**Міністерство освіти і науки України**

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Факультет інформатики та обчислювальної техніки**

**Кафедра обчислювальної техніки**

**Розрахунково-графічна робота**

з дисципліни

«Об’єктно-орієнтоване програмування»

Виконав: Перевірив:

студент групи ІП-93 Порєв Віктор Миколайович

Домінський Валентин Олексійович

Номер залікової книжки: 9311

Номер у списку: 9

Київ 2020

**Вступ:**

За списком Моєю темою став звичайний калькулятор. Спочатку Я хотів писати на звичному для Мене C#, але в кінці передумав, з кількох причин:

1. Постійно писати на C# нудно
2. Викладач заохочує писати на інших мовах, таких як Java, Kotlin та програми під Android у цілому

Отже мовою програмування став Kotlin, оскільки ця мова молода та доволі перспективна, цільовою операційною системою став Android, а середовищем для створення – Android Studio.

**Початок роботи:**

При роботі над новим проектом Я завжди використовую схему «зроби хоч якось, аби працювало». У даного методу є багато противників і хочу зізнатися, що Я теж один з них, але, все-таки, особисто Мені так легше працювати. Ось деякі плюси цього способу:

1. Доволі швидко можна отримати готовий продукт або фінальний результат.

2. При рефакторингу у Вас завжди буде працюючий код, за допомогою якого Ви зможете перевіряти правильність Своїх дій для створення нового, більш ефективного коду.

Першою новинкою, з якою ще треба було ознайомитися, стала XML-розмітка. Я ніколи не працював з даною технологогією та перша година роботи була непростою, але потім Я звик до неї і вона стала виглядати доволі зручною. Також в Andoid studio є гарна функція показу коду розмітки та її вигляду для користувачів, що стало в нагоді ще не раз.

**Перша реалізація:**

Як Я вже казав, користуючись методом «зроби хоч якось, аби працювало», можна доволі швидко отримати результат, що в Моєму випадку і сталося. Приблизна реалізація виглядала якось так:

1. Кожна кнопка мала на собі Listener, який викликав функцію А.1
2. Функція А.1 змінювала теперішню операцію (наприклад на «+») та передавала в іншу функцію Б, яка була спільна для багатьох функцій А
3. У функції Б, у залежності від операції та тексту, який вводив користувач, виконувалися певні операції (наприклад операція, яка працювала з двома числами або ж інша, та, що мала діло лише з одним числом), в яких уже відбувалися математичні операції, а результат передавався в один з TextView

Мінуси даного варіанту очевидні:

1. Усе знаходиться лише в одному класі
2. Кожен раз викликати лише певну функцію не давало змоги програмі стати більш гнучкою
3. Великі Switch / if, які й відповідали за логіки виконання певних операцій

*Приклад одного з умовних операторів:*

*if (operation == '+') {*

*result = tempValue1.toDouble() + tempValue2.toDouble()*

*} else if (operation == '-') {*

*result = tempValue1.toDouble() - tempValue2.toDouble()*

І це лише дві операції! А у Мене таких велика кількість! Тому це треба міняти…

**Рефакторинг**

Рефакторинг Я вирішив почати зі створення класу для математичних операцій. Його архітектура вийшла доволі простою: просто оголошення функції, її параметри, певна операція всередині та значення, яке повертає.

*Приклад однієї з функцій:*

*fun add(firstNumber: Double, secondNumber: Double): Double*

*{*

*return firstNumber + secondNumber*

*}*

Далі Я вирішив прибрати великі Switch. Ідея вирішення прийшло доволі швидко: на одній з лекції була тема, яка чудово підходить для цього, а саме заміна умовних операторів поліморфізмом. Коли Я вперше почув про це, то Мені було незрозуміла її реалізація, тому Я не сильно приділяв увагу цьому способу, але спробувати Я хотів. І фінальний результат Мене вразив. Усе працювало, а код став набагато краще! І тема, яка нещодавно була неясною, відразу почала мати вагому роль у рефакторингу Мого коду.

*Приклад коду до рефакторингу:*

*…*

*multiply\_btn.setOnClickListener { calculate('\*') }*

*…*

*private fun calculate(operationChar: Char) {*

*when (operation) {*

*'+' -> {*

*result = MathClass.Add(tempValue1.toDouble(), tempValue2.toDouble())*

*}*

*'-' -> {*

*result = MathClass.Substract(tempValue1.toDouble(), tempValue2.toDouble())*

*}*

*'/' -> {*

*when {*

*firstNumber.text=="0.0" && !secondNumber.text.isNullOrEmpty() -> {*

*result = tempValue2.toDouble()*

*}*

*!firstNumber.text.isNullOrEmpty() && !secondNumber.text.isNullOrEmpty() ->*

*{*

*result = MathClass.Divide(tempValue1.toDouble(), tempValue2.toDouble())*

*}*

*}*

*}*

*'\*' -> {*

*when {*

*firstNumber.text=="0.0" && !secondNumber.text.isNullOrEmpty() -> {*

*result = tempValue2.toDouble()*

*}*

*!firstNumber.text.isNullOrEmpty() && !secondNumber.text.isNullOrEmpty() ->*

*{*

*result = MathClass.Multiply(tempValue1.toDouble(), tempValue2.toDouble())*

*}*

*}*

*}*

*}*

Тобто код вище отримував повідомлення від кнопки з певним символом та в функції calculate() у залежності від параметру у великому Switch робив перевірку.

Мало того, що це виглядає неестетично, так ще й читаємість та масштабування стають набагато гіршими.

А от після переходу на поліморфізм усе стало набагато краще:

*Приклад коду після рефакторингу:*

…

*multiply\_btn.setOnClickListener { calculate('\*', OperatorMultiply(), false) }*

*…*

*open class Operator {*

*open fun checkTwoNumbers(firstNumber: Double, secondNumber: Double): Double*

*{*

*var mathClass = MathOperations()*

*var result: Double = 0.0*

*return result*

*}*

*open fun checkOneNumber(firstNumber: Double): Double*

*{*

*var mathClass = MathOperations()*

*var result: Double = 0.0*

*return result*

*}*

*}*

*…*

*class OperatorMultiply : Operator(){*

*override fun checkTwoNumbers(firstNumber: Double, secondNumber: Double): Double {*

*var mathClass = MathOperations()*

*return mathClass.multiply(firstNumber, secondNumber)*

*}*

*}*

*…*

*private fun calculate(operationChar: Char, OperatorClass: Operator, oneNumberOperation: Boolean) {*

*…*

*when {*

*!firstNumber.text.isNullOrEmpty() && !secondNumber.text.isNullOrEmpty() ->*

*{*

*result = OperatorClass.checkTwoNumbers(tempValue1.toDouble(), tempValue2.toDouble())*

*}*

*…*

*!firstNumber.text.isNullOrEmpty() && oneNumberOperation ->{*

*result = OperatorClass.checkOneNumber(tempValue1.toDouble())*

*}*

*}*

*…*

*}*

*}*