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/

Вступ - крос-платформні засоби програмування

JS

ReactNative

Xamarin

C#

Flutter

Dart

.apk

.ipa

Android

iOS

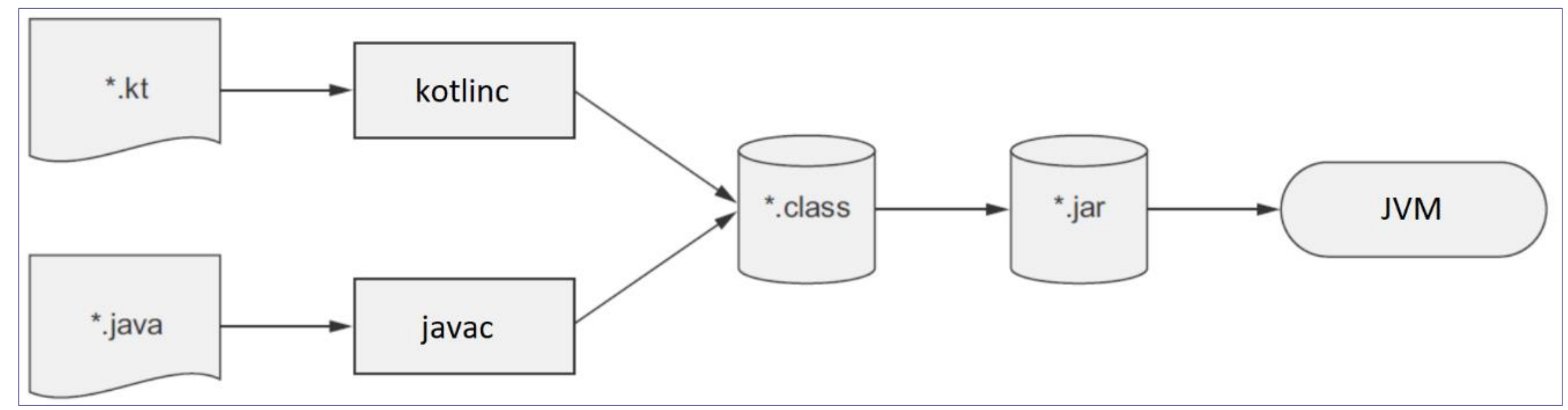
Kotlin - загальна характеристика

- 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
- Kotlin, як і Java, є *статично типізованою* та компільованою мовою програмування

Kotlin - переваги

- лаконічний та інтуїтивно зрозумілий синтаксис;
- підтримка безпечної роботи з посиланнями зі значенням null;
- автоматичне приведення типів при їх перевірці;
- малі накладні видатки при виконанні застосунків;
- підтримка, разом з об'єктно-орієнтованою, функціональної парадігми 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 - 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(args: Array<String>) {
    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
Boolean	boolean	-	true or false (for Kotlin can not be null)
Boolean?	-	-	true or false (can be null)
Char	char	8-32	Unicode characters
String	-	-	Послідовність символів
Array<T>	-	-	Клас, об'єкти якого представляють масиви

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
  ```
  RECOMMENDED
- 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 - 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 = *intArrayOf*(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*** : RECOMMENDED

```
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
```

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 і повинні використовуватися як функції, що можуть викликатися з базових числових об'єктів, 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 VarDefinition 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* IntelIJ IDEA пропонує сконвертувати код Java у код Kotlin.

Convert Code F

Clipboard content co

Don't show this d

```
50      outer:
51          for (int i = 0; i < 5; i++) {
52              for (int j = 0; j < 5; j++) {
53                  if (j > i) {
54                      System.out.println();
55                      continue outer;
56                  }
57                  System.out.print(" " + (i * j));
58              }
59          }
```

```
50      outer@ for (i in 0 ≤ .. ≤ 4) {
51          for (j in 0 ≤ .. ≤ 4) {
52              if (j > i) {
53                  println()
54                  continue@outer
55              }
56              print(" " + i * j)
57          }
58      }
```

Код Java, скопійований до методу у файлі 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.
- 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, вона все рівно неявно повертає тип **Unit**, який має єдине значення – **Unit** (у такому випадку зазначення типу_результату є опціональним)

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/класу 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

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:

```kotlin
package basics

import basics.subpkg.checkStatus

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

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

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

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())
```

- }

Kotlin - функції-розширення (extension functions)

- У лівій частині оголошення такої функції зазначається тип-отримувач розширення (у даному випадку бібліотечний клас kotlin.String, а у правій – об'єкт-отримувач, з яким власне і виконуються маніпуляції функції):

```
package basics

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

File/class Kotlin

- ```
  package basics

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

Робота з функціями-членами класів та об'єктів буде розглянута далі.
  ```

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 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 вигляді подвійної двокрапки.

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 - 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);
}
```

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

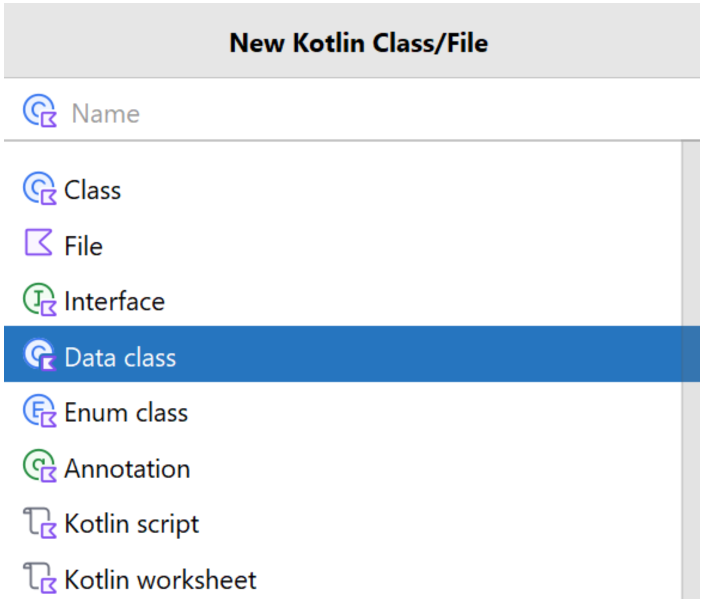
- Спрощена форма оголошення класу не дає можливості додавати додатковий окрім ініціалізації полів код, тому Kotlin (як і Java) підтримує блоки ініціалізації, які визначаються ключовим словом init та виконуються при створенні об'єктів класу:

```
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

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) {  }
```

- 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** 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 не є фінальними, 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