншого пакету);
- ***локальні*** (***або вкладені***) ***функції*** (***local*** (***or nested***) ***functions***) – оголошуються всередині іншої функції, можуть мати доступ до val локальних змінних зовнішньої функції (аналогічно локальним внутрішнім класам Java);
- ***функції-розширення*** (***extension functions***) – Kotlin надає механізм розширення функціональності вже створених класів без використання успадкування або різних форм паттерну Декоратор.

Kotlin - функції верхнього рівня (top-level functions)

```kotlin
package basics

fun checkUserStatus(): String {
    return "online"
}
```

File Kotlin

```kotlin
package basics.subpkg

fun checkStatus(): String {
    return "online"
}

class TopLevelFuncClass(a: Int, b: Int) {
}
```

Class Kotlin, що став файлом Kotlin після додання функції верхнього рівня

See FuncDefinition

- Усі функції верхнього рівня у файлі Kotlin компілюються у статичні методи Java однойменного з файлом Kotlin класу.

Kotlin - функції верхнього рівня (top-level functions)

- Таку функцію можна викликати безпосередньо з іншої функції у іншому файлі у тому ж пакеті або після імпорту (чи з зазначенням повного імені) цієї функції:

```
package basics

import basics.subpkg.checkStatus

fun main() {
    /*Call top-level function from file*/
    println(checkUserStatus())

    /*Call top-level function from class*/
    println(checkStatus())
}
```

See FuncDefinition

Kotlin - локальні функції (local functions)

```
package basics

/*Outer function*/
fun printArea(width: Int, height: Int): Unit {
    /*Local (nested) function can use local variable
        of the outer function*/
    fun calculateArea() = width * height
    val area = calculateArea()
    println("The area is $area")
}
```

File/class Kotlin

See FuncDefinition

Kotlin - локальні функції (local functions)

- Локальна функція не доступна ззовні, тому вона використовується для відокремлення складеної функціональності зовнішньої функції та її інкапсуляції:

```
package basics

fun main() {
    /*Call outer function with local function definition*/
    printArea(3, 4)          //The area is 12
    /*Local (nested) function are not available
        outside the outer function*/
//    println(calculateArea())
}
```

See FuncDefinition

Kotlin - функції-розширення (extension functions)

- **Функція-розширення (extension function)** - це функція, яка може викликатись як член класу, але визначена за його межами-***механізм розширення функціональності класів без їх зміни***.
- У лівій частині оголошення такої функції зазначається **тип-отримувач (receiver type)** додаткової функціональності (у даному випадку бібліотечний клас kotlin.String), а у правій – **об'єкт-отримувач (receiver object)**, з яким власне і виконуються маніпуляції функції):

```
package basics
/*Extension function*/
fun String.lastChar(): Char = this[this.length - 1]
```

Receiver type

Object of the extension receiver type

File/class Kotlin

```
package basics

fun main() {
    /*Call extension function*/
    println("Hello".lastChar())    }
```

See FuncDefinition

Kotlin - функції-розширення (extension functions)

- Функція-розширення Kotlin компілюється у статичний метод Java однойменного класу з класом/файлом Kotlin, де перший параметр є об'єктом, на якому викликається ця функція:

```
public static char lastChar(String receiver) {
    return receiver.charAt(receiver.length() - 1);
}
```

- На відміну від методів, оголошених у класі (Receiver type)/його нащадках, функції-розширення не мають доступу до закритих/захищених членів класу.
- У разі, якщо функція-розширення оголошена у іншому пакеті, її потрібно імпортувати до класів/файлів, де вона використовується:

```
import basics.subpkg.maxChar
```

Kotlin - властивості-розширення (extension properties)

- Аналогічно функціям-розширенням до класів Kotlin назовні класів можуть додаватися **властивості-розширення (extension properties)** - це поля з гетерами та/або сетерами, що зазначаються після їх типу-отримувача:

```
val Int.bitwise: String
    get() = Integer.toBinaryString(this)
```

- Такі властивості-розширення можуть використовуватися з об'єктів типу-отримувача:

```
println(10.bitwise)     //1010
```

- Зауважимо, що в Kotlin API існують файли зі стандартними функціями-розширеннями та властивостями-розширеннями

Tools-Kotlin-Show Kotlin Bytecode Decompile

See FuncExtensions
& javap FuncExtensions.class

Kotlin - лямбда-вирази

- **Лямбда-вирази** – це функції без імені (анонімні), з якими можна поводитись як зі значеннями: передавати як параметри, повертати з інших функцій тощо. Синтаксис лямбда-функцій наступний:

  { [argumentList ->] codeBody }

  (квадратними дужками позначений необов'язковий список аргументів зі стрілкою – у разі відсутності параметрів лямбда-виразу, лямбда-вираз у Kotlin завжди вказується у фігурних дужках).
- Лямбда-вираз можна зберегти у змінній, а потім звертатися до неї як до звичайної функції.
- Змінній Kotlin може бути призначена довільна функція, аналогічно призначенню лямбда-виразу, при цьому використовується посилання на таку функцію (у поточному або іншому файлі/класі) у вигляді подвійної двокрапки (після імені об'єкта класу. якщо функція класу).

See LambdaDefinition

Kotlin - лямбда-вирази

- **Лямбда-вирази не підтримують оператор return** (фактично його замінює обов'язкова стрілка лямбда-виразу).
- Якщо лямбда-вираз не повертає результат, можна оголосити у лямбда-виразі значення, яке буде повертатися та привласнюватися змінній.
- Функції, які приймають лямбда-вирази як параметри (та/або повертають їх), називаються *функціями вищого порядку* (*higher-order functions*). Такі функції в Run-Time приймають як параметри посилання на функції, що відповідають типу функції-параметра.
- При визначенні функції вищого порядку перед типом через двокрапку вказують назву параметру, який має цей тип (аналогічно звичайним параметрам).

See LambdaDefinition

Kotlin - лямбда-вирази

- Лямбда-вирази, як і всі інші різновиди об'єктів, мають тип, в ньому вказуються типи параметрів лямбда-виразу і значення, що повертається:

```
val add: (Int, Int) -> Int        //оголошення змінної з типом лямбда
add = { x: Int, y: Int -> x + y }  //визначення змінної
println(add(3, 5))          //8 - використання змінної лямбда типу
```

- Оголошення та визначення змінної лямбда-типу можуть бути об'єднані:

```
val add: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

- Лямбда-вираз може бути спрощений уникненням оголошенням типів параметрів лямбда-виразу:

```
val add = { x: Int, y: Int -> x + y }
```

- Оголошення та визначення змінної з лямбда-типом, що нічого не приймає і нічого не повертає:

```
var printInfo: () -> Unit
printInfo = {println(42)}
printInfo()          //42
```

```
var printInfo: () -> Unit = {println(42)}
```

```
var printInfo = {println(42)}
```

See LambdaDefinition

Kotlin - inline-функції

- Лямбда-вирази є функціями вищого порядку - можуть бути передані як аргументи та повернуті як результат з інших функцій.
- Використання функцій вищого порядку накладає певні витрати: кожна функція є об'єктом анонімного класу і фіксує замикання - це область видимості зовнішніх для функції змінних, до яких можна отримати доступ у тілі функції. Виділення пам'яті (як для об'єктів функцій, так і для класів) та віртуальні виклики створюють накладні витрати під час виконання.
- Оголошення функцій, що приймають лямбда-вираз як параметр або повертають функцію, з модифікатором **inline** :
- inline fun <T> inlineMeasureTime(block: () -> T): T {

  ...

  }
- Для перегляду коду декомпільованих методів потрібно встановити у IntelliJ IDEA плагін ***Kotlin to Java decompiler***, перезавантажити IDE, відкрити файл та натиснути CTRL+ALT+SHIFT+D

See InlineFunc

Kotlin - inline-функції

- У Kotlin існує модифікатор **reified** для generic параметру типу: **<reified T>,** який може використовуватися тільки в оголошеннях inline-функцій.
- **<reified T>** – це особливий вид параметра generic типу, який зберігає інформацію про свій тип під час виконання, на відміну від стандартних універсальних типів, які зазнають стирання типу на JVM. Це дозволяє виконувати перевірки та операції під час виконання, що включають універсальний тип, такі як **is T** або **T::class.**
- Ключове слово **inline** вказує компілятору замінити виклики функцій фактичним тілом функції на місці виклику під час компіляції. Цей процес інлайнінгу дозволяє "реафікувати" інформацію про універсальний тип – по суті, конкретний аргумент типу замінює параметр універсального типу у згенерованому байт-коді.
- ```
  inline fun <reified T> printTypeName() {
      println(T::class.simpleName)
  }
  ```

See InlineFunc
  ```

Kotlin - оголошення класів та властивостей

- Kotlin оголошується аналогічно класу Java, за тим виключенням, що при оголошенні поля класу повинні бути ініціалізовані (часто ініціалізація виконується відповідними для типів полів значеннями за замовчуванням):

```
package oop

class Student {
    var name: String = ""
    var surname: String = ""
    var age: Int = 0
}

package oop

fun main() {
    val stud1 = Student()
}
```

оператор new у Kotlin не використовується

Kotlin - оголошення класів та властивостей

- Декомпіляція найпростішого класу Kotlin:

c:\KotlinStudyApp\out\production\KotlinStudyApp\oop>
javap -p Student.class

```
Compiled from "Student.kt"
public final class oop.Student {
  private java.lang.String name;
  private java.lang.String surname;
  private int age;
  public oop.Student();
  public final java.lang.String getName();
  public final void setName(java.lang.String);
  public final java.lang.String getSurname();
  public final void setSurname(java.lang.String);
  public final int getAge();
  public final void setAge(int);
}
```

incapsulated fields

default constructor

accessors
for var fields

Kotlin - оголошення класів та властивостей

- При викликах геттерів та сеттерів префіксів get та set не зазначаються – зазначаються просто поля класу, а геттер чи сеттер використовувати є зрозумілим з контексту.
- Говорять, що Kotlin надає можливість читання-зміни полів об'єкта через *властивості (properties) класу*, які являють собою приватні поля разом з публічними геттерами та сеттерами.
- Класи, які містять тільки властивості і конструктор, подібні наведеному у прикладі, називають *об'єктами-значеннями.*

See opp.Main

Kotlin - оголошення класів та властивостей

- Можна додати до класу конструктор з параметрами, але після цього конструктор без параметрів задаватися не буде, якщо вони потрібні обидва, необхідно їх оголосити::

```
class Student {
    var name: String = ""
    var surname: String = ""
    var age: Int = 0
    constructor()
    constructor(name: String, surname: String, age: Int) {
        this.name = name
        this.surname = surname
        this.age = age
    }
}
    fun main() {
        val stud2 = Student("Ron", "Weasley", 12)
        println("The stud2 data is: ${stud2.name} ${stud2.surname}
                - ${stud2.age}")    }
```

Kotlin - оголошення класів та властивостей

- Kotlin підтримує *спрощену форму оголошення класів:*

```
class SimpleStudent(val name: String, val surname: String,
                    var age: Int)
```

- Якщо перед іменем поля вказане ключове слово val, відповідне поле оголошується фінальним і його значення може встановлюватися тільки при створенні об'єкта (сеттер для такого поля буде відсутнім), а якщо перед іменем поля вказане ключове слово var, до класу додаються і геттер і сеттер цього поля.

```
fun main() {
    val stud3 = SimpleStudent("Hermione", "Granger", 13)
    println("The stud3 data is: ${stud3.name} ${stud3.surname}
            - ${stud3.age}")
}
```

Kotlin - оголошення класів та властивостей

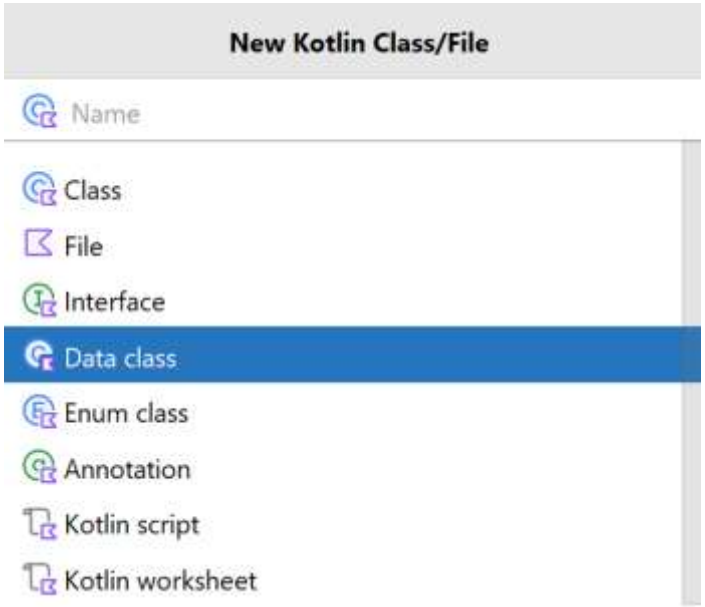
- Спрощена форма оголошення класу не дає можливості додавати додатковий окрім ініціалізації полів код, тому Kotlin (як і Java) підтримує блоки ініціалізації, які визначаються ключовим словом init та виконуються при створенні об'єктів класу:

```
class SimpleStudent(val name: String, val surname: String,
                    var age: Int) {
    init {
        println("Init block code")
    }
}
```

Kotlin - оголошення класів та властивостей

- У Kotlin автоматичне перевизначення функцій toString(), equals() та hashcode() (та інших корисних функцій) досягається доданням модифікатора data у оголошення класу :

```
data class StudentData(val name: String, val surname: String,
                       var age: Int)
```




```
val stud4 = StudentData("Neville", "Longbottom", 11)
println("The stud4 data is: $stud4")
```

Kotlin - оголошення класів та властивостей

```
data class StudentData(val name: String, val surname: String,
                       var age: Int)
```

```
Compiled from "StudentData.kt"
public final class oop.StudentData {
  private final java.lang.String name;
  private final java.lang.String surname;
  private int age;
  public oop.StudentData(java.lang.String, java.lang.String, int);
  public final java.lang.String getName();
  public final java.lang.String getSurame();
  public final int getAge();
  public final void setAge(int);
  public final java.lang.String component1();
  public final java.lang.String component2();
  public final int component3();        ...
```

Kotlin - оголошення класів та властивостей

```
data class StudentData(val name: String, val surname: String,
                       var age: Int)

…
  public final oop.StudentData copy(java.lang.String,
                                    java.lang.String, int);
  public static oop.StudentData copy$default(oop.StudentData,
                java.lang.String, java.lang.String, int, int,
                java.lang.Object);
  public java.lang.String toString();
  public int hashCode();
  public boolean equals(java.lang.Object);
}
```

Kotlin - оголошення класів та властивостей

- Окрім перевизначених функцій toString(),equals() та hashcode(), компілятор додає функцію copy(), яка дозволяє створювати копії об'єктів класу (з можливістю перевизначення їх властивостей) та функції componentN() по кількості полів класу, які являють собою альтернативний функціям-аксессорам спосіб доступу до полів об'єкта класу:

```
val stud4 = StudentData("Neville", "Longbottom", 11)
val stud5 = stud4.copy()
println("The stud5 data is: $stud5")
val stud6 = stud4.copy(name = "Dean", surname = "Thomas")
println("The stud6 data is: $stud6")
```

```
The stud5 data is: StudentData(name=Neville, surname=Longbottom, age=11)
The stud6 data is: StudentData(name=Dean, surname=Thomas, age=11)
```

Kotlin - оголошення класів та властивостей

- Створені компілятором для класу даних функції componentN() використовують для поділу об'єкта класу даних на набір значень його властивостей – такий процес називають *деструктуризацією* :

```
val s6name = stud6.component1()
val s6surname = stud6.component2()
val s6age = stud6.component3()
println("The stud6 name=$s6name, surname=$s6surname
        and age=$s6age")
```

```
The stud6 name=Dean, surname=Thomas and age=11
```

- Технологія деструктуризації об'єкта дозволяє вище наведений код замінити одним оператором: :

```
val (st6name, st6surname, st6age) = stud6
```

Kotlin - оголошення класів та властивостей

- До класу Kotlin можуть бути додані користувацькі функції, що реалізують поведінку об'єктів класу.
- Також існує можливість додання коду до стандартних гетерів та сетерів.
- У геттері та сеттері властивості використовується ідентифікатор field, який можна розглядати як посилання на значення властивості. Дуже важливо використовувати field у геттерах та сеттерах замість імені властивості, тому що так Ви запобігаєте зациклюванню та виникненню виключення StackOverflowError.
- Клас Kotlin може містити так званий *об'єкт-компаньйон* (*companion object*), який зберігає методи та змінні, що є загальними для усіх об'єктів класу (аналог static-членів класу Java).

See StudentData See Student See SimpleStudent

See opp.Main

Kotlin - успадкування класів та поліморфізм

- Усі класи у Kotlin неявно успадковуються від класу **kotlin.Any**, який містить три функції: equals(), hashCode() та toString() (аналогічно класу Object в Java).
- За замовчуванням класи Kotlin є final – їх не можна успадкувати. Щоб зробити клас успадковуваним, необхідно додати до його оголошення модифікатор **open** :

```
open class Person(var name: String,
                  var surname: String, var age: Int) { }
```

при цьому на рівні байт коду в оголошенні класу відсутній модифікатор final

- Явне успадкування класів у Kotlin позначається двокрапкою.
- При оголошенні класу-нащадка у його первинному конструкторі повинні бути зазначені як параметри властивості батьківського класу та оголошені свої. Первинний конструктор батьківського класу приймає параметри для своїх властивостей з первинного конструктору класу-нащадка.

```
class Student(name: String, surname: String, age: Int,
              var speciality: String) : Person(name, surname, age) {
```

Kotlin - успадкування класів та поліморфізм

- У разі, якщо батьківський клас оголошений із використанням вторинного конструктора, оголошення класу-нащадку виглядає простішим: параметри вторинного конструктора класу-нащадка передаються до конструктору батьківського класу за допомогою ключового слова super, як у Java:

```
open class SimplePerson {
    var name: String = ""
    var surname: String = ""
    var age: Int = 0


    constructor(name: String, surname: String, age: Int) {
        this.name = name
        this.surname = surname
        this.age = age
    }
}
```

see next slide

Kotlin - успадкування класів та поліморфізм

- У разі, якщо батьківський клас оголошений із використанням вторинного конструктора, оголошення класу-нащадку виглядає простішим: параметри вторинного конструктора класу-нащадка передаються до конструктору батьківського класу за допомогою ключового слова super, як у Java:

```
class SimpleStudent : SimplePerson {
    var speciality: String = ""

    constructor(name: String, surname: String, age: Int,
                speciality: String)
        : super(name, surname, age) {
        this.speciality = speciality
    }
}
```

Kotlin - успадкування класів та поліморфізм

- Kotlin вимагає явних модифікаторів **open** у оголошенні методу батьківського класу, який дозволяється перевизначати у класі-нащадку (модифікатор **open** методу не має ефекту, коли додається до членів **final** класу – класу без модифікатора **open**):

```
open class Shape(val name: String) {
    open fun calcArea(): Double {
        return 0.0
    }
}
```

властивість (навіть val) може перевизначатися, коли вона повинна ініціалізуватися іншим, аніж у суперкласі, значенням

- Kotlin вимагає явних модифікаторів **override** у оголошенні методу класу-нащадку, що перевизначає батьківський метод:

```
class Circle(val radius: Double) : Shape() {
    final override val name: String = "circle"
    override fun calcArea(): Double {
        return Math.PI * radius * radius
    }    }
```

повторне перевизначення забороняється модифікатором final

Kotlin - абстрактні класи та інтерфейси

- Kotlin підтримує роботу з абстрактними класами та інтерфейсами аналогічно Java:

```
abstract class AbsShape {
    abstract var name: String
    abstract fun calcArea(): Double
}

class Circle(var radius: Double) : AbsShape() {
    override var name: String = "circle"
    override fun calcArea(): Double {
        return Math.PI * radius * radius
    }
}
```

абстрактний клас може містити абстрактні властивості та методи

- Абстрактні класи автоматично є відкритими (open), а абстрактні методи та властивості автоматично є відкритими для перевизначення.

see oop.inheritance.Main

Kotlin - абстрактні класи та інтерфейси

- Інтерфейси використовуються для визначення протоколу загальної поведінки, щоб переваги поліморфізму були доступні без жорсткої структури успадкування.
- Клас може реалізувати декілька інтерфейсів, але успадковується лише від одного суперкласу (як у Java).

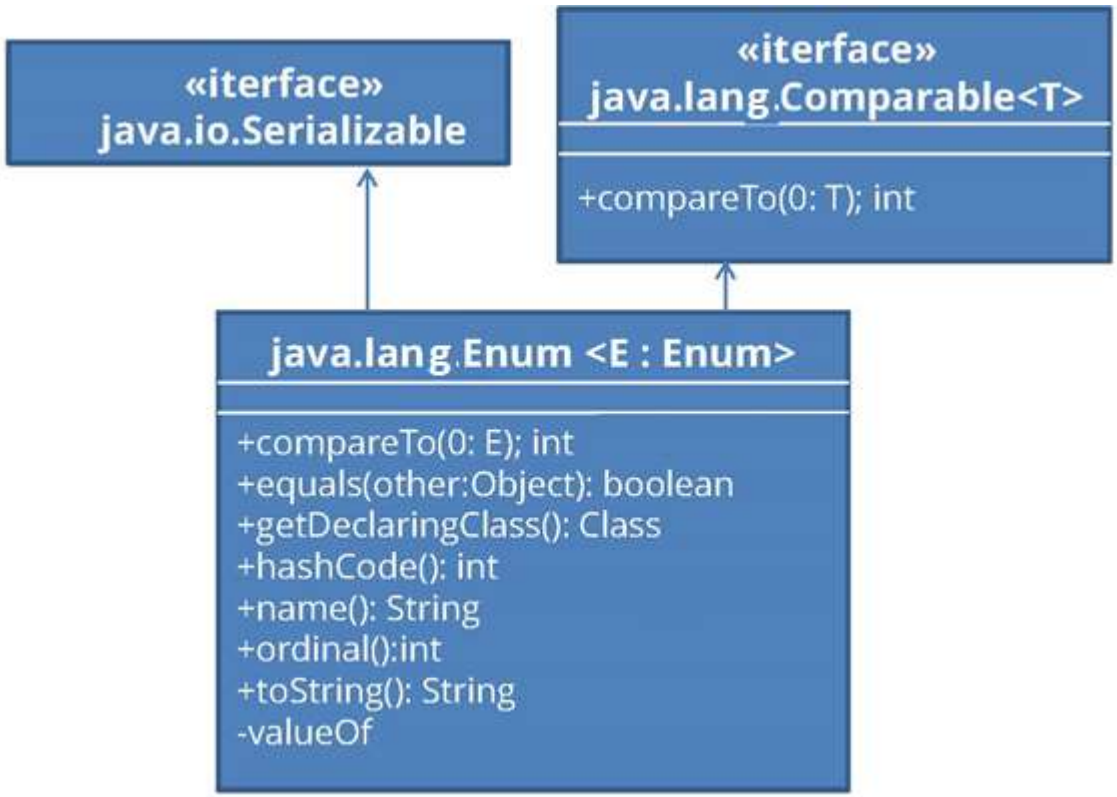
```
abstract class AbsShape {
    abstract var name: String
    abstract fun calcArea(): Double
}

class Circle(var radius: Double) : AbsShape() {
    override var name: String = "circle"
    override fun calcArea(): Double {
        return Math.PI * radius * radius
    }
}
```

абстрактний клас може містити абстрактні властивості та методи

Kotlin - енумератори

- Енумератори Kotlin (як і у Java) є final класами - нащадками класу java.lang.Enum<E:Enum>



See javap -p MyDirection.class

Kotlin - енумератори

- Енумератори Kotlin (як і у Java) є final класами - нащадками класу java.lang.Enum<E:Enum>

Кожен енумератор має **додані компілятором методи**:

- **public static** EnumName[] **values()** - returns an array EnumName[] $VALUES of all enum constants
- **public static** EnumName **valueOf(String name)** - parse enum constant

Кожен енумератор має **методи, успадковані від**

public abstract class Enum<E **extends** Enum<E>>
implements Constable, Comparable<E>, Serializable

public final String **name()** - returns the name of this enum constant

- **public final int ordinal()** - returns an ordinal value of this enum constant
- **public static** <T **extends** Enum<T>> T valueOf(Class<T> enumClass, String name) - Returns the enum constant of the specified enum class with the specified enum constant name

Kotlin - енумератори

Оскільки enum є класом, ви можете:

- додати до нього спеціальні поля;
- додати до нього перевантажені конструктори;
- додати до нього спеціальні методи;
- перекривати стандартні методи.

See Shape

See Fruit

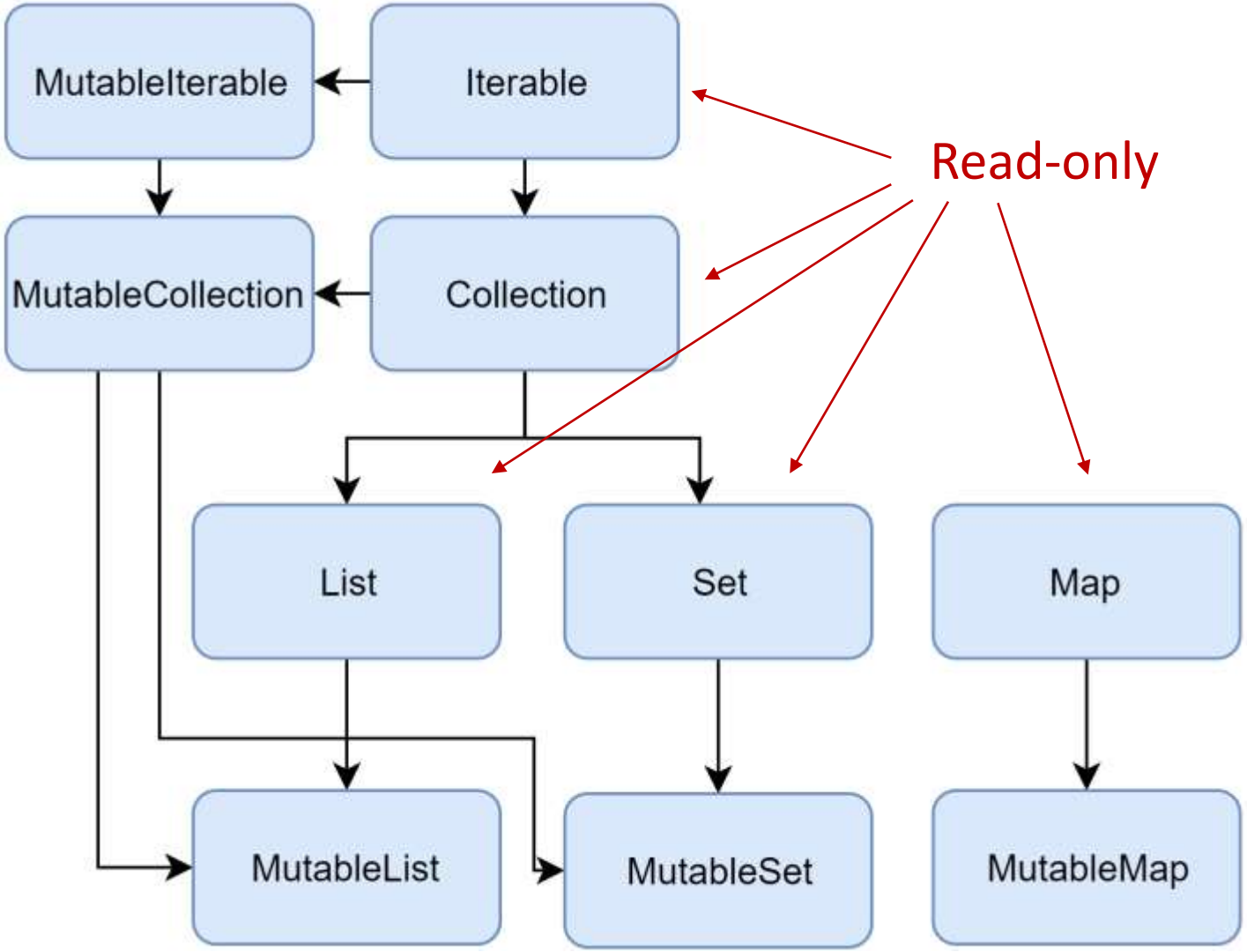
See DocumentStatus

Kotlin - колекції

- Стандартна бібліотека Kotlin надає реалізації для основних типів колекцій: списків (lists), множин (sets), та карт (maps).
- Кожен тип колекції представляє пара інтерфейсів :
  - Інтерфейс лише для читання, який надає операції для доступу до елементів колекції.
  - Змінний інтерфейс (mutable), який розширює відповідний інтерфейс лише для читання операціями запису: додавання, оновлення та видалення її елементів.
- Kotlin підтримує перевантажені оператори +, -, += та -= для додання та видалення елементів з mutable колекції.
- Більшість функцій для роботи з колекціями зберігаються у файлі Kotlin kotlin.collections._Collections.kt стандартної бібліотеки kotlin-stdlib, яка підключається з файлу .m2\repository\org\jetbrains\kotlin\kotlin-stdlib\версія\kotlin-stdlib-версія.jar.

See CollectionCreating

Kotlin - діаграма інтерфейсів колекцій



Kotlin - створення колекцій

- Стандартна бібліотека Kotlin надає функції для створення колекцій з елементів:

```
fun <T> listOf(vararg elements: T): List<T>
fun <T> setOf(vararg elements: T): Set<T>
fun <K, V> mapOf(vararg pairs: Pair<K, V>): Map<K, V>
fun <T> mutableListOf(vararg elements: T): MutableList<T>
fun <T> mutableSetOf(vararg elements: T): MutableSet<T>
fun <K, V> mutableMapOf(vararg pairs: Pair<K, V>): MutableMap<K, V>
```

- Колекцію можна створити за допомогою функцій-білдерів: **buildList**(), **buildSet**(), or **buildMap**(). Вони створюють нову, змінну колекцію відповідного типу, заповнюють її за допомогою операцій запису та повертають колекцію лише для читання з тими ж елементами.
- Для списків існують конструктори List та MutableList, які приймають розмір списку, та функцію-ініціалізатор, яка визначає значення елемента на основі його індексу. See CollectionCreating

Kotlin - отримання елементів колекцій

З колекції можна отримати один елемент функціями:

- Елемент колекції за індексом отримується функцію *elementAt(index)* або *ім'я_колекції*[*індекс*] або генерується *IndexOutOfBoundsException* при неіснуючому індексі. *getOrNull(index)* повертає null заміть виключення. *getOrElse(index, default value)* повертає 2 аргумент замість виключення.
- *first* - повертає перший елемент колекції. абогенерується NoSuchElementException для пустої колекції. У разі, якщо приймає предикат-аргумент - повертає перший елемент колекції, що відповідає предикату, або NoSuchElementException. *firstOrNull* повертає null заміть виключення.
- *last* - повертає останній елемент колекції. або NoSuchElementException для пустої колекції. У разі, якщо приймає предикат-аргумент - повертає останній елемент колекції, що відповідає предикату, або NoSuchElementException. *lastOrNull* повертає null заміть виключення.

See CollectionRetrieving

Kotlin - отримання елементів колекцій

З колекції можна отримати її частину функціями:

- *take* - повертає задану аргументом кількість елементів, починаючи з першого. *takeLast* - повертає задану аргументом кількість останніх елементів колекції. *takeWhile* та *takeLastWhile* - приймають предикат - перебирають елементи з початку та з кінця колекції та повертають тільки ті, що відповідають предикату. При першому, що не відповідає, завершують роботу.
- *drop* - повертає всі елементи, окрім заданої аргументом кількості починаючи з першого елемента. *dropLast* - повертає задану аргументом кількість останніх елементів колекції. *dropWhile* та *dropLastWhile* - приймають предикат - перебирають елементи з початку та з кінця колекції та повертають тільки ті, що НЕ відповідають предикату. При першому, що відповідає, завершують роботу.

See CollectionRetrieving

Kotlin - отримання елементів колекцій

- *chunked* - розбиває колекцію на частини заданого розміру-аргумента та повертає список списків заданого розміру. Останній список-фрагмент може бути меншого аргументу розміру. Може приймати другий лямбда-аргумент, що перетворює вихідні значення елементів.
- *windowed* - повертає задану аргументом кількість елементів, починаючи з першого. *takeLast* - повертає задану аргументом кількість останніх елементів колекції. *takeWhile* та *takeLastWhile* - приймають предикат - перебирають елементи з початку та з кінця колекції та повертають тільки ті, що відповідають предикату.
- *partition* - розбиває оригінальну колекцію на пару списків, де перший список містить елементи, для яких предикат-аргумент повернув значення true, а другий список містить елементи, для яких предикат повернув значення false.

See CollectionRetrieving

Kotlin - отримання елементів колекцій

- *slice* - повертає новий список з елементами колекції із заданими індексами. Індекси можуть бути передані або як діапазон, або як колекція цілочисельних значень.
- *sublist* - представлення елементів колекції із заданими індексами. Ідекси зазначаються аргументами from (inclusive) та to (exclusive).

See CollectionRetrieving

Kotlin - фільтрація та трансформація колекцій

- З колекцій (файл kotlin.collections._Collections.kt) можна викликати функцію
  `inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T>`
  яка повертає нову колекцію тільки з елементами, що задовольняють аргументу-предикату.
- Також наявні функції *filterNot,* що використовує інверсію предикату-аргументу та *filterIndexed,* що використовує предикат, побудований на основі індексів елементів.
- З колекцій можна викликати функцію
  `inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R>`
  яка повертає нову колекцію з елементами, значення яких визначається аргументом-функцією transform з вихідних значень елементів. Для Map викликається *mapValues.*
- Функція *filterNotNull*() повертає колекцію з видаленими елементами null з оригінальної колекції.

See CollectionFilteringTransforming

Kotlin - угрупування елементів колекцій

- Елементи колекцій можна групувати. Функція *groupBy* приймає лямбда-функцію та повертає об'єкт типу Map. У цьому об'єкті кожен ключ є результатом лямбда, а відповідне значення — списком елементів, для яких повертається цей результат.
- Ви також можете викликати *groupBy* з другим лямбда-аргументом – функцією перетворення значень. У результуючій карті ключі, створені першим лямбда-аргументом, зіставляються з результатами перетворення вихідних значень другим лямбда-аргументом.
- Якщо необхідно згрупувати елементи, а потім застосувати операцію до всіх груп одночасно, використовуйте функцію *groupingBy*. Вона повертає екземпляр типу *Grouping*.

See CollectionElementsGrouping

Kotlin - угрупування елементів колекцій

Grouping підтримує такі основні операції:

- *eachCount* підраховує елементи в кожній групі.
- *fold* - дозволяє виконувати користувацьке накопичення (accumulation) для згрупованих у кожній групі елементів з використанням функції-аргумента для зміни їх значень.
- *reduce* - дозволяє виконувати користувацьку редукцію для згрупованих у кожній групі елементів з використанням функції-аргумента для зміни їх значень.
- *aggregate* - застосовує задану операцію послідовно до всіх елементів у кожній групі та повертає результат. Це загальний спосіб виконання будь-яких операцій над *Grouping*. Використовуйте його для реалізації користувацьких операцій, коли *fold* або *reduce* недостатньо.

See CollectionElementsGrouping

Kotlin - сортування колекцій

- Елементи колекцій можна упорядкувати, визначаючи їх з використанням інтерфейсу Comparable<T> - забезпечує так званий природний (natural) порядок сортування та інтерфейсу Comparator<T> - забезпечує користувацький (custom) порядок сортування. Але зручніше використовувати функції:
- *sorted* - повертає список з відсортованими у натуральному порядку елементами колекції;
- *sortedDescending* - повертає список з відсортованими у зворотному натуральному порядку елементами колекції;
- *sortedBy* - повертає список з відсортованими елементами колекції у натуральному порядку для результата функції-аргумента;
- *sortedByDescending* - повертає список з відсортованими елементами колекції у зворотному натуральному порядку для результата функції-аргумента;

See CollectionSorting

Kotlin - сортування колекцій

- *sortedWith*(*compareBy*) - виконує сортування у прямому порядку відповідно до компаратора, визначеного аргументом *compareBy**;
- *sortedWith*(*compareByDescending*) - виконує сортування у зворотному порядку відповідно до компаратора, визначеного аргументом *compareByDescending**;
- *sortedWith*(*compareBy*<T>().*thenBy* - виконує багаторівневе сортування у прямому порядку відповідно до компаратора, визначеного аргументом *compareBy,* а потім компаратора, визначеного аргументом *thenBy;*
- *sortedWith*(*compareByDescending* <T>(). *thenByDescending* - виконує багаторівневе сортування у зворотному порядку відповідно до компаратора, визначеного аргументом *compareByDescending,* а потім компаратора, визначеного аргументом *thenByDescending;*

See CollectionSorting

- * можна використовувати nudlfirst() та nulllast() компаратори

Kotlin - сортування колекцій

- *sortedBy*.*reversed*() - змінює порядок сортування на зворотний для відсортованих функцією *sortedBy* елементів;
- *sortedBy*.*asReversed*() - повертає список-представлення з відсортованими елементами відповідно до аргументу *sortedBy* але у зворотному порядку. При зміні оригінальної колекції відповідні зміни будуть і у представленні.;
- *shuffled*() - перемішування елементів колекції у випадковому порядку.

See CollectionSorting

Kotlin - агрегатні операції над колекціями

- Колекції Kotlin містять функції для часто використовуваних агрегатних операцій – операцій, які повертають одне значення на основі вмісту колекції:
- *sum*() - повертає суму елементів у колекції чисел.
- *sumOf* - приймає функцію-селектор і повертає суму її застосування до всіх елементів колекції
- *average()* - повертає середнє значення елементів у колекції чисел.
- *count()* - повертає кількість елементів у колекції.

See CollectionAggregating

Kotlin - агрегатні операції над колекціями

- *maxOrNull()* та *minOrNull()* - повертають відповідно найбільший та найменший елементи.
- *maxByOrNull()* та *minByOrNull()* приймають функцію селектора та повертають елемент, для якого вона повертає найбільше або найменше значення.
- *maxWithOrNull()* та *minWithOrNull()* приймають об'єкт Comparator та повертають найбільший або найменший елемент відповідно до цього Comparator.
- *maxOfOrNull()* та *minOfOrNull()* приймають функцію селектора та повертають найбільше або найменше значення, що повертається самим селектором.

See CollectionAggregating

- *maxOfWithOrNull()* та *minOfWithOrNull()* приймають об'єкт Comparator та повертають найбільше або найменше значення, що повертається селектором, відповідно до цього Comparator.

УСІ ФУНКЦІЇ ПОВЕРТАЮТЬ null НА ПОРОЖНІХ КОЛЕКЦІЯХ.

Kotlin - агрегатні операції над колекціями

- Також існують альтернативи – *maxOf*, *minOf*, *maxOfWith* та *minOfWith* – які роблять те саме, що й їхні аналоги, але викидають виняток NoSuchElementException для порожніх колекцій.
- *reduce* - агрегує елементи колекції в одне значення. Приймає два аргументи: попередньо накопичене значення та елемент колекції. Не приймає явного початкового значення. Використовує перший елемент колекції як початковий акумулятор. Викидає виняток UnsupportedOperationException, якщо викликається для порожньої колекції, оскільки немає першого елемента для використання як початкового акумулятора.
- *fold* - агрегує елементи колекції в одне значення. Приймає два аргументи: попередньо накопичене значення та елемент колекції. Потрібне явне початкове значення, яке служить відправною точкою для процесу накопичення. Безпечно обробляє порожні колекції, повертаючи надане початкове значення, коли колекція порожня.

See CollectionAggregating

Kotlin - операції над колекціями

- Операції, що змінюють колекції:
- *add* та *addAll* або *+=*.
- *remove* та *removeAll* або *-=*
- *retainAll* - видаляє всі елементи, крім тих, що знаходяться в колекції аргументів. При використанні з предикатом залишаються лише ті елементи, які йому відповідають.
- *distinct* - видаляє дублікати, повертає список без дубліктів.
- *clear()*

See CollectionChangeOperations

Kotlin - специфічні операції для списків

- *get* або *[index]*, також *getOrElse* та *getOrNull*.
- *sublist(index_start, index_end)*
- *indexOf* та *lastIndexOf*, також *indexOfFirst(predicate)* та *indexOfLast(predicate)* - останні дві повертають перший/останній елемент, що відповідає предикату
- *add(index, value)* та *addAll(index, another list)* - вставляє елемент/список у список за вказаним індексом
- *collection_name[index]=value* - оновлення елементу за індексом
- *fill(value)* - заповнює всі елементи mutable списку значенням-аргументом
- *removeAt(index)* - видаляє елемент поточного списку за вказаним індексом-аргументом
- *sort, sortDescending, sortBy, sortByDescending, reverse, shuffle* (замість *sorted* - для колекцій) - виконують сортування елементів mutable списку.

See ListSpecificOperations

Kotlin - специфічні операції для множин

- *union - повертає множину, що містить усі різні елементи з поточної колекції та колекції-аргуента.*
- *intersect* - повертає множину, яка містить усі елементи, що містяться як у поточній колекції, так і у колекції-аргументі.
- *subtract* - повертає множину, яка містить усі елементи, що містяться у поточній колекції та не містяться у колекції-аргументі.

See SetSpecificOperations

Kotlin - специфічні операції для карт

- *get* або *[key]*, *getOrElse*, *getOrDefault*.
- *keys*, *values*
- *put(key,value)* або *map_name[key]=value*
- *filter*, *filterKeys*, *filterValues*
- *+-*, *-= with Pair* або *Map value*
- *set(key, value)* або *map_name[key]=value*
- *associate(transform: (T) -> Pair<K, V>): Map<K, V>* - повертає карту (Map), що містить пари ключ-значення, надані функцією перетворення, застосованою до елементів заданої колекції. Якщо будь-яка з двох пар має однаковий ключ, остання з них додається до карти. Повернена карта зберігає порядок ітерацій записів оригінальної колекції.
- *assosiateBy(keySelector)*, *associateWith(valueSelector)*

See MapSpecificOperations

Kotlin - копіювання колекцій

- Щоб створити колекцію з тими ж елементами, що й існуюча колекція, можна використовувати **функції копіювання** такі, як toList(), toMutableList(), toSet() та інш, що створюють **поверхневі** (**shallow**) копії колекцій з посиланнями на ті ж елементи. Таким чином, зміна, внесена до елемента колекції, відображається на всіх її копіях.
- Функції копіювання колекцій створюють знімок колекції в певний момент. Їх результатом є нова колекція з тих самих елементів. Якщо ви додаєте або видаляєте елементи з оригінальної колекції, це не вплине на копії. Копії також можна змінювати незалежно від джерела.
-

See CollectionCopying

Kotlin - послідовності (sequences)

- Колекції та послідовності Kotlin пропонують способи роботи з даними, але вони принципово відрізняються тим, коли виконується кожне перетворення колекції або послідовності:
- Для колекцій таке перетворення (так зване нетерпляче eager перетворення) виконується зразу ж над кожним елементом колекції, а результат операції зберігається в новій колекції, яка передається наступному у ланцюжку оператору. Використовується для оброки невеликих наборів даних та нескладних перетворень.
- Для послідовностей кожен елемент обчислюється ліниво (lazy) - ланцюжком проміжних операцій (до термінальної). При цьому для кожної операції у ланцюжку не створюється проміжної колекції. Використовується для обробки великих або нескінчених наборів даних та складних перетворень.

See SequenseVsCollection

Kotlin - перевантаження операторів

- Оператори Java прив'язані до об'єктів певних класів. Наприклад, об'єкти String та чисельних класів-огорток можуть використовувати оператор + для конкатенації та додавання, відповідно.
- Об'єкти інших класів Java не можуть перевизначати такий оператор для власних потреб, наприклад, складання значень властивостей двох об'єктів.
- Kotlin підтримує узгодження між обмеженим переліком операторів та іменами функцій, перевантаження яких забезпечує перевантаження операторів для об'єктів довільних класів - так зване обмежене перевантаження операторів.

Kotlin - перевантаження операторів

- Обмежене перевантаження операторів та відповідні функції:

Оператор	Функція
a + b	plus
a - b	minus
a * b	times
a / b	div
a % b	mod
+a	unaryPlus
-a	unaryMinus
!a	not
++a, a++	inc
--a, a--	dec

*Також можуть перевантажуватися складені оператори привласнення та оператори порівняння.

See oop.operatoroverloading

Kotlin - функції області видимості

- Функції області видимості (scope functions) - це допоміжні функції бібліотеки Kotlin, що дозволяють будувати більш зрозумілі та короткі конструкції коду. Існують такі функції області видимості: apply, let, run, with, also, takeif.
- Кожна функція області видимості викликається з об'єкта-отримувача (receiver) лямбда-виразу, який визначає дії, що виконуються з цим об'єктом-отримувачем.
- Кожна функція приймає аргументом лямбда-вираз, розміщений поза дужками аргументів функції - заключний лямбда-вираз (trailing lambda).
- Зазначені функції фактично обмежують область видимості кожного свого виклику в середині лямбда-функції об'єктом-отримувачем, для якого викликана сама функція - обмеження відносної області видимості (relative scoping).

Kotlin - функції області видимості - apply

- Функція **apply** викликається з об'єкта-отримувача і виконує визначений лямбда-виразом блок коду з об'єктом-отримувачем та повертає результуючий об'єкт-отримувач.
- Усередині блоку об'єкт-отримувач зазначається неявним параметром **this**, яке може бути опущене.
- Ця функція використовується для ініціалізації та модифікації декількох властивостей об'єктів.

See scopefunc.ScopeFunctions

Kotlin - функції області видимості - let

- Функція **let** виконує визначений лямбда-виразом блок коду з об'єктом-отримувачем та повертає значення останнього виразу у блоці.
- Усередині блоку об'єкт-отримувач зазначається неявним параметром **it**.
- Ця функція часто використовується для null-безпечних операцій з об'єктом-отримувачем, перетворення об'єкта-отримувача та поєднання декількох таких операцій у ланцюжок, коли результат попередньої операції **let** стає об'єктом-отримувачем наступної.

See scopefunc.ScopeFunctions

Kotlin - функції області видимості - run

- Функція **run** виконує визначений лямбда-виразом блок коду з об'єктом-отримувачем та повертає результат останнього виразу в блоці.
- Усередині блоку об'єкт-отримувач зазначається неявним параметром **this**, яке може бути опущене.
- Цю функцію можна викликати з об'єкта (як функцію розширення) або використовувати як функцію, що не є функцією розширення, для виконання блоку коду без певного отримувача.

See scopefunc.ScopeFunctions

Kotlin - функції області видимості - with

- Функція **with** - це функція без розширення, яка має об'єкт-отримувач та визначений лямбда-виразом блок коду як аргументи (останній зазначається як заключний лямбда-вираз).
- Усередині лямбда-виразу можливо звертатися до членів об'єкта-отримувача без використання його імені (фактично використовується **this**).
- Функція **with** повертає результат останнього виразу в блоці.
- Ця функція використовується Коли потрібно виконати кілька операцій над одним об'єктом без необхідності ланцюжкових викликів, особливо коли об'єкт не може мати значення null, та для групування операцій над об'єктом для кращої читабельності.

See scopefunc.ScopeFunctions

Kotlin - функції області видимості - also

- Функція **also** використовується для виконання додаткових дій або побічних ефектів над об'єктом-отримувачем без зміни самого об'єкта.
- Усередині лямбда-виразу можливо звертатися до членів об'єкта-отримувача без використання його імені (фактично використовується **this**).
- Функція **also** повертає оригінальний об'єкт-отримувач.
- Ця функція використовується для ланцюгових операцій або для виконання таких дій, як ведення журналу чи налагодження протягом життєвого циклу об'єкта.

See scopefunc.ScopeFunctions

Kotlin - функції області видимості - takeIf

- Функція **takeIf** повертає результуючий об'єкт-отримувач, з якого вона викликається, якщо наданий за допомогою лямбда-виразу аргумент-предикат має значення true; інакше вона повертає null.
- Усередині лямбда-виразу об'єкт-отримувач зазначається неявним параметром **it**.
- Ця функція використовується як стислий спосіб умовної обробки об'єкта або ланцюгових операцій на основі певної умови.

See scopefunc.ScopeFunctions

Kotlin - функції області видимості

Функ-ція	Аргументи	Повертає	Посилання на об'єкт-отримувач	Використовується
apply	заключний лямбда-вираз	результуючий об'єкт-отримувач	this (опускається)	ініціалізація/модифікація декількох властивостей об'єкта-отримувача
let	заключний лямбда-вираз	результат останнього виразу у лямбді	it	null-safe операції з об'єктом отримувачем та його перетворення
run	заключний лямбда-вираз	результат останнього виразу у лямбді	this (опускається)	розширення об'єкта-отримувача або звичайна функція
with	об'єкт-отримувач, заключний лямбда-вираз	результат останнього виразу у лямбді	this (опускається)	угрупування кількох операцій на об'єкті отримувачі

See scopefunc.ScopeFunctions

Kotlin - функції області видимості

Функ-ція	Аргументи	Повертає	Посилання на об'єкт-отримувач	Використовується
also	заключний лямбда-вираз	оригінвльчий об'єкт-отримувач	it	додаткові дії або побічні ефекти над об'єктом-отримувачем без зміни самого об'єкта
takeIf	предикат	результуючий об'єкт-отримувач або null	it	умовна обробка об'єкта або ланцюгові операції на основі певної умови

See scopefunc.ScopeFunctions

Kotlin - корутини (coroutines)

- Програмам часто потрібно виконувати кілька завдань одночасно: обробка введення користувача, завантаження даних або оновлення екрана. Найпоширеніший спосіб одночасного виконання завдань – це використання потоків, однак потоки є відносно важкими, і створення багатьох з них може призвести до проблем із продуктивністю.
- Kotlin використовує асинхронне програмування, побудоване на основі **корутин**, які дозволяють писати асинхронний код у природному, послідовному стилі, використовуючи функції призупинення (suspend). Корутини – це легкі альтернативи потокам. Вони можуть призупинятися, не блокуючи системні ресурси, і є ресурсоємними, що робить їх кращими для дрібнозернистого паралелізму (приклад - запити до Веб-сервера).
- Більшість функцій корутин надаються бібліотекою kotlinx.coroutines, яка включає інструменти для запуску корутин, обробки паралельності, роботи з асинхронними потоками тощо.