Kotlin курс

Структура

Вступ

Моє шануваннячко, любі друзі! Мене звуть Вадим, відсьогодні я вам буду розповідати про Kotlin, але почнемо з простого: що таке Kotlin, чому саме Kotlin та інше.

Що таке Kotlin?

Kotlin — статично типізована об'єктно-орієновна мова програмування і бла-бла-бла. Не будемо вас нудити і перейдемо відразу до основного.

Чому саме Kotlin?

Перед тим, як розпочати на екскурс в світ розробки на Kotlin, не завадило б сказати, що ідеальних мов програмування не існує. Ви не зможете вивчити один тільки Kotlin і бути дійсно затребуваним спеціалістом. Кожна мова програмування створенна щоб вирішити якусь проблему: починаючи з простоти вивчення і користування, закінчуючи будь-яким іншим інструментарієм. Яку ж проблему вирішує Kotlin — я зараз розповім.

Головна перевага Котліна перед іншими мовами програмування — відірваність від оточення. Котлін без проблем працює в різних екосистемах: *JVM* (де, наприклад, існують такі мови програмування як Java або Scala), Web (вміє компілюватись в JS або WebAssembly), Desktop (компілюється в C++) та на мобільних девайсах (Android, iOS).

Що ж воно таке? Все дуже просто — мова буде плинно допомагати вирішувати різні за направленностю задачі. Тобто, вивчаючи Kotlin, ви зможете охопити всі популярні нині платформи. Також це означає, що ви зможете, наприклад, працювати з кодом, що написаний на інших мовах програмування (Desktop — C++; JVM, Android - Java; iOS - Swift / Objective-C і, звичайно, Web - JS / WASM).

Крім того, Котлін дуже простий та консистентний. Давайте ж, перейдемо до діла!

Як буде будуватись курс?

Якщо я вас все ж зацікавив, давайте розглянемо, як буде будуватись наш з вами курс.
При вивченні будемо користуватись наступними правилами:

- Створюємо проблему: для того, щоб пояснити, що для чого потрібно, створимо проблему та вирішимо її.
- Теорія: перед тим, як перейти до вирішення, розглянемо теоретичну частинуюю
- Вирішуємо задачу: беремо до уваги теорію та вирішуємо нашу проблему.
- Спробуй сам: залишаємо можливість попрактикуватись вам самому.

Це головні принципи курсу. Я не буду розповідати щось нове, але постараюсь розповісти зрозуміло.

Середовище розробки

Для початку, давайте розберемось з місцем, де код будемо писати код та його запускати. Зазвичай при роботі з Котлін використовують Intellij Idea, тому давайте візьмемо її.

Як встановити

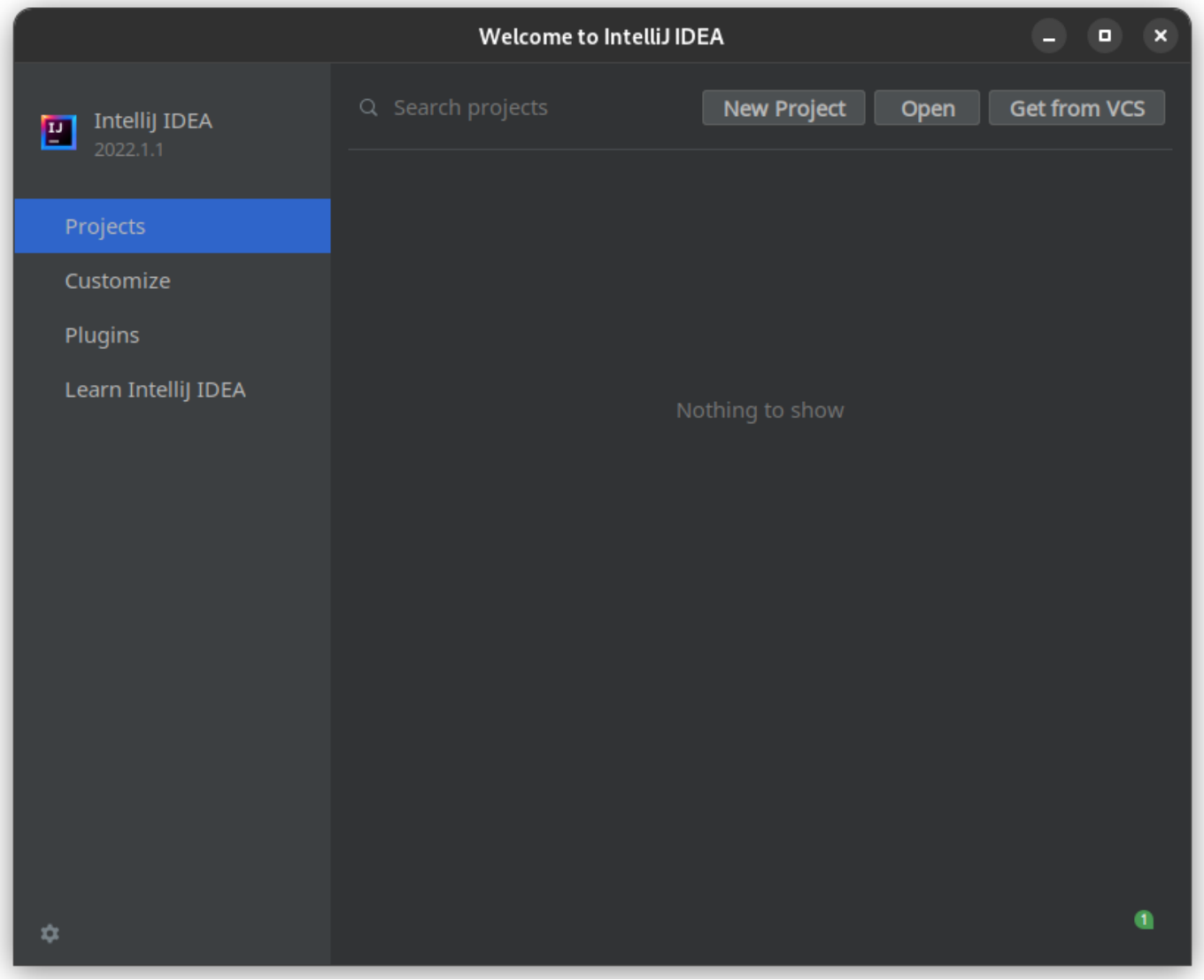
Intellij Idea поділяється на дві версії: **Community Edition** (безкоштовна версія) і **Ultimate Edition** (версія з передплатою).

Вибирайте те, що підходить саме вам, але, якщо чесно, для новачків різниці немає.

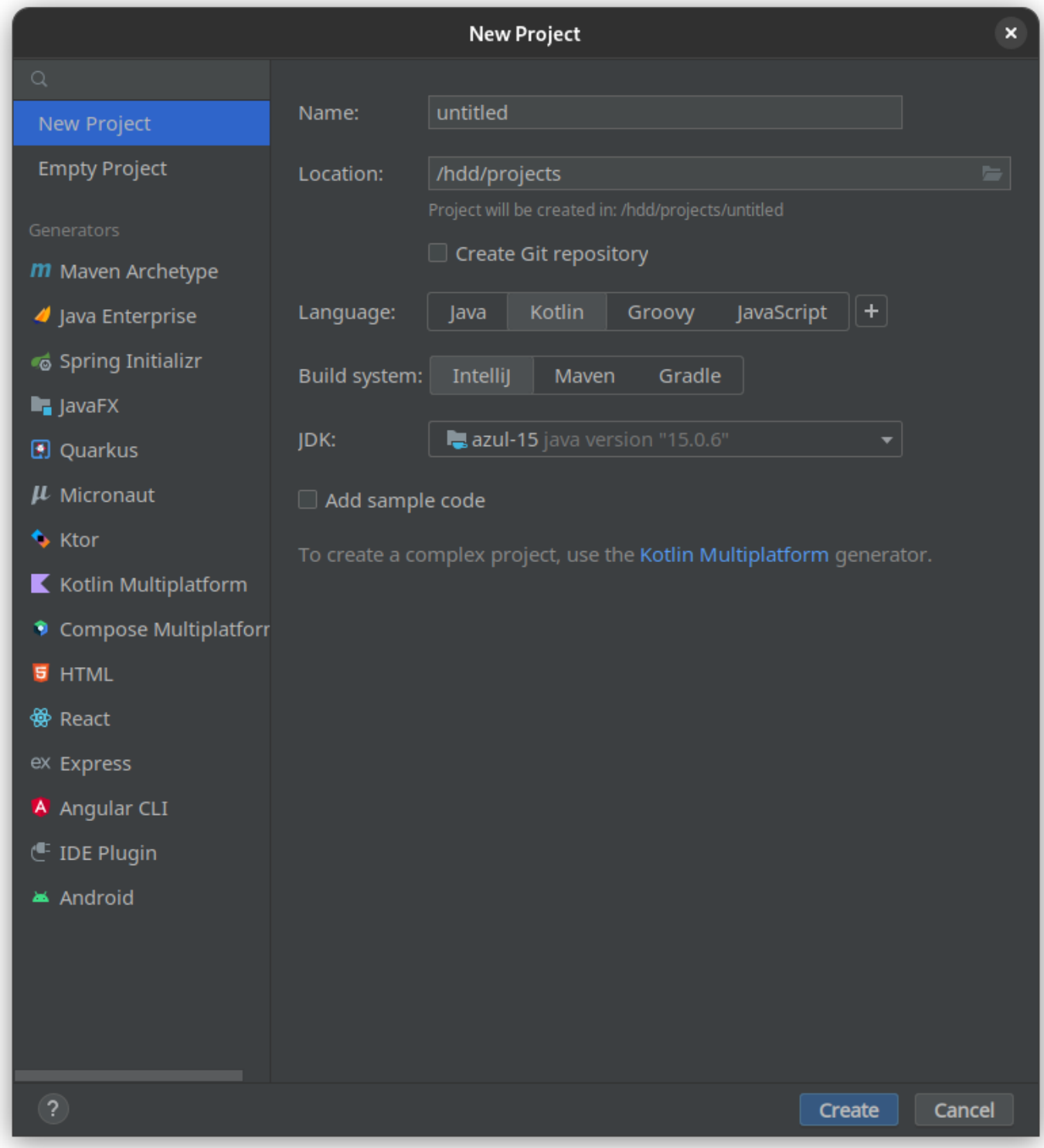
Завантажити можна зі спеціальної утиліти (Jetbrains Toolbox) або скачати напряму тут.
Далі встановлюйте настройки, що підходять саме вам (тему та ін.).

Створення проєкту

Нумо створимо наш проєкт:



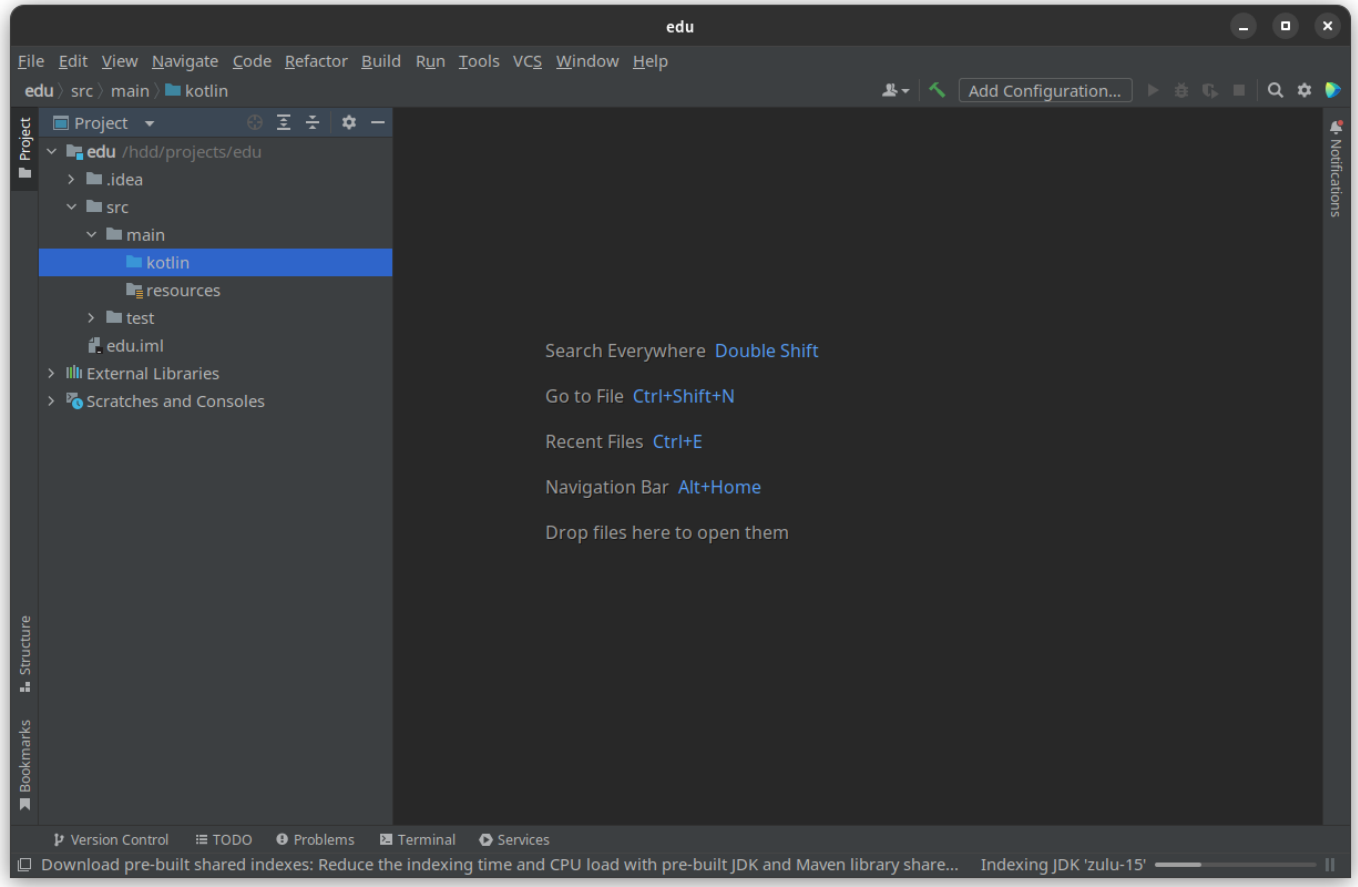
Натискаємо на «New Project» при створенні проєкту:



Називати проєкт можна як вашій душі завгодно, але тільки латиницею.

Мова, звичайно, Kotlin, а Build система Intellij (поки не будемо розглядати, що це).

Натискаємо «Create» та створюємо проєкт.
В новому віконці з'явиться наш проєкт зі стандартною структурою.



Поки можете створити файл «Main.kt», щоб пізніше там програмувати.

Для того, щоб створити файл, клацніть правою кнопкою мишки на папку «kotlin» → «New» → «Kotlin File» та впишіть «Main».

Чтобы его создать нажмите правой кнопкой мыши на папку «kotlin» → «New» → «Kotlin File» и впишите «Main».

Основи Kotlin

Що ж, перейдемо нарешті до основ мови програмування.

Змінні

Що таке змінна? **Змінна** — символ або набір символів, які являють собою якусь величину чи значення.
Навіщо вони потрібні? - для запису результатів ваших обчислень та їх подальшого використання.
Наприклад, ви зробили якусь частину обчислень, записали результат у змінну і перевикостали ваші обчислення пізніше. Для цього й існують змінні!

Як створити змінну в Kotlin

Щоб створити змінну в Kotlin, ми використовуємо ключеве слово `var` (від англ. variable).

```
var [назва]: [ТИп] = [значення]
```

Складається з:

- Назва змінної, як і назва будь-якої іншої сутності - має бути унікальним, починатися з маленької літери і не мати прогалин. Якщо у змінній кілька слів, всі слова після першого починаються з великої (наприклад: `kotlinCourse`. Цей вид запису називають *lower camel case**).
- Тип - сутність, що описує наші дані і буде міститись в змінній. Наприклад, це може бути ціле число (тобто `Int`) або число з комою (тобто `Double`).
  Для того щоб задати якесь значення змінній, використовується знак `=` (і ніяк інакше).
  Після оголошення такої змінної її можна змінити в такий спосіб:

```
[назва] = [новеЗначення]
```

Цей вид змінних може змінюватись впродовж роботи програми (данний термін у програмуванні ще називають *мутабельністю*, від англ. mutable — змінюваний).
Але, зачекайте, а бувають змінні, що не змінюються? Сама ж назва кричить про те, що «я змінююсь!».

Незмінювана змінна (immutable variables)

Як би це безглуздо не звучало, але такий вид змінних існує. Існує для того, щоб ви зберігали дані, що *не змінюються* впродовж роботи програми.

Наприклад, якщо вам потрібно одноразово зробити якесь обчислення та переконатися, що ви ніде його випадково не зміните (щоб не викликати помилки у роботі вашої програми).
Запис нічим не відрізняється від змінюваної змінної, за винятком того, що для незмінюваної змінної ми використовуємо ключове слово `val`.

```
val [название]: [Тип] = [значение]
```

Подібний тип змінної варто використовувати завжди, за винятком ситуацій, де вам потрібно змінювати значення. Це спростить код і позбавить вас від проблем.

Типи даних

З видами змінних розібрались, а що щодо типів даних? Які типи даних існують в Kotlin? Давайте розглянемо типи, які є фактичним ґрунтом будь-якої програми:

- **Int** — ціле число, що є обмеженим від `-2147483647` до `2147483647` (число обмежене 32-я бітами).
- **Float** — число з плаваючою точкою (або, якщо простіше, число з комою), що моє таке ж, як й *Int*, обмеження в вигляді 32-битной размерности (тобто числа до `340,282,346,638,528,860,000,000,000,000,000,000,000.000000`).
- **Long** — це тей же самий Int, але відрізняється більшою розмірністю в два рази (до `9,223,372,036,854,775,807`).
- **Double** — це той же Float, але знову ж, більшої розмірності (десь $1.7 \cdot 10^{308}$).
- **String** — простий текст. Не має обмежень, якщо не враховувати RAM.

Всі ці типи можна записати наступним чином:

```
val integer: Int = 999
val long: Long = 999_999_999 // для простоты чтения подобных чисел, можно использовать '_'
val float: Float = 1.0f // добавляется 'f' для явного указания, что мы вводим флоат
val double: Double = 999.99
val string: String = "я строка"
```

Також існують деякі інші вбудовані типи даних, але ми їх поки що розглядати не будемо.

Висновок

Підіб'ємо підсумок зі змінними:

- Змінні діляться на два типи: змінювані `var` і незмінюванні `val`.
- у змінних завжди є назва, яка має починатися з маленької літери, а наступні слова – з великої. Також назва має бути унікальною.
- У змінних завжди є тип - сутність, що описує дані або набір даних (наприклад, числа).
- У змінних завжди є якесь значення зазначеного типу (сутності).

- бувають такі типи даних: цілі числа (`Int` і `Long`), числа з комою (`Float` і `Double`) та рядки (звичайний текст, `String`).

Функції

Уявіть будь-яку однотипну дію або декілька дій, яку ви робите кожен день, і дайте їй якусь назву. За приклад такого набору дій візьмемо біг.

Біг – однотипна дія перебирання ногами, що залежить від деяких змінних (наприклад, вашої фізичної форми, темпу чи втоми).
Всі подібні дії у світі програмування називають *функціями*.
Що таке функція? **Функція** — це фрагмент коду, який має унікальне ім'я, тип, що повертається (aka значення або величина) та/або параметри (ті самі змінні, від яких залежить однотипна дія).

- Фрагмент коду - це ті самі дії, які ви робите, наприклад, при тому ж бігу. Виділення якогось коду під функцію зроблено лише одного: позбутися однакового коду.
- Параметри функції – це значення, від яких залежить ваш фрагмент коду. Наприклад, числа над якими виконуватимуться деякі обчислення функції. Параметри не можуть змінюватися (їм не можна надати інше значення).
- Тип, що повертається (тип результату роботи) - це те, що ви отримуєте в результаті вашої дії. Наприклад, ви отримуєте кількість часу, коли ви бігали, і це, і є результатом (тобто називається *типом повернення* у програмуванні). Записати функцію можна так:

```
fun [назваФункції]([назваПараметру]: [типПараметра]): [типРезультату] {
        ...
}
```

Не лякайтеся, що тут так багато тексту! Насправді все дуже просто:

- Ключове слово `fun` використовується для оголошення того, що ми будемо декларувати (створювати) нову функцію.
- Далі у нас йде назва функції, що має ті ж правила, що й назви змінних - унікальна назва без прогалин, перший символ маленький і таке інше.
- Далі відкриваються круглі дужки, де вказуються параметри функції. Навіть якщо параметри функції не потрібні — вони все одно вказуються.
    - Далі йде назва аргументу, де застосовуються такі ж правила як і до назви функцій та змінних

    - Після чого йде обов'язкова вказівка типу (сутності) нашого аргументу.
    - Якщо у нас кілька аргументів, далі йде `,` та вказівка наступного аргументу.
    - Ну і закриваємо дужки

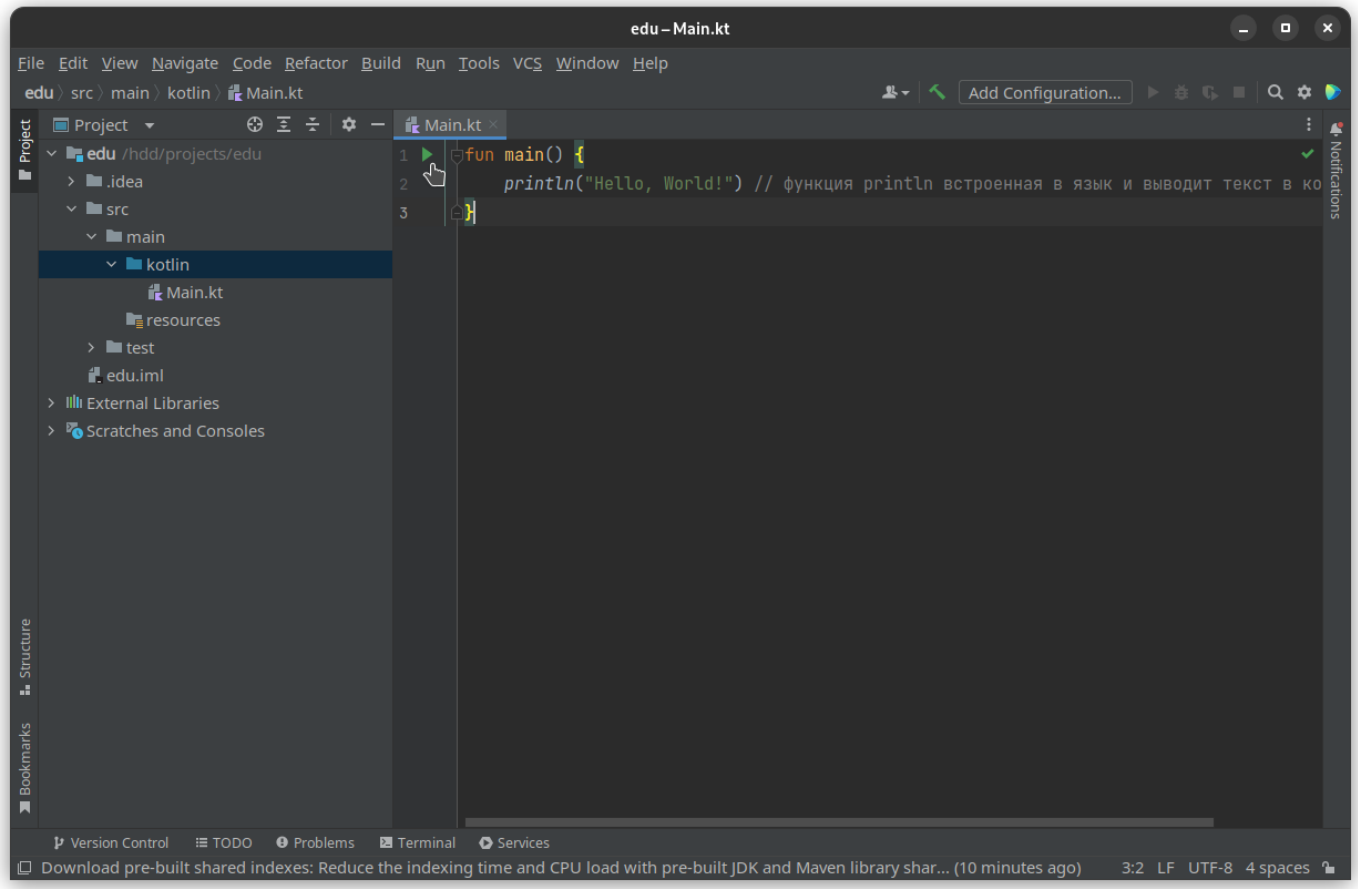
- Далі йде вказівка типу, що повертається до функцій. Але варто враховувати, що функції не завжди повинні щось повертати (якщо функція нічого не повертає, вона має тип `Unit`).
- Далі відкриваються фігурні дужки (все, що всередині називають **тілом функції**), де вже йде наш фрагмент коду. Якщо нам потрібно щось повернути з функції, використовується ключове слово `return` (наприклад, щоб повернути ціле число ми напишемо `return 10`).
  Тепер, коли ми бачили формули для наших програм, давайте перейдемо до бойового завдання!

Приклад

Щоб закріпити наші знання, нумо спробуймо щось зробити.
У кожному туторіалі всі спочатку роблять 'Hello, World'. Давайте не змінюватимемо традиціям.
Для того щоб створити програму, логічно, що нам потрібно визначити точку її початку. У Kotlin, точку початку оголошують створюючи функцію `main()` (яка, до речі, не повертає жодного результату).

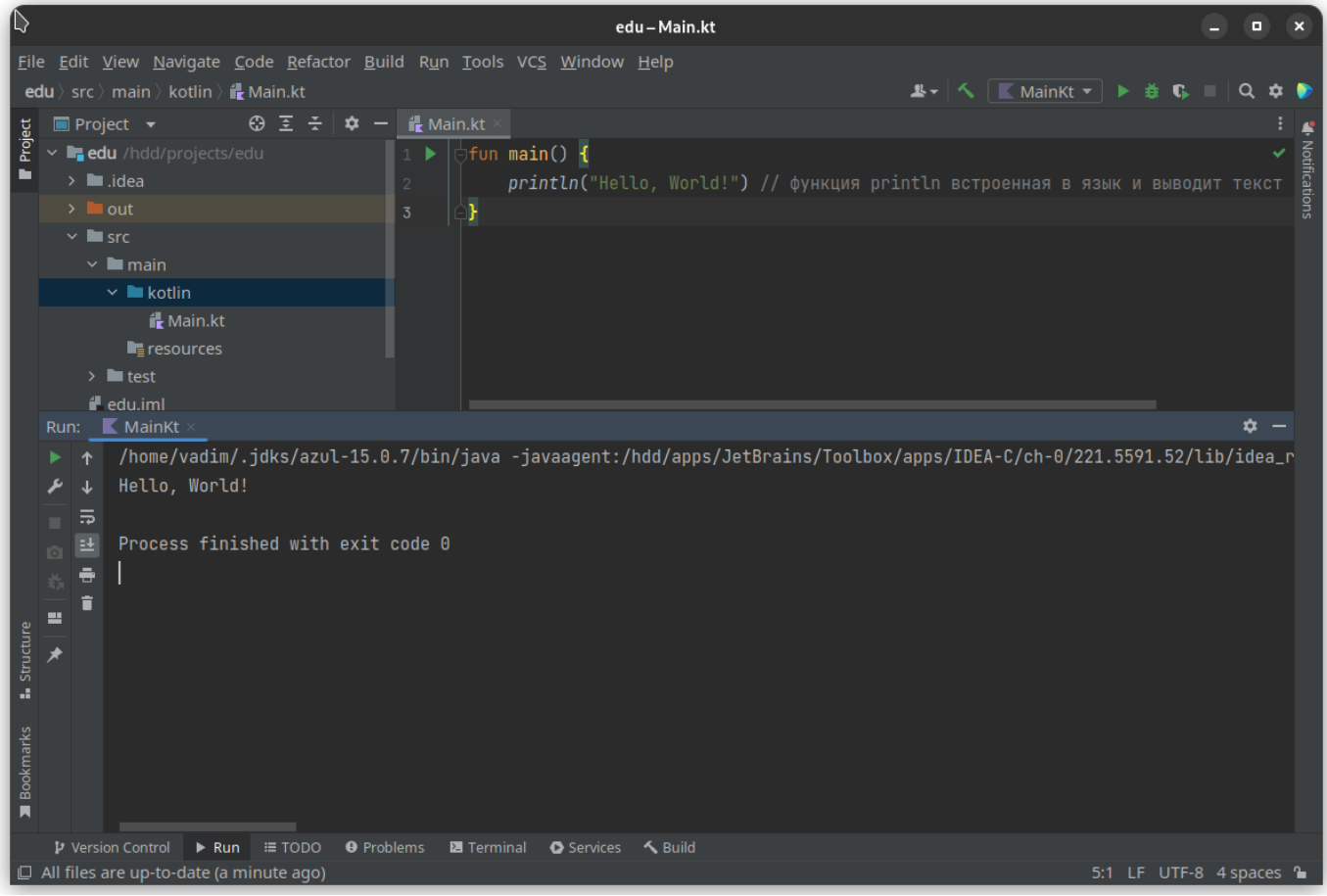
```
fun main(): Unit {
        println("Hello, World!") // функция println встроенная в язык и
выводит текст в консоль
}
```

Для того, щоб запустити нашу програму, в IDE ми натискаємо наступну кнопку:



І в списку, що випав, натискаємо 'Run Main.kt'.

І отримуємо результат:



До речі, тип `Unit` є стандартним для всіх функцій, тому вказувати його не обов'язково.

Перейдемо ж, до більш складних завдань.

Мені недовподоби торкатися математики при поясненні деяких тем, але давайте все ж таки, спробуємо вирішити фукцію засобами мови програмування.
За рівняння візьмемо наступне нескладний у розумінні вираз:
$f(x) = x + 1$
У нас деяка функція з параметром `x` (число, очевидно), яка повертає вказане число плюс один.
Як подібна функція виглядатиме на котлині?

```
fun f(x: Int): Int {
        return x + 1
}
```

Начебто нескладно, так? Головне, спочатку, підглядувати на формулу того, як будується функція.
Давайте, викличемо цю функцію:

```
fun main() { // оголошуємо початок нашої програми
        val x: Int = 10 // задаємо довільне число
        val result: Int = f(x) // отримуємо результат від виконання функції
        println(result) // функція println є вбудованою в Kotlin, що пише
текст у консоль. Ми же маємо текст (результат) від виконання функції в
змінній.
}
```

Можна спростити до наступного варіанту:

```
fun f(x: Int) = x + 1
```

Код, який має лише одну послідовність дій, можна оголосити через `=`. Тип, що повертається, буде визначений автоматично мовою програмування. У нашому випадку, це буде `Int`, тобто - ціле число.

До речі, той самий «Hello, World!» можна було написати так:

```
fun main() = println("Hello, World!")
```

Але найчастіше код у вас буде довшим, ніж одна послідовність дій.

До речі, це також працює зі змінними:

```
fun main() {
        val x = 10
        val result = f(x)
        println(f(x))
}
```

Закріплювати знання потрібно практикою! Для цього спробуйте написати невелику програму для вирішення наступної функції:
$f(y) = \frac{y}{2}$
Також варто відзначити, що ділення виконується за допомогою оператора `/`.
І знову незнайомий термін! Хто цей ваш оператор?

Оператори

У програмуванні оператором називають символ, що являє собою якусь дію над сутністю (наприклад число, яке ділять на інше число). Це функція, що має свій спеціальний символ у словнику мови програмування (як наприклад це з ключовими словами `fun` або `var`).
Давайте за приклад візьмемо те саме ділення. Щоб зробити ділення, ми можемо використовувати значок `/`.

```
val result: Int = a / b
```

По-іншому це можна виразити таким чином:

```
fun divide(a: Int, b: Int): Int {...}
```

Але, для зручності та розуміння, вирішили скорочувати до `/`.
Однак за вами залишається можливість викликати це словесно:

```
val result = a.div(b) // div - скороченно від divide
```

Тепер, знаючи це, спробуйте вирішити задачу самотужки:
$f(y) = \frac{y}{2}$

Ця задача досить проста:

```
fun f(y: Int): Int {
        return y / 2
}
```

Не дуже відрізняється від додавання чисел.
Але які ще існують оператори?

Види операторов

Оператори поділяють такі типи:

- арифметичні - додавання чисел, віднімання, множення, ділення.

- `+` оператор додавання значень (`a + b`)
- `-` оператор віднімання значень (`a - b`)
- `*` оператор множення значень (`a*b`)

- `/` оператор ділення значень (`a/b`)
- `%` оператор ділення із залишком (`22 % 4` рівнятиметься `2`)
- складені оператори:

- `+=` оператор додавання до існуючої величини ще одне значення. Еквівалентно наступному: `a = a + b`.
- `-=` оператор віднімання значення від існуючої величини. Еквівалентно наступному: `a = a - b`.
- `*=` оператор множення існуючої величини на деяке значення. Еквівалентно наступному: `a = a * b`.
- `/=` оператор ділення існуючої величини на деяке значення. Еквівалентно наступному: `a = a/b`.
- `!=` оператор 'не дорівнює', що відповідає `!(n == 1)` (дужки використовуються для підсумування результатів виразів)
- логічні - порівняння значень:
- `>` оператор більше ніж (`a > b`)

- `<` оператор менше ніж (`a < b`)
- `&&` оператор 'і' (`a > b && b < c`: `a` більше `b` і `b` менше `c`)
- `||` оператор 'або' (`a > b || c < a`: `a` більше `b` або ж `c` менше `a`)
- `==` оператор рівності двох значень (`a == 5`)
- `!` оператор протилежності (якщо у нас `false`, перетворюється на `true` і навпаки)
- складені оператори:
- `>=` оператор більше або дорівнює (`a >= 5`)
- `<=` оператор менше або дорівнює (`a <= 5`)
- та умовні, які працюють з логічними операторами:
- `if` оператор 'якщо це правда (true), то щось'. Працює в кооперації з `else`: "якщо це правда, то це, але якщо ні, то (фрагмент коду з else)".
- `when` оператор для кількох 'якщо це правда, то ..'

Логічні оператори

Давайте розглянемо логічні оператори, що в мові повертають тип `Boolean`, де є тільки два можливі значення (варіанта): `true` (істина) та `false` (не істина, тобто брехня).

Для уточнення роботи деяких операторів вирішимо кілька дуже простих завдань:

№1.

> Створіть функцію, що буде говорити чи x більше y.
> x, y — целые числа.

```
fun isBigger(x: Int, y: Int): Boolean {
        return x > y
}
```

№2.

> Створіть функцію, що перевірятиме чи не є число нулем.

Ддля цього нам знадобиться оператор 'не рівне' (`!=`).

```
fun isNonZero(x: Int): Boolean = x != 0
```

№3.

> Створіть функцію, що перевіряє, чи число ділиться націло на 3.

Щоб вирішити це завдання, нам потрібно буде скористатися оператором залишку від ділення (тобто `%`).

Якщо число ділиться націло на інше число, логічно, що залишком від ділення буде нічого, тобто `0`.
Це означає, що нам потрібно скомпонувати два оператори — залишок від ділення та рівності. Тобто потрібно зробити наступне `n % 3 == 0` (`n % 3` виконується перед так як вирази читаються зліва направо).

```
fun isDivisibleOnThree(x: Int): Boolean {
        return x % 3 == 0
}
```

Умовні оператори

Давайте більше приділимо увагу умовним операторам.
Для того, щоб зробити програму, що спирається на якісь умови, які потрібно обробити (наприклад, як той випадок, що ми обговорили вище), використовують умовні оператори.
Сама назва говорить нам про те, що ми маємо якусь умову. Давайте ж розберемо якісь види умовних операторів є в Kotlin.

If else

Одним із умовних операторів є `if-else`.
Дуже проста конструкція, що означає «якщо це істина, то зроби це, якщо ж ні, то це».
Записується так:

```
...
val isBigger: Boolean = a > b
if(isBigger) {
        println("a більше b!")
} else {
        println("b більше a!")
}
```

Але що ж робити, якщо у нас кілька умов? Наприклад, нам потрібно дізнатися найбільше з трьох довільних.
Використовуючи логіку, можна прийти до того, що в `else {...}` можна дописати ще один `if`. І це буде правильно! Це спрацює.

```
fun getBiggest(a: Int, b: Int, c: Int) {
        if(a > b && a > c) { // тут, до речі, використовується логічний
оператор 'і'
                return a
        } else {
                if(b > a && b > c) {
                        return b
                } else {
                        return c
                }
        }
}
```

Але, код помітно, став набагато складнішим. Може можна спростити?
Так, дійсно, цю конструкцію можна спростити.
Для if, як і для else застосовується одне спрощення:

- Якщо у вас лише один ланцюжок дій, то вказувати фігурні дужки необов'язково. Тобто в результаті вийде таке:

```
if(a > b && a > c)
        return a
```

```
else if(b > a && b > c)
        return b
else return c
```

Тепер код став помітно кращим, аніж до цього. Але, друзі, на цьому магія не закінчується:

```
return if(a > b && a > c) {
        a
} else if(b > a && b > c)
        b
else c
```

Воу-воу, що це таке? Якщо ви раніше вивчали інші мови програмування, ви, можливо, знаєте про «тернарний оператор». Що ж, у Kotlin вирішили зробити можливість використовувати умовний оператор if-else (і забігаючи наперед, так само оператор `when`) як *вираз* (все, що може виражати значення: сире значення типу `10`, функція, що повертає якесь значення та інше, що в результаті повертає нам якесь значення, називають *виразом*).

Що це означає? Це означає, що фрагменти коду в if будуть виступати як умовні функції, які повертають якесь значення (число або щось інше) з кожної обробленої гілки умови. Для того, щоб повернути щось із фрагмента коду, потрібно написати значення (або змінну/функцію, яка матиме потрібне нам значення) останнім у нашому фрагменті коду.

Давайте закріпимо матеріал зробивши нескладне математичне рівняння:

$$f(x) = \begin{cases} x & \text{якщо } x \geq 0 \\ 2x & \text{якщо } \mathrm{x} < 0 \end{cases}$$

Тут у нас тільки два значення, що робить задачу дуже примітивною:

```
fun f(x: Int): Int {
        return if(x >= 0)
                x
        else 2 * x
}
```

Добре, з цим розібралися. Але що робити, якщо у нас буде більше умов? Робити нескінченні ланцюжки `if-else`? Як би не так.

When

Для великої вибірки умов створили оператор `when`.
Щоб зрозуміти, для яких випадків він використовується, давайте вирішимо таке завдання:

> Створіть функцію, яка повертатиме день тижня за його порядковим номером.
> Тобто: якщо в функцію ввести параметр '1', функція поверне «Понеділок». І так далі.

Для цього, ми можемо використати оператор `when`, який буде куди зрозумілішим нескінченних ланцюжків `if-else`.

```
fun getDay(ordinal: Int): String {
        return when {
                ordinal == 1 -> "Понеділок"
                ordinal == 2 -> "Вівторок"
                ordinal == 3 -> "Середа"
                ordinal == 4 -> "Четвер"
                ordinal == 5 -> "П'ятниця"
                ordinal == 6 -> "Субота"
                ordinal == 7 -> "Неділя"
                else -> "Вказаний невірний номер"
        }
}
```

Варто відзначити `else`, який також існує в операторі `when`. Працює аналогічно, обробляючи умову, яка не була задоволено доти.
У нашому випадку, якщо ввести в функцію число більше 7, то повернеться повідомлення про те, що вказаний невірний день.
До речі, якось однотипно виглядають умови, чи не так? "Якось дуже багато мороки" - сказав Kotlin, і зробив чергове спрощення:

```
fun getDay(ordinal: Int): String {
        return when(ordinal) {
                1 -> "Понедельник"
                2 -> "Вторник"
                3 -> "Среда"
                4 -> "Четверг"
                5 -> "Пятница"
                6 -> "Суббота"
```

```
                7 -> "Воскресенье"
                else -> "Указан неверный день"
        }
}
```

Як усе зрозуміліше і очевидніше стало, чи не так?
Для закріплення матеріалу вирішимо наступне математичне рівняння:

$$f(x) = \begin{cases} x + 1 & \text{если } x < 0 \\ 2x & \text{если } x \geq 1 \leq 10 \\ x + x & \text{если } x > 10 \\ 0 & \text{в ином случае} \end{cases}$$

У цьому виразі ми маємо цілих 4 умови. Для цього підходить якраз таки наш `when`:

```
fun f(x: Int): Int {
        return when {
                x < 0 -> x + 1
                x >= 1 <= 10 -> 2 * x
                x > 10 -> x + x
                else -> 0
        }
}
```

Також ви можете самі оголошувати свої варіанти операторів, але про це ми поговоримо якось в інший раз.

Стандартні функції в Kotlin

Що ж, ми розглянули ґрунт, на якому вже може будуватися програма.
Раніше ми вже розглядали одну з вбудованих функцій — `println`. Ця функція друкувала в наш термінал (консоль) текст, який ми їй задавали. Давайте розглянемо й інші.
Перейдемо до завдання: нам потрібно написати програму, що буде перемножувати введені користувачем цифри. Досить просто.
Для цього нам знадобиться функція `readln`, що читає введення користувача з консолі.
Тобто, у нас буде щось на кшталт:

```
fun main() {
        val number1: Int = readln()
        val number2: Int = readln()
```

```
        println(number1 * number2)
}
```

Ось і вся програма! Але, спробувавши її запустити, ми отримаємо помилку:

```
Type mismatch: inferred type is String but Int was expected
```

Переклавши на українську мову ми отримаємо наступне:
`Невідповідність типів: передбачуваний тип - String, але очікувався Int.`
Що це означає? Це означає, що нашим змінним `number1` та `number2` типу `Int` (цілі числа) задається невідповідний тип `String` (рядок, aka звичайний сирий текст).
На жаль, оскільки введення користувача може бути будь-яким: від цифр, до літер та будь-яких інших символів, функція `readln` повертає тип `String` (рядок, що може містити не тільки цифри). Щоб це виправити, ми повинні перетворити рядок на ціле число (ну правда, бажано, перед цим перевірити введення користувача).
У Kotlin є функція `toInt` для строк (ak рядків або тексту), що дуже полегшує нам життя. Вона конвертує рядок у цифру шляхом її парсингу.

```
fun main() {
        val number1: Int = readln().toInt()
        val number2: Int = readln().toInt()
        println(number1 * number2)
}
```

Подібні функції також існують і для інших типів-чисел:

- `String.toDouble()`
- `String.toShort()`
- `String.toFloat()`

- `String.toLong()`
  Також подібні функції існують і між числами. Наприклад, ціле число може перетворюватися на число з плаваючою точкою (aka з комою, типу `Double`).
  Для чого це потрібно? Ну, наприклад, вам потрібно скласти одне ціле число і одне число з плаваючою точкою. Оскільки Kotlin - це строго-типізована мова, просто так скласти цифри різних типів не вийде. Залежно від того, що вам потрібно, ви можете привести число з плаваючою точкою в ціле і навпаки.

```
val x = readln().toInt()
val y = readln().toDouble()
val double: Double = x.toDouble() + y
println(double)
val integer: Int = x + y.toInt()
println(integer)
```

Взагалі, якщо ви спробуєте скласти `x+y` без перетворень, Kotlin автоматично виводитиме тип `Double`. Але, під капотом, робиться те, що ми щойно розглянули. Але подібне працює лише з числами.

```
val x = readln().toInt()
val y = readln().toDouble()
val result: Double = x + y
println(result)
```

До речі, функція `println` також виводить рядок у консоль. А ми їй віддаємо число. Чому для неї таких обмежень немає?
Все просто: функція `println` приймає "Any" (тобто будь-який тип даних) і неявно викликає на цьому типі `toString()` (така ж вбудована функція в мову, що застосовується на будь-який створений об'єкт), що повертає рядок з нашою цифрою.
Усередині там щось подібне:

```
fun println(value: Any) {
        val string: String = value.toString()
        // ...
}
```

Які ще цікаві функції існують?
Давайте, вирішимо наступне завдання, заодно згадавши про умовні оператори:

$$f(x) = \begin{cases} 3x & \text{если } x < 0 \\ x^5 & \text{если } x \geq 1 \leq 100 \\ 1 & \text{в ином случае} \end{cases}$$

І тут ми бачимо ступінь! Що ж, знаючи, що таке степінь, неважко вирішити це завдання.
Я віддаю перевагу для більш ніж двох гілок з умовами, використовувати `when`.

```
fun f(x: Double) {
        return when {
                x < 0 -> 3 * x
                x >= 1 && x <= 100 -> x * x * x * x * x
                else -> 1
        }
}
```

І наше завдання вирішено! Але якщо ми говоримо про вбудовані функції, то напевно є якесь спрощення? Особливо, якщо думати про те, що було б, якщо степінь була б більшою.
Так, спрощення є. Це функція `pow`.

```
fun f(x: Double) {
        return when {
                x < 0 -> 3 * x
                x >= 1 && x <= 100 -> x.pow(5)
                else -> 1
        }
}
```

Дуже зручно! Особливо якщо у нас довільна степінь.
Давайте ж виведемо результат у консоль:

```
fun main() {
        val input: Double = readln().toDouble()
        val output: String = f(input).toString()
        println(output)
}
```

Начебто не складно, чи не так? Заодно закріпили інші вбудовані функції.
Подібних функцій досить багато, тому ми будемо їх розглядати під час проходження інших тем.

Область видимості

Тепер перейдемо до досить цікавої, але трішки складної теми — області видимості. Раніше ми розглядали змінні та функції, тому тепер варто розглянути випадки, коли функція або змінна може бути недоступна або навпаки доступна в деяких місцях.

Область видимості (англ. scope) в **програмуванні** — важлива концепція, що визначає доступність змінних, функцій та інших сутностей. Ця концепція поділяє змінні, функції тощо на глобальні та локальні.
Давайте розглянемо на прикладі:

```
fun foo() {
        val a = 1
}
fun main() {
        println(a + 1)
}
```

Даний код викличе помилку:

```
Unresolved reference: a
```

Що означає, що змінна створена у функції `foo()` недоступна у функції `main()`.
Чому? У цьому конкретному випадку, `a` не може існувати і в теорії, тому що вона створюється при виклику функції `foo`, а вона не викликається.
А якщо ми її викличемо?

```
fun foo() {
        val a = 1
}
fun main() {
        foo()
        println(a + 1)
}
```

Тепер вона створена і, по-ідеї, програма повинна працювати, але як би не так, ми отримаємо ту ж помилку:

```
Unresolved reference: a
```

Справа в тому, що змінні видно тільки в місці створення і нижче за ієрархією.
Нижче за ієрархію? Давайте перепишемо наш код так щоб наша змінна була доступною:

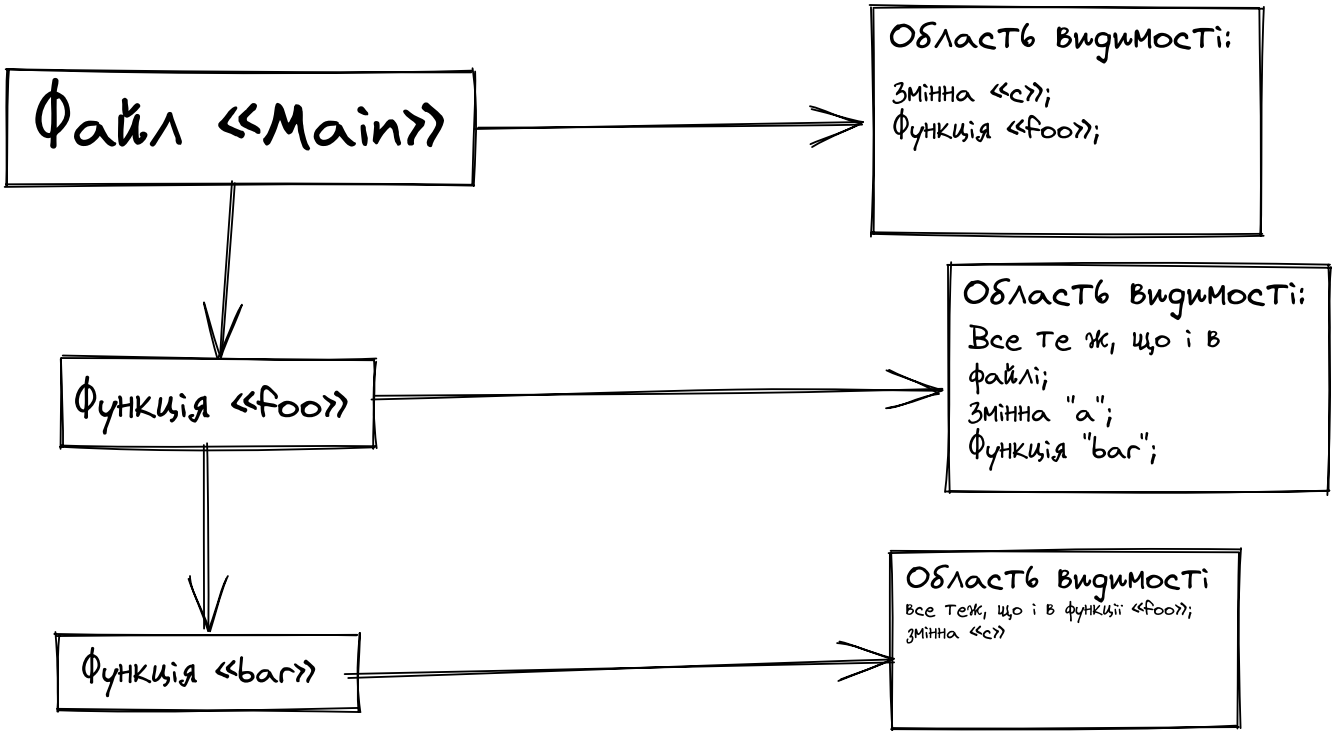
```
var a = 0
fun foo() {
        a = 1
}
fun main() {
        foo()
        println(a + 1)
}
```

(Для наочності змінюємо змінну під час виклику `foo()`)

Подібне і означає «нижче за ієрархію».
Функція, що використовує змінну `a` успадковує область видимості (aka scope) файлу, у якому її створили.
І так працює із будь-яким місцем, де змінну створюють. Навіть у функції:

```
var c = 2
fun foo() {
        var a = 2 // створюємо функції на рівні функції.
        c = 4 // змінна, що знаходиться поза функцією (в файлі), доступна
        fun bar() {
                val b = a.pow(2) // змінна `a` доступна у функції `bar` через
успадкування області видимості
                a = b
        }
        bar() // функція доступна у місці її створення (декларування)
        println(a) // змінна доступна у місці її створення
}
```

Так, так, у котлині можна навіть створювати функції у функціях, не дивуйтесь! Хоча зараз не про це.
У цьому випадку, функція `bar` успадковує область видимості файлу, функції `foo` і так може тривати до безкінечності. Взагалі фігурні дужки `{}` можна розглядати як оператор, який створює нову область видимості.
Давайте візуалізуємо те, як будується наша область видимості:



Тобто кожна нова область видимості успадковує «батьків», в яких вона створюється. Батьки (все, що вище за ієрархією), не бачать створене нижче за ієрархією.
PS: Взагалі змінні, що створюються в функціях, називають - **локальними змінними** (функції ж, логічно, локальними функціями).
А змінна, яку ми створювали поза функцією — «глобальною». Вона помітна скрізь, починаючи з того ж файлу, закінчуючи іншими.
Закінчуючи іншими? Так само, як ми створювали файл під ім'ям «Main», ми можемо створити будь-який інший файл. Як мінімум, щоб не тримати весь код в одному файлі. Це спростить навігацію за кодом у проектах трошки складніших за ті, які ми робили раніше.
А що буде, якщо створити ще один файл, у якому ми створимо деякі функції та змінні? Що ж, перевіримо:

```
// File: another.kt
val abc = 999_999_999
fun someFunction() {
        println("someFunction()")
}
```

Перейшовши у файл «Main» і спробувавши викликати ці функції, на нас чекає успіх:

```
fun foo() {
        val a = 2
```

```
        println(a + abc) // получаем переменную с файла `another.kt`
}
```

Що це означає? А це означає те, що файл також, як і наприклад функція, має дочірній scope (область видимості) і це деякі інші файли.
Деякі інші файли? Не всі?
Справа в тому, що файли ідентифікуються не тільки за їхньою назвою, а й за їхнім **пакетом**.
Пакет? Логічно припустити, що ніхто не мав на увазі поліетиленовий або якийсь інший пакет, а якийсь унікальний ідентифікатор.
Що за унікальний ідентифікатор та навіщо він?
Все для того ж, для чого створюються інші файли: для зручності. Потрібно ж розділяти та сортувати написаний код.
З реальних прикладів, ви можете взяти системні папки типу Music, Videos, Images та інші, що містять інформацію тільки певної категорії.
У Котліні подібна система категоризації коду, єдине, що відрізняється, — це термін (**пакет**).
Власне, як і з системними папками, ми можемо робити структуру нашого проекту поділяючи на якісь осмислені частини.
Наприклад, для будь-яких математичних обчислень ми можемо створити такий пакет:
`math.calculations`.
У файловій структурі ми просто створюємо відповідні частин пакета (розділені точкою) папки:
Тобто папку `math`, а в ній ще одну папку `calculations`. Після чого можна вже створювати наші файли з кодом.
Наприклад створимо файл з функцією, яка вирішуватиме наступний вираз:

$$f(x) = \begin{cases} 2x^2 & \text{если } x < 0 \\ x & \text{если } x \geq 1 \leq 50 \\ (x \cdot 2)^2 & \text{если } x > 50 < 200 \\ 1 & \text{в ином случае} \end{cases}$$

```
// файл Function.kt
package math.calculations // автоматично додалось нашою IDE (ідентифікатор нашого файлу)
fun f(x: Double): Double {
        return when {
                x < 0 -> 2 * x.pow(2)
                x >= 1 <= 50 -> x
                x > 50 < 200 -> (x * 2).pow(2)
                else -> 1
```

```
        }
}
```

Як ви вже помітили, зверху у нас додався рядок коду з місцем нашого файлу.
Він є обов'язковим, навіть якщо ви помістили його у відповідну папку. Це тому, що Kotlin допускає вказівку пакета вільно (тобто ви можете не створювати файлову структуру, що відповідатиме пакету).
Це робиться у нескладних проектах, де 8-10 файлів та проблем з навігацією немає, але я вам рекомендую завжди створювати відповідну файлову структуру.
Що ж, перейдемо до виклику нашої функції:

```
// файл Main.kt
fun main() {
        println(f(1.0))
}
```

По-ідеї має працювати, але запустивши ми отримаємо наступну помилку:

```
Unresolved reference: foo
```

Справа в тому, що за замовчуванням, область видимості обмежується поточним пакетом (у нашому випадку хоч він і відсутній, але він такий ж самий ідентифікатор, навіть якщо він і порожній).
Для того, щоб отримати щось з іншої області видимості (aka пакета), потрібно для початку «імпортувати» ідентифікатор.
"Імпорт" роблять за допомогою ключового слова `import`. Він завжди повинен вказуватись зверху, відразу після пакета (ну або за його відсутності, просто зверху).
Схема імпорту така:

```
import [пакет].[ідентифікатор]
```

Тобто, щоб викликати функцію `f(x: Double)`, нам потрібно зробити наступне:

```
// файл Main.kt
import math.calculations.f

fun main() {
        println(f(1.0))
}
```

І у нас все запуститися успішно!
Але якщо без імпортування ідентифікаторів інших пакетів не видно, то чи можна створювати дублікати назв?
Так, ви можете створювати дублікати імен за винятком ситуацій, коли ви намагаєтеся створити однаковий ідентифікатор в одному конкретному скоупі (області видимості).
Тобто наступне заборонено:

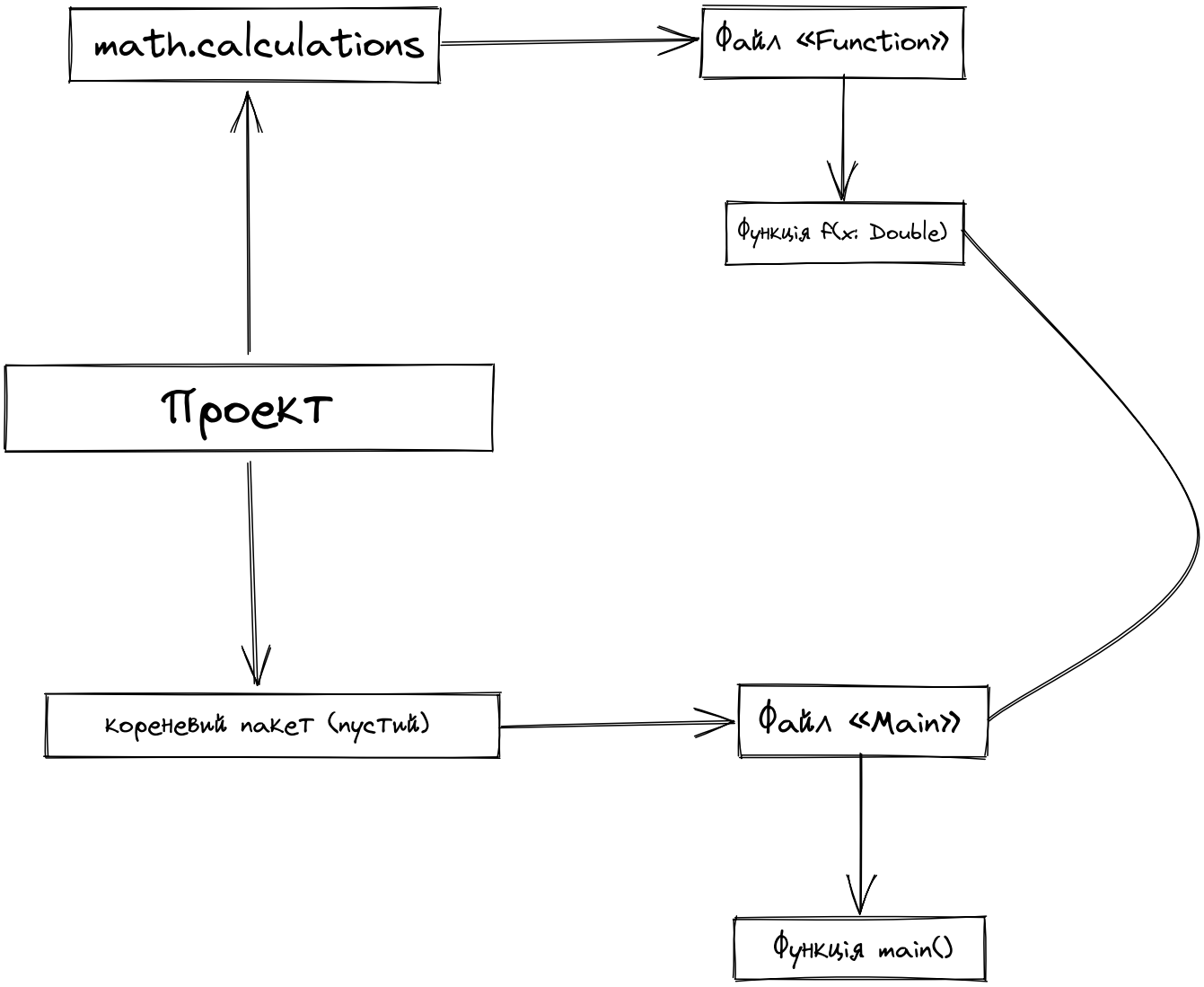
```
fun main() {
        val a = 1 // Conflicting declarations: val a: Int, val a: Int
        println(a)
        val a = 2 // Conflicting declarations: val a: Int, val a: Int
        println()
}
```

Але, таке можливо:

```
val a = 1

fun main() {
        println(a)
        val a = 2
        println(a)
}
```

Справа в тому, що пріоритетним простором імен (з нашими ідентифікаторами) є поточна область видимості (aka скоуп).
Це все тому, що функція (або будь-яке інше місце) — це новий незалежний скоуп (область видимості). Ми не можемо бути впевнені, що рано чи пізно ми не імпортуємо якусь змінну або не оголосимо таку ж у цьому ж файлі. А вигадувати нові імена не зробить код простіше, а тільки ускладнить його.
До речі, варто зазначити, що створення дублікатів в одному пакеті неможливе.
Справа в тому, що для котліна файл не є незалежною структурною одиницею і вона існує тільки у вашій структурі.
Згадайте приклад з функцією в пакеті `math.calculations`, чи ми вказуємо конкретний файл при виклику функції або її імпорті? Ні. Тому дублікати в одному пакеті і неможливі, оскільки визначити конкретний ідентифікатор з файлу неможливо.
Що ж, для закріплення, візуалізуємо все те, що ми обговорили вище:



У нас є проект з двома унікальними пакетами: `math.calculations` та батьківським (порожнім). Файл "Main" зав'язаний на функцію `f(x: Double)` у пакеті `math.calculations` (ми це виділили лінією для візуалізації).
Що ж, підіб'ємо проміжний підсумок:

- Програма поділяється на різні області видимості (скоупи), які мають чітку ієрархію залежно від того, де та що ви створюєте.
- Ієрархія зазвичай така: область видимості на рівні пакета → область видимості на рівні декларації (функції, наприклад) → тощо (наприклад вкладені функції або умовні оператори).
- Батьківською областю видимості є пакет, у якому є наш ідентифікатор (функція, змінна). Ідентифікатори інших пакетів не видно за замовчуванням.
- При необхідності можна розширити простір імен (ідентифікатори, що видно в іншій області видимості) за допомогою імпорту (`import [пакет].[ідентифікатор]`).

Модифікатори видимості

До речі, говорячи про те, що файл — це не незалежна структура, я трішки збрехав і зараз поясню чому.
Давайте вирішимо наступний приклад:

$$f(x) = \begin{cases} x^2 & \text{если } x < 0 \\ a(x) & \text{в ином случае} \end{cases}$$

Функція $a(x)$ у нас така:

$$f(x) = \begin{cases} 2x & \text{если } x > 0 < 200 \\ 1 & \text{в ином случае} \end{cases}$$

На Kotlin нам треба написати наступне (в файлі math.calculations.Function):

```
fun f(x: Double): Double {
        return if(x < 0) x.pow(2) else a(x)
}
fun a(x: Double): Double {
        return if(x > 0 < 200) 2 * x else 1
}
```

Тепер ж, викличемо функцію `f(x)`

```
fun main() {
        val input: Double = readln().toDouble()
        println(f(input))
}
```

І на цьому наша програма, умовно, закінчена.
Подивившись на функцію `a(x: Double)` ми можемо подумати про те, що вона використовується тільки у функції `f(x: Double)` і в принципі вона ніде крім у файлі 'Function.kt' не потрібна.
Чи можна цю функцію просто ігнорувати в підказках і не імпортувати, однак, якщо таких функцій багато? Це очевидно, захаращує глобальний простір імен, навіть якщо він не імпортований.
На допомогу до нас приходять модифікатори видимості! **Модифікатори видимості** — ключові слова, що описують те, де видно ідентифікатор.
Для нашого випадку існує модифікатор `private`. Він показує, що змінна видно лише там, де її створили і нижче за ієрархією.
Насправді формула створення тієї ж функції виглядає так:

```
[visibility-modifier] fun [названиеФункции](параметр: Тип): Тип {...}
```

За умовчанням, до всіх декларацій (функцій, змінних та іншого) неявно застосовується модифікатор `public` (тобто публічний, видимий назовні).
`fun main()` → `public fun main()`.
У нашому випадку, ми робимо таке:

```
private fun a(x: Double): Double {
        return if(x > 0 < 200) 2 * x else 1
}
```

До речі, вказівка однакових приватних ідентифікаторів у одному пакеті, але різних файлах допускається, оскільки конфлікт просто неможливий.

Зі змінною буде так само:

```
private val a: Int = 0
```

Висновок

Початковий розгляд унікальності імен, створення змінних та функцій виявився не таким простим, як ви вже зрозуміли.
Як я вже згадував раніше, ідентифікатор функції будується на наступних його властивостях – це ім'я та параметри.
З урахуванням розглянутих тем: *область видимості* та *модифікаторів видимості*, ми їх також додаємо в унікальність ідентифікатора (зазвичай це називають сигнатурою).
Та й те саме ми робимо зі змінною.
Підсумковим варіантом ідентифікаторів у нас будуть:

- Функція — модифікатор видимості + область видимості + ім'я + набір параметрів (відмінність у кількості чи типі).
- Змінна – модифікатор видимості + область видимості + ім'я.

Бажано самому погратись із цим для більшого розуміння!

Цикли та Рекурсії

Тепер же, перейдемо до досить цікавої, але, знову ж, трішки непростої теми — цикли.

Щоб більше зрозуміти, що таке цикли, давайте створимо якесь завдання. Наприклад, візьмемо завдання, яке ми вирішували у минулій темі. Для того, щоб вирішити рівняння з використанням введення, ми щоразу запускаємо нашу програму. А що, якщо зробити в нашій програмі нескінченне введення, щоб щоразу не перезапускати нашу програму?

Взагалі, без нашої теми циклів це цілком можна було вирішити наступним чином:

```
fun main() {
        printlt("Введите число:")
        val input: Double = readln().toDouble()
        println("Результат: " + input.toString())
        return main() // в кінці функції просто викликаємо її ще раз
}
```

І ось, рішення знайдено!

Подібне називають рекурсією. Простими словами - це поняття оголошення (напису, опису) коду функції через саму себе. Це як матрьошка, яка в нашому випадку не має кінця.

Що ж, а як тепер завершити нашу програму? Можна, звичайно, це зробити, закривши примусово процес програми через інструменти системи або IDE, але давайте будемо людьми і зробимо якийсь механізм виходу.

Щоб сильно не морочитися, введемо умову, що для виходу з програми нам потрібно написати «:q».

```
fun main() {
        printlt("Введите число (или воспользуйтесь :q для выхода):")
        val input: String = readln() // створюємо змінну з текстом, тому що
нам потрібно перевіряти введення користувача
        if(input != ":q") {
                val input: Double = input.toDouble() // сила областей
видимості!
                println("Результат: " + f(input).toString())
                main()
        }
}
```

Це так само залишиться рекурсією, тільки вже не нескінченною (у нас з'явилася умова).

Що ж, розглянувши досить простий приклад рекурсії, до якого можна було прийти самому під час спроби розв'язати завдання з перезапуском рішення рівнянь.

Що тоді таке цикли? **Цикли** - це засоби мови, які відтворюють рекурсію. Їх також відносять до операторів, називаючи *циклічними* операторами.

Тож тепер розглянемо, як це можна вирішити іншими засобами мови. Не завжди ж ви створюватимете окремо функції для 'повторення чогось', так?

While

Для полегшення вам життя вигадали досить корисну конструкцію - `while`.
Записується так:

```
while(boolean) {
        // тут действие, что повторяется
}
```

Подібна конструкція виконує свій вміст у `{}`, але перед кожним виконанням дивиться в умову (aka boolean-вираз) і якщо там `true`, то зміст виконується, а якщо `false` - ні.

Наш попередній код можна виразити через `while` наступним чином:

```
fun main() {
        var shouldRun: Boolean = true
        while(shouldRun) {
                printlt("Введите число (или воспользуйтесь :q для выхода):")
                val input: String = readln()
                if(input == ":q") {
                        shouldRun = false // при наступному виконанні цикл
побачить, що умова `false`
                } else {
                        val input: Double = input.toDouble()
                        println("Результат: " + f(input).toString())
                }
        }
}
```

Ось і наш перший цикл! Але якийсь він складний, вам не здається?
Все це можна спростити скориставшись спеціальними додатковими операторами: `break` та `continue`.

Що роблять ці два оператори? Давайте розберемося.

- `break` (можна перекласти як розірвати, обірвати) – примусово закінчує цикл. Тобто навіть якщо умова буде `true` цикл все одно закінчиться.
- `continue` (перекладається як продовжити) - закінчує виконання поточного *повторення*. На відміну від `break`, `continue`, грубо кажучи, виходить з коду (код після нього не виконується) і переходить відразу до наступного повторення (до перевірки умови та подальшого повторення у разі, якщо там `true`).

Давайте перепишемо наш код:

```
fun main() {
        while(true) { // умова нам не потрібна
                printlt("Введите число (или воспользуйтесь :q для выхода):")
                val input: String = readln()
                if(input == ":q") {
                        break // виходимо з цикло
                } else {
                        val input: Double = input.toDouble()
                        println("Результат: " + f(input).toString())
                        continue // взагалі, він необов'язковий у нашому
випадку, але для наочності додамо
                        println("А меня не будет!") // IDE нам підкаже, що до
цієї ділянки коду ми ніколи не дійдемо через continue
                }
        }
}
```

Через непотрібність ми викинули змінну `shouldRun`, тому що є куди зручніший спосіб з `break`.

Do-while

Одним із підвидів циклу while є `do-while`. Крім назви, він відрізняється тим, що в **do while** спочатку виконується тіло циклу, а потім перевіряється умова продовження циклу. Через таку особливість **do while** називають циклом з *постумовою*. У свою чергу, звичайний **while** називають циклом з *передумовою*.

Записується так:

```
do {
        // щось
} while(bool)
```

У такому циклі також існує `break` та `continue`, які ніяк не відрізняються.
Однак, наше завдання можна вирішити через **do-while** і без них:

```
// створимо змінну з повідомленням, щоб потім її перевикористовувати
val numberInputMessage = "Введите число (или :q для выхода):"

// створимо окрему функцію для зручності
private fun requestInput(message: String): String {
        println(message)
        return readln()
}

fun main() {
        var input = requestInput(numberInputMessage)
        do {
                println("Результат: " + f(input.toDouble()).toString())
                input = requestInput(numberInputMessage) // записуємо
наступне введення, щоб перевірити після повторення, що було введено
        } while(input != ":q") // якщо введення не ":q" програма
продовжуватиме працювати
}
```

Ми створили для зручності функцію та змінну, що поєднувала схожий код. Знову зробили змінну поза циклом і запис її в кінці циклу (для того, щоб перевіряти після повторення введення користувача).

Це альтернативне рішення, хоч і не найкраще.

For

І тепер перейдемо до не менш важливого виду циклів - `for`.
Відмінність цього виду циклів у цьому, що він будується за умови, але на ітераторі.
Що таке ітератор? Ітератор - це вбудована утиліта в мову, яка переміщається між якоюсь сумою елементів. Тобто кожне *повторення* буде відповідати одному елементу у цій сумі.
У нашому випадку, ця сума елементів буде відповідати діапазону, а елемент — одиниці рахування цього діапазону.

Що таке діапазон? Простими словами - інтервал значень будь-якої величини.
Прикладом діапазону може бути [0; 5] (описує інтервал чисел від 0 до 5, включно).
Бувають різні види діапазонів, але поки що ми розглянемо найпростіший варіант із діапазоном цілих чисел.

Як створити такий цикл? Для початку розглянемо прогресію з цілими числами:

```
for(i in 0..5) {
        println(i)
}
```

Тут ми бачимо оператор `in`, який працює з ітератором (у нашому випадку, з тим, що його виражає - діапазоном).

Цей код надрукує наступне:

```
0
1
2
3
4
5
```

Досить очевидно працює, чи не так?

Давайте вирішимо наступне завдання:

> Відтворіть функцію ступеня для позитивних чисел.
> Еквівалентні функції Int.pow (x: Double).

```
fun pow(number: Int, times: Int): Int {
        var output = number // створюємо змінну, де зберігається помножене
значення
        for(i in 0..times) { // через діапазон вказуємо, скільки разів має
цикл повторитись
                output *= number // множимо те, що вже є, на параметр number
        }
        return output // повертаємо число в степені
}
```

Тут нам IDE підкаже, що ідентифікатор не використовується і його бажано замінити на `_`. Справа в тому, що в котлині за код-стилем прийнято, що ідентифікатори, що не використовуються, називають саме так.

Що ж до завдання, тут нескладний імперативний варіант рішення.

Давайте вирішимо ще одне завдання:

> Напишіть програму, де користувач вводить **будь-яке ціле позитивне число.** А програма підсумовує всі числа від 1 до введеного користувачем числа.
> Тобто, якщо введуть число 4, ми маємо підсумовувати такі числа: 1+2+3+4.

У цьому нам дуже допоможуть діапазони!

```
fun sum(input: Int): Int {
        var output: Int = 0 // создаём всё так же временную переменную, к
которой будем добавлять результат цикла.

        for(i in 1..input)
                output += i // можно убрать `{}` так как одна
последовательность действий

        return output
}
```

Ми створили тимчасову змінну і так само використали діапазони зі змінною `i`, що містить елемент інтервалу цього діапазону на кожну ітерацію (повторення) циклу (який і відповідає тому, що ми по суті і робимо).

І наостанок, вирішимо ще одне завдання:

> Дані натуральні числа від 1 до 50. Знайти суму з них, які діляться на 5 чи 7.

Перед розв'язанням цієї задачі, згадаємо один з арифметичних операторів - `%` (залишок від поділу).

```
fun main() {
        println(22 % 4)
        println(4 % 2)
}
```

Надрукує `2` і `0`, оскільки буде такий залишок після поділу (у першому не ділиться націло, у другому - ділиться).

Наше завдання полягає в тому, щоб знайти числа, що діляться націло на 5 та 7.
Це буде еквівалентно наступному:

```
number % 5 == 0 || number % 7 == 0
```

Ця умова нам підходитиме. Тепер же залишається тільки зробити цикл та тимчасову змінну в яку ми додаватимемо результат.

```
fun main() {
        var temp: Int = 0
        for(i in 1..50)
                if(i % 5 == 0 || i % 7 == 0)
                        temp += i
        println("Сума: " + temp)
}
```

Відповіддю у нас має вийти: `436`.

Висновок

У цій частині курсу ми розглянули чимало базових тем, які є вже досить вагомим фактором у вивченні як і котліну, так і інших мов (оскільки, зазвичай мови не сильно відрізняються в цьому плані, крім, звичайно іншого синтаксису та ідиом).

Перед тим, як перейти до вивчення наступних тем, раджу вам попрактикуватися у вирішенні деяких завдань, які пов'язані з тими, що ми розглянули цього разу.

> **№1.**
> Створіть простий калькулятор, що складатиметься з наступного введення: ціле число, дія (рядок, що може помістити +, -, /, *) і ще одне число з комою.
> Приклад введення ('>' означає новий рядок введення):
> > 10
> > -
> > 9.99
> 0.0099999999999999787

> **№2.**
> Створіть програму, що прийматиме число від 1 до 12, яке

відповідає місяцю за його порядковим номером. Надрукуйте на екран назву місяця та назву сезону.

№3.
Створіть програму, що рахуватиме суму арифметичної прогресії (суму елементів діапазону від нуля до введеної величини користувача) без циклів чи рекурсій.

№4.
У змінній min, значення якої ввів користувач, лежить число від 0 до 59. Визначте, в яку чверть години потрапляє це число (у першу, другу, третю чи четверту).
При не попаданні в якусь четвер годину вивести відповідне повідомлення.

№5
Створіть програму, яка перевірить на кратність трьох усі числа від 1 до 100.

№6
Створіть програму, що виводитиме табличку множення від 1 до 9 використовуючи цикл.
Підказка: необхідно зробити вкладений цикл (цикл у циклі).

Дуже важливо практикуватися, тому раджу вам зайнятися цим відразу ж після перегляду.