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

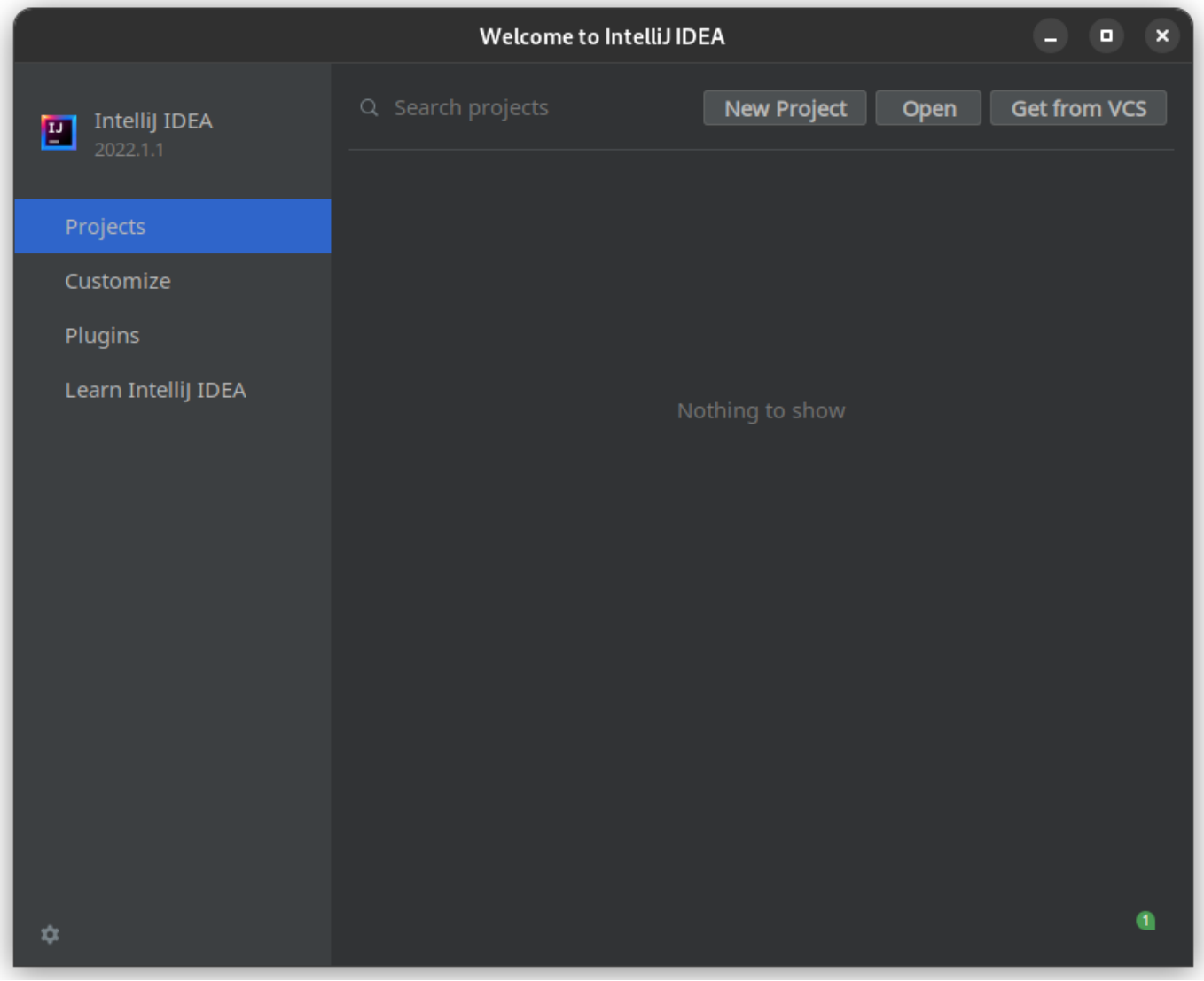
Для початку, давайте розберемось з місцем, де код будемо писати код та його запускати. Зазвичай при роботі з Котлін використовують Intellij Idea, тому давайте візьмемо її.

Як встановити

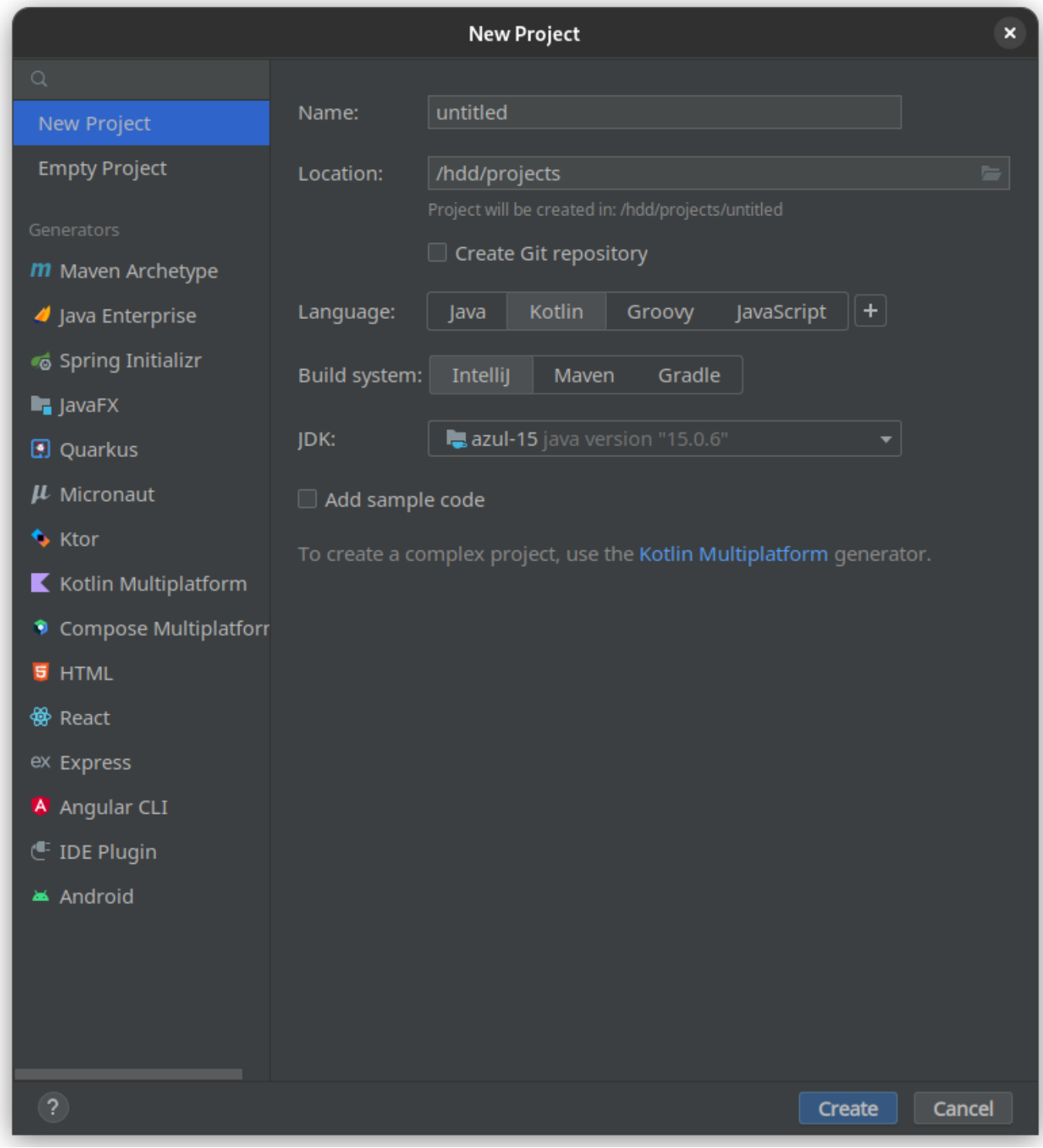
Intellij Idea делиться на два вида: **Community Edition** (бесплатная версия) и **Ultimate Edition** (платная версия).
Выбирайте то, что вам подходит. Ну и, стоит уточнить, что, если вы студент или школьник, вы можете получить бесплатно Ultimate Edition для учебных целей заполнив форму с вашими данными (студенческий).
Я буду использовать первый вариант так как принципиально они ничем не отличаются.
Скачать можно из специальной утилиты (Jetbrains Toolbox) или же скачать конкретно IDE тут.
Устанавливая ставьте удобные вам параметры темы и прочего.

Создание проекта

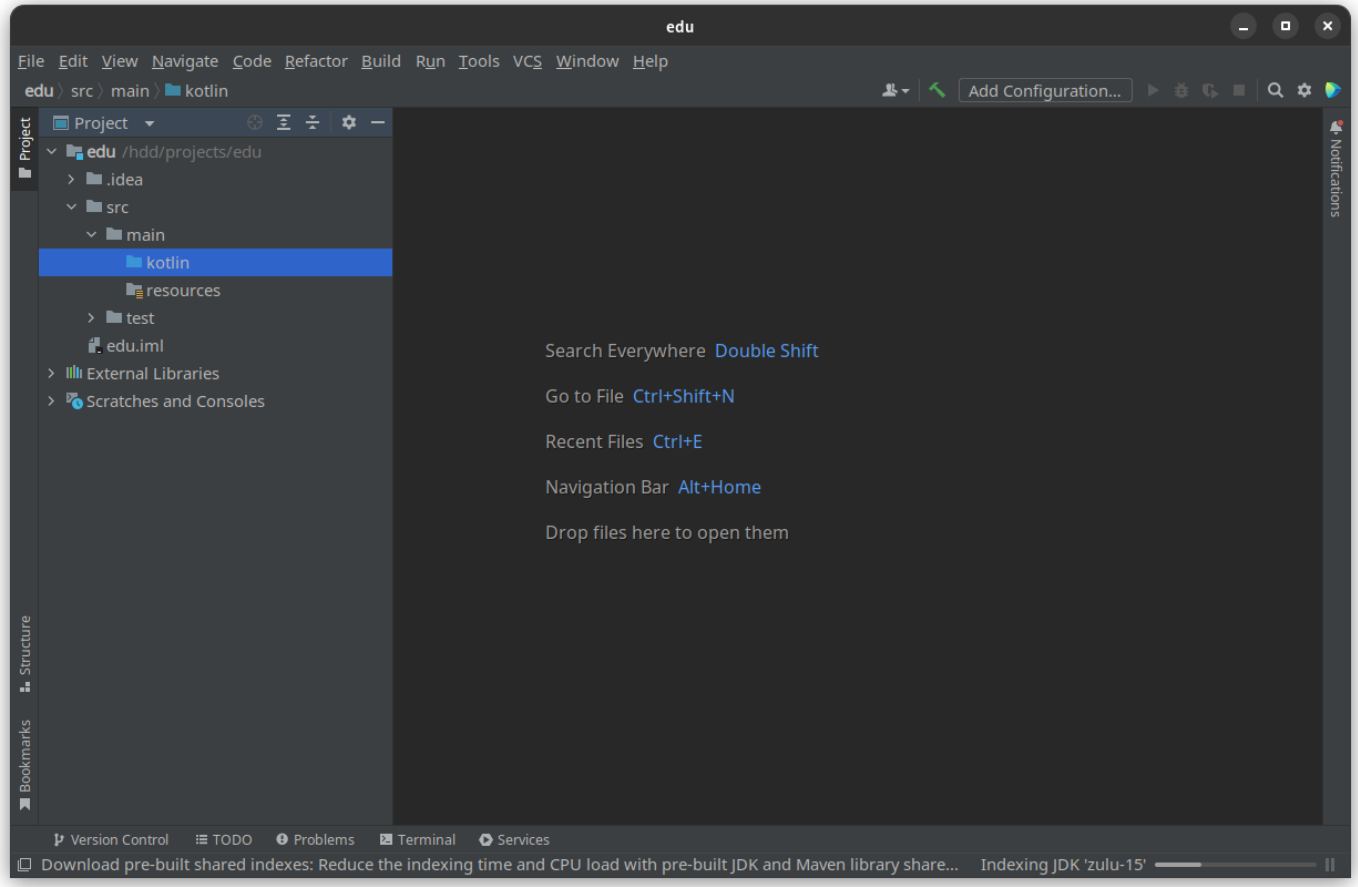
Давайте же создадим проект:



Нажимаем «New Project» для создания нашего проекта:



Называть проект можно как угодно, однако желательно называть латиницей.
Язык, естественно, Kotlin, а Build система Intellij (не будем менять, рассмотрим системы сборки потом).
Нажимаем «Create» и создаём наш проект.
В новом окне появится наш проект со структурой по умолчанию.



Пока можете создать файл «Main.kt» чтобы позже писать туда код и его проверять. Чтобы его создать нажмите правой кнопкой мыши на папку «kotlin» → «New» → «Kotlin File» и впишите «Main».

Основы Kotlin

Что ж, перейдём к основам языка программирования Kotlin.

Переменные

Не изменяя традициям — перейдём к изучению переменных и всего, что с этим связано.
Что такое переменная? **Переменная** — символ или набор символов, которые представляют из себя какую-то величину или значение.
Для чего они нужны? — для записи результатов ваших вычислений и их дальнейшего использования.
Например, вы сделали какую-то часть вычислений, записали результат в переменную и переиспользовали ваши вычисления позже. Для этого и существуют переменные!

Как создать переменную в Kotlin

Чтобы создать переменную в Kotlin, мы используем ключевое слово `var` (от англ. variable).

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

Состоит из:

- Название переменной, как название любой другой сущности — должно быть уникальным, начинаться с маленькой буквы и не иметь пробелов. Если в переменной несколько слов, следующие слова начинаются с большой (этот вид записи называют *lower camel case*).
- Тип — сущность, что описывает наши данные и будет содержаться в переменной. Например, это может быть целое число (т.е `Int`) или же число с запятой (т. е. `Double`).
  Для того чтобы присвоить какое-то значение переменной, используется знак равно `=` (и никак иначе).
  После объявления такой переменной её можно изменить следующим способом:

```
[название] = [новоеЗначение]
```

Данный тип переменной, может меняться (данный термин в программировании ещё называют *мутабельностью*, от англ. mutable — изменяемый).
Но, подождите-ка, а бывают переменные, что не меняются? Само же название кричит о том, что «я меняюсь!».

Неизменяемые переменные

Да, такие существуют. Существуют для того, чтобы записывать значения, которые не меняются (что логично).
Например, если вам нужно только разово сделать какое-то вычисление и убедиться, что вы нигде его случайно не поменяете (дабы не вызвать ошибки в работе нашей программы).
Запись ничем не отличается от изменяемой переменной, за исключением того, что для неизменяемых переменных мы используем ключевое слово `val`.

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

Подобный тип переменной стоит использовать всегда, за исключением ситуаций, где вам нужно менять значение. Это упростит код и избавит вас от некоторых проблем.

Типы данных

С этим разобрались, а какие типы данных существуют? Из встроенных существуют следующие:

- **Int** — это целое число, что имеет ограничение в размере от `-2147483647` до `2147483647` (число ограничено 32-я битами).
- **Float** — это число с плавающей точкой (или число с запятой), что имеет такое же, как и *Int*, ограничение в виде 32-битной размерности (т.е числа до `340,282,346,638,528,860,000,000,000,000,000,000,000.000000`).
- **Long** — это тот же Int, однако имеет большую размерность в два раза (до `9,223,372,036,854,775,807`).
- **Double** — это тот же Float, но опять же, большей размерности (где-то $1.7 \cdot 10^{308}$).
- **String** — простой текст. Не имеет ограничений, кроме как в количестве RAM.
Все эти типы можно записать следующим образом:

```
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"
```

Так же, существуют некоторые другие встроенные типы данных, но мы их пока рассматривать не будем.

Умозаключение

Подведём итог с переменными:

- переменные делятся на два типа: изменяемые `var` и неизменяемые `val`.
- у переменных всегда есть название, которое должно начинаться с маленькой буквы, а последующие слова — с большой. Так же, оно должно быть уникальным.
- у переменных всегда есть тип — сущность, что описывает данные или набор данных (например числа).
- у переменных всегда есть какое-то значение указанного типа (сущности).

- бывают следующие типы данных: целые числа (`Int` и `Long`), числа с запятой (`Float` и `Double`) и строки (обычный текст, `String`).

Функции

Представьте любое однотипное действие в вашей жизни, которое вы делаете каждый день, и дайте ему условное название. За пример такого действия, возьмём, например, условный бег.
Бег — однотипное действие перебирания ногами, что зависит от некоторых переменных (например, вашей физической формы, темпа или усталости).
Все подобные действия в мире программирования, называют *функцией*.
Что же такое функция? **Функция** — это фрагмент кода, который имеет уникальные имя, возвращаемый тип (aka значение или величина) и/или параметры (те самые переменные, от которых зависит однотипное действие).

- Фрагмент кода — это те самые действия, которые вы делаете, например, при том же беге. Выделения какого-то кода под функцию сделано лишь для одного: избавиться от одинакового кода.
- Параметры функции — это значения, от которых зависит ваш фрагмент кода. Например, числа над которыми будут выполняться некие вычисления в функции. Параметры не могут меняться (им нельзя присвоить другое значение).
- Возвращаемый тип — это то, что вы получаете в результате вашего действия. Например, вы получаете количество времени, когда вы бегали, и это, и есть результатом (т. е. возвращаемым типом в программировании).
  Записать функцию можно следующим образом:

```
fun [названиеФункции]([названиеАргумента]: [типАргумента]): [возвращаемыйТип]
{
        ...
}
```

Не пугайтесь тому, что здесь так много текста! На самом деле, всё очень просто:

- Ключевое слово `fun` используется для объявления того, что мы будет декларировать (создавать) новую функцию.
- Дальше у нас идёт название функции, что имеет те же правила, что и названия переменных — уникальное название без пробелов, первый символ маленький и так далее.

- Дальше открываются круглые скобки, где указываются параметры функции. Стоит учитывать то, что даже если параметры для функции не нужны — они всё равно указываются.
    - Дальше идёт название аргумента, где применяются такие же правила как и к названию функций и переменных
    - После чего, идёт обязательное указание типа (сущности) нашего аргумента.
    - Если у нас несколько аргументов, дальше идёт `,` и указание следующего аргумента.
    - Ну и закрываем скобки

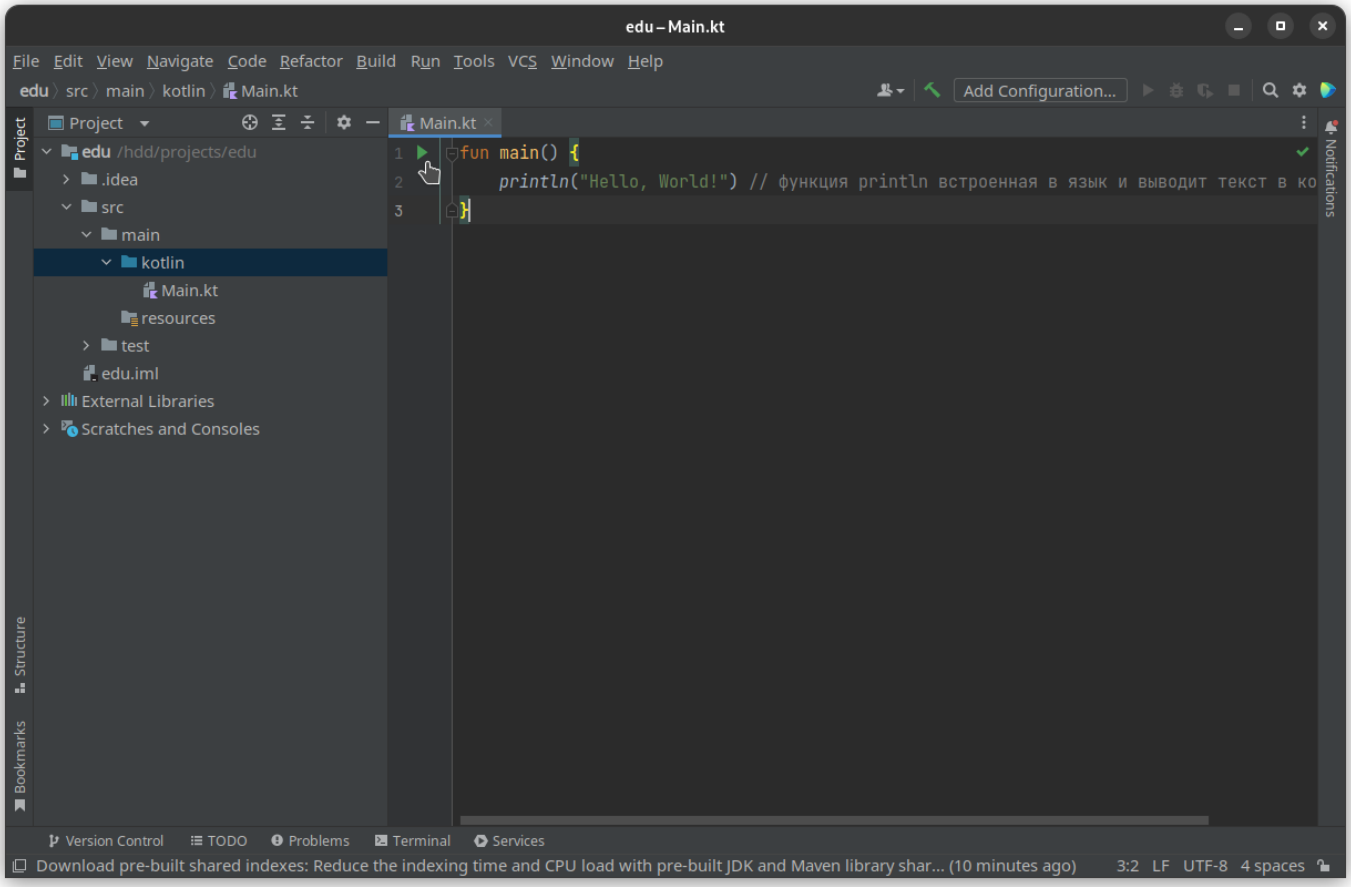
- Дальше идёт указание типа, что возвращается в функции. Но, стоит учитывать, что функции не всегда должны что-либо возвращать.
- Дальше открываются фигурные скобки (всё, что внутри, называют **телом функции**), где уже идёт наш фрагмент кода. Если нам нужно что-то вернуть из функции, используется ключевое слово `return` (например, для того чтобы вернуть целое число мы напишем `return 10`).
  Теперь, когда мы лицезрели формулы для наших программ, давайте перейдём к боевому заданию!

Пример

Чтобы закрепить наши знания, давайте попробуем что-то сделать.
В каждом туториале все изначально делают «Hello, World». Давайте не будем изменять традициям.
Для того, чтобы создать программу, логично, что нам нужно определить точку её начала. В Kotlin, точку начала объявляют создавая функцию `main()` (которая, кстати, не возвращает какого либо результата).

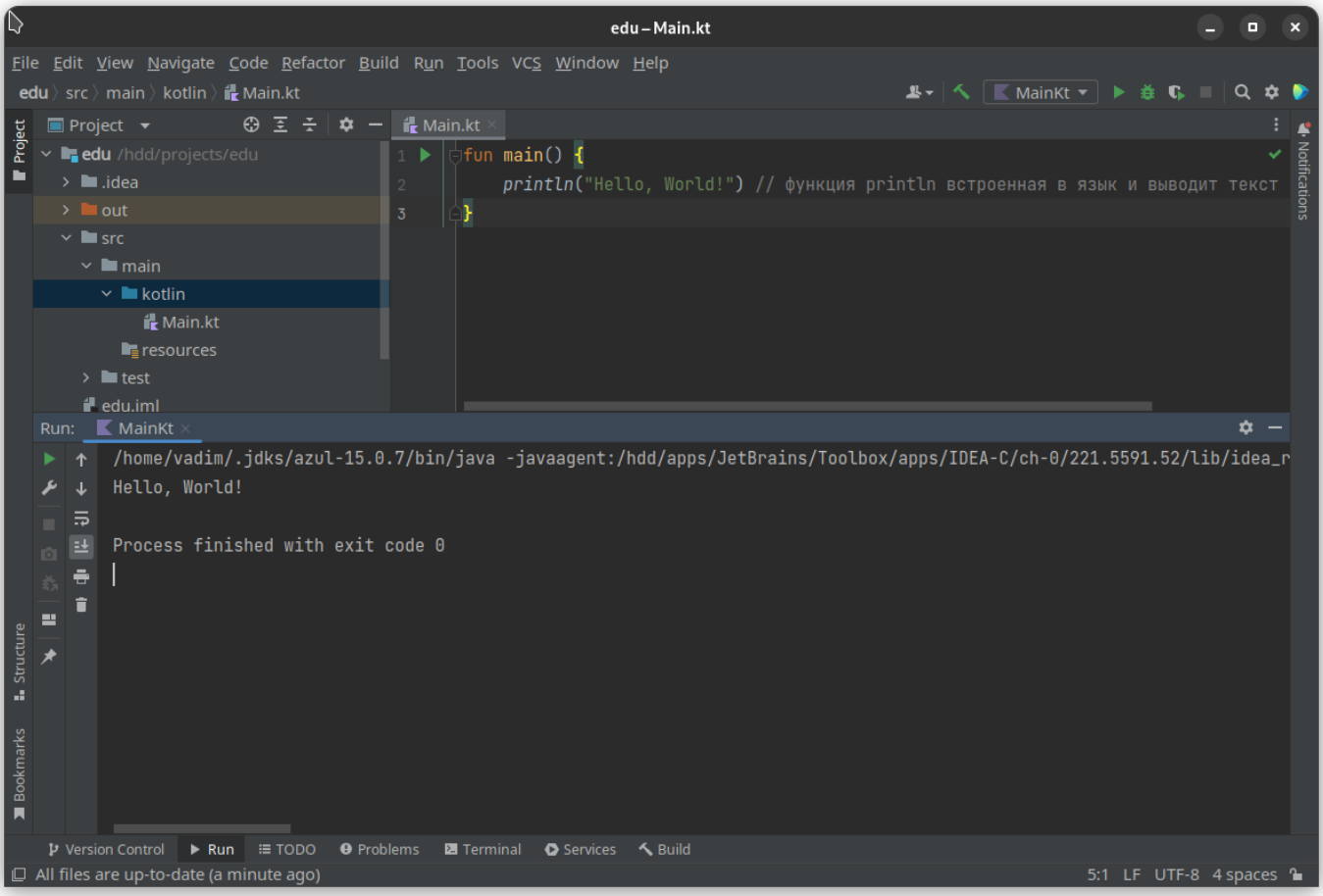
```
fun main() {
        println("Hello, World!") // функция println встроенная в язык и
выводит текст в консоль
}
```

Для того чтобы запустить нашу программу, в IDE мы нажимаем следующую кнопочку:



И в выпавшем списочке нажимаем 'Run Main.kt'.

И получаем результат:



Не очень люблю затрагивать математику при объяснении некоторых тем, но давайте всё же, попробуем решить уравнение средствами языка.
За уравнение возьмём следующее несложное в понимании уравнение:
$f(x) = x + 1$
У нас некая функция с параметром `x` (число, очевидно), которая возвращает указанное число плюс один.
Как подобная функция будет выглядеть на котлине?

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

Вроде несложно, не так ли? Главное, в начале, поглядывать на формулу того, как строится функция.
Давайте, вызовём данную функцию:

```
fun main() { // объявляем точку входа программы
        val x: Int = 10 // зададим любое число
        val result: Int = f(x) // получаем результат в переменную result
        println(result) // функция println является встроенной в Kotlin,
выводит текст в консоль. В нашем случае результат выполнения функции.
}
```

Можно упростить до следующего варианта:

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

Код, который имеет лишь одну последовательность действий, можно объявить через `=`. Возвращаемый тип будет определён автоматически котлином. В нашем же случае, это будет `Int`, т.е — целое число.

Кстати, тот же «Hello, World!» можно было написать следующим образом:

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

Но, чаще всего, код у вас будет длиннее, чем одна последовательность действий.

Кстати, это так же работает с переменными:

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

Закреплять знания нужно на практике! Для этого, попробуйте написать небольшую программу для решения следующей функции:
$f(y) = \frac{y}{2}$
Так же стоит отметить, что деление выполняется с помощью оператора `/`.
И опять незнакомый термин! Кто этот ваш оператор?

Операторы

В программировании, оператором называют символ, что представляет собой какое-то действие над сущностью (например число, которое делят на другое число). Это функция, что имеет свой специальный символ в словаре языка программирования (как например это с `fun` или `var`).
Давайте за пример возьмём то же деление. Чтобы сделать деление, мы можем использовать значок `/`.

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

По-другому, это можно было бы выразить вот так:

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

Но, для удобства и понимания, решили сокращать до `/`.
Однако, за вами остаётся возможность вызвать это словесно:

```
val result = a.div(b) // сокращённо от divide (разделить)
```

Теперь, зная это, попробуйте решить задачу, которая была выше.
$f(y) = \frac{y}{2}$
Собственно, чтобы решить задачу, нужно просто сделать следующее:

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

Не сильно отличается от сложения чисел.
Но, какие ещё существуют операторы?

Виды операторов

Операторы разделяют на такие типы:

- арифметические — сложение чисел, вычитание, умножение, деление.
  - `+` оператор сложения значений (`a + b`)
  - `-` оператор вычитания значений (`a - b`)
  - `*` оператор умножения значений (`a * b`)
  - `/` оператор деления значений (`a / b`)
  - `%` оператор деления с остатком (`22 % 4` будет соответствовать `2`)
  - `!` оператор противоположности (меняет `true` на `false` и наоборот)
  - сложенные операторы:
    - `+=` оператор сложения к существующей величине ещё одно значение.
      Эквивалентно следующему: `a = a + b`.
    - `-=` оператор вычитания значения от существующей величины.
      Эквивалентно следующему: `a = a - b`.
    - `*=` оператор умножения существующей величины на некоторое значение.
      Эквивалентно следующему: `a = a * b`.
    - `/=` оператор деления существующей величины на некоторое значение.
      Эквивалентно следующему: `a = a / b`.
    - `!=` оператор 'не равно', что соответствует `!(n == 1)`
- логические — сравнение значений
  - `>` оператор больше чем (`a > b`)
  - `<` оператор меньше чем (`a < b`)
  - `&&` оператор 'и' (`a > b && b < c`: `a` больше `b` и `b` меньше `c`)
  - `||` оператор 'или' (`a > b || c < a`: `a` больше `b` или же `c` меньше `a`)
  - `==` оператор равенства двух значений (`a == 5`)
  - `!` оператор нет (если у нас false, превращается в true и наоборот)
  - сложенные операторы:
    - `>=` оператор больше или равно (`a >= 5`)
    - `<=` оператор меньше или равно (`a <= 5`)

- и условные, которые работают с логическими операторами:
  - `if` оператор 'если это правда (true), то что-то'. Работает в кооперации с `else`: 'если это правда, то это, но если нет, то (фрагмент кода с else)'.
  - `when` оператор для нескольких 'если это правда, то ..'

Логические операторы

Давайте затронем логические операторы, что в языке возвращают тип `Boolean`, где есть только два возможных значения (варианта): `true` (истина) и `false` (ложь).

Для уточнения работы некоторых операторов, решим несколько очень простых задач:

№1.

> Создайте функцию, которая будет говорить больше ли x, чем y.
> x, y — целые числа.

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

№2.

> Создайте функцию, что будет проверять не является ли число нулём.

Для этого нам понадобиться оператор 'не равно' (`!=`).

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

№3.

> Создайте функцию, что проверяет, делится ли число нацело на 3.

Чтобы решить эту задачу нам нужно будет воспользоваться оператором остатка от деления (т.е `%`).

Если число будет делится нацело на другое число, логично, что остатком от деления будет ничего, т.е `0`.

Это значит, что нам нужно скомпонировать два оператора — остатка от деления и равенства. Т.е нужно сделать следующее `n % 3 == 0` (`n % 3` выполняется перед так как выражения читаются справа на лево).

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

Условные операторы

Давайте больше уделим внимания условным операторам.
Для того чтобы сделать программу, что опирается на какие-то условия, которые нужно обработать (например, как тот случай, что мы оговорили выше), используют *условные операторы*.
Само название говорит нам о том, что у нас есть некое условие. Давайте же разберём какие виды условных операторов есть в Kotlin.

If else

Одним из условных операторов является `if-else`.
Очень простая конструкция, что обозначает «если это истина, то сделай *это*, если же нет, то *это*.»
Записывается вот так:

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

Но, что же делать, если у нас несколько условий? Например, нам нужно узнать наибольшее число из трёх произвольных.
Логически, можно прийти к тому, что в `else { ... }` можно дописать ещё один `if`. И это будет верно! Это сработает.

```
fun getBiggest(a: Int, b: Int, c: Int) {
        if(a > b && a > c) { // тут, кстати, используется логический оператор
'и'
                return a
```

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

Но, код заметно, стал куда сложнее. Может можно как-то упростить?
Да, действительно — данную конструкцию можно упростить.
Для if, как и для else применяется одно упрощение:

- Если у вас только одна цепочка действий, то указывать фигурные скобки необязательно.
  То есть, в итоге получится следующее:

```
if(a > b && a > c) // тут, кстати, используется логический оператор 'и'
        return a
else if(b > a && b > c)
        return b
else return c
```

Теперь выглядит лучше, не так ли? А что, если я вам скажу, что можно ещё проще?

```
return if(a > b && a > c) { // тут, кстати, используется логический оператор
'и'
        a
} else if(b > a && b > c)
        b
else c
```

Воу-воу, что это такое? Если вы ранее изучали другие языки программирования, возможно, знаете о «тернарном операторе». Что ж, в Kotlin решили сделать возможность использовать условный оператор if-else (и забегая наперёд, так же оператор `when`) в качестве *выражения* (всё, что может выражать значение: сырое указание как `10`, функция, что возвращает какое-то значение и прочее, что в результате возвращает нам какое-то значение, называют *выражением*).

Что же это значит? Это значит, что фрагменты кода в if будут выступать в качестве условных функций, которые возвращают какое-то значение (число или что-либо

другое) из каждой обработанной ветки условия. Для того чтобы вернуть что-либо из фрагмента кода, нужно написать значение (или переменную/функцию, которая будет иметь нужное нам значение) последним в нашем фрагменте кода.

Давайте закрепим материал сделав несложное математическое уравнение:

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

Тут у нас только два условия, что делает эту задачу не сложной:

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

Хорошо, с этим разобрались. Но, что делать, если у нас добавиться больше условий? Делать бесконечные цепочки `if-else`? Как бы не так.

When

Для множественной выборки условий был создан оператор `when`.
Чтобы понять, для каких случаев он используется, давайте решим следующую задачу:

> Создайте функцию, что будет возвращать день недели по его порядковому номеру.
> Т.е: если в функцию ввести параметр '1', то функция вернёт «Понедельник». И так далее.

Для этого, мы можем использовать оператор `when`, который будет куда понятней бесконечных цепочек `if-else`.

```
fun getDay(ordinal: Int): String {
        return when {
                ordinal == 1 -> "Понедельник"
                ordinal == 2 -> "Вторник"
                ordinal == 3 -> "Среда"
                ordinal == 4 -> "Четверг"
                ordinal == 5 -> "Пятница"
                ordinal == 6 -> "Суббота"
                ordinal == 7 -> "Воскресенье"
```

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

Стоит отметить `else`, который так же существует в операторе `when`. Работает аналогично, обрабатывая условие, которые не было удовлетворено до этого.
В нашем случае, если ввести в функцию число больше 7, то нам вернётся сообщение о том, что указан неверный день.
Кстати, как-то однотипно выглядят условия, не так ли? «Как-то слишком много мороки» — сказал Kotlin, и сделал очередное упрощение:

```
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"
        }
}
```

Как всё понятней и очевидней стало, не так ли?
Для закрепления материала, решим следующее математическое уравнение:

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

В этом уравнении у нас целых 4 условия. Для этого, подходит как раз 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
        }
}
```

Так же, вы можете сами объявлять свои варианты операторов, но об этом мы поговорим в другой раз.

Встроенные функции в Kotlin

Что ж, мы рассмотрели базис, на котором уже может строиться программа.
Ранее, мы уже рассматривали одну из встроенных функций — `println`. Данная функция печатала в консоль текст, который мы ей задавали. Давайте рассмотрим и другие.
Перейдём к задаче: нам нужно написать программу, что будем перемножать введённые пользователем цифры. Довольно просто.
Для этого нам понадобиться функция `readln`, что читает пользовательский ввод с консоли.
Т.е, у нас будет что-то типа:

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

Вот и вся программа! Но, попытавшись её запустить, мы получим ошибку:

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

Переведя на русский язык мы получим следующее:
`Несоответствие типов: предполагаемый тип — String, но ожидался Int.`
Что же это значит? Это значит, что нашим переменным `number1` и `number2` типа `Int` (целые числа) присваивается несоответствующий тип `String` (строка, aka обычный сырой текст).
Увы, так как ввод пользователя может быть любым: от цифр, до букв и других символов, функция `readln` возвращает тип `String` (строку, что может содержать не только цифры). Чтобы это исправить, мы должны преобразовать строку в целое число (ну правда, желательно, перед этим проверить ввод пользователя).
В Kotlin есть функция `toInt` для строки, что очень упрощает нам жизнь. Она конвертирует строку в цифру путём её парсинга.

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

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

Подобные функции так же существуют и для других типов-цифр:

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

- `String.toLong()`
  Так же, подобные функции существуют и между числами. Например, целое число может превращаться в число с плавающей точкой (aka с запятой, типа `Double`).
  Для чего это нужно? Ну например, вам нужно сложить одно целое число и одно число с плавающей точкой. Так как Kotlin — это строго-типизированный язык, просто так сложить цифры разных типов не получится. В зависимости от того, что вам нужно, вы можете привести число с плавающей точкой в целое и наоборот.

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

Вообще, если вы попробуете сложить `x + y` без преобразований, Kotlin автоматически будет выводить тип `Double`. Но, под капотом, делается то, что мы только что рассмотрели. Но подобное работает только с числами.

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

Кстати, функция `println` так же выводит строку в консоль. А мы ей отдаём число. Почему для неё таких ограничений нет?
Всё просто: функция `println` принимает «Any» (т.е любой тип данных) и неявно вызывает на этом типе `toString()` (такая же встроенная функция в язык, что

применяется на любой созданный объект), что возвращает строку с нашей цифрой. Внутри там что-то подобное:

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

Какие ещё интересные функции существуют?
Давайте, решим следующую задачу, заодно вспомним про условные операторы:

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

И тут мы видим степень! Что же, зная, что такое степень, несложно решить данную задачу.
Я предпочитаю для более чем двух веток с условиями, использовать `when`.

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

И наша задача решена! Но, раз мы говорим о встроенных функциях, то наверное есть какое-то упрощение?
Да, есть. Это функция `pow`.

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

Очень даже удобно! Особенно, если у нас произвольная степень.
Давайте же выведем результат в консоль:

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

Вроде не сложно, не так ли? Заодно закрепили другие встроенные функции. Подобных функций довольно много, так что мы будем их рассматривать во время прохождения других тем.

Области видимости

Теперь перейдём к довольно интересной, но немножко сложной теме — области видимости.
Ранее мы рассматривали переменные и функции, так что теперь стоит рассмотреть случаи, когда функция или переменная может быть недоступна или наоборот доступна в некоторых местах.
Область видимости (англ. scope) в **программировании** — важная концепция, определяющая доступность переменных, функций и других сущностей. Данная концепция разделяет переменные, функции и пр. на глобальные и локальные.
Давайте рассмотрим на примере:

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

Данный код выведет следующую ошибку:

```
Unresolved reference: a
```

Что означает, что переменная созданная в функции `foo()` недоступна в функции `main()`.
Почему? В этом конкретном случае, `a` переменной не может существовать и в теории, так как она создаётся при вызове функции `foo`, а она у нас не вызывается.
А что если мы её вызовём?

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

Теперь она создана и, по-идее, программа должна работать, но как бы не так, мы получим всё ту же ошибку:

```
Unresolved reference: a
```

Дело в том, переменные видны только в месте создания и ниже по иерархии.
Ниже по иерархии? Давайте, перепишем наш код так чтобы переменная была видна:

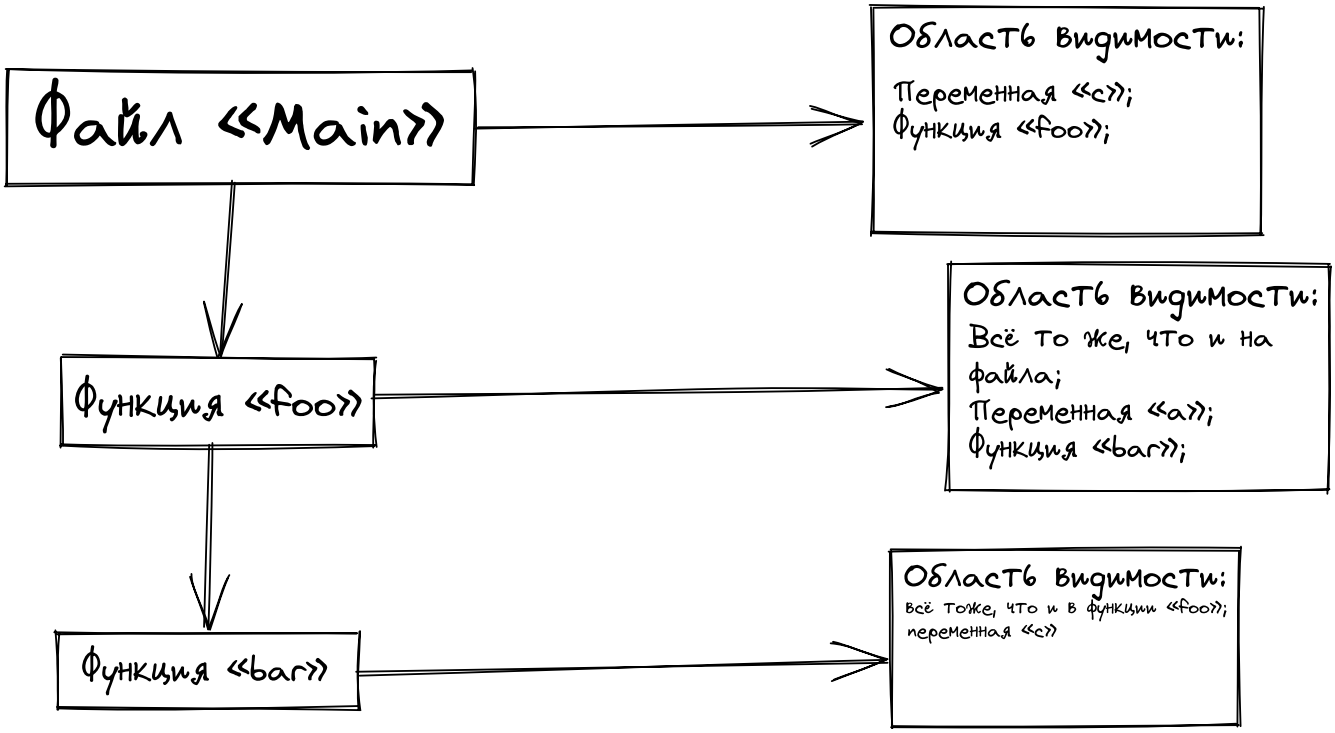
```
var a = 0
fun foo() {
        a = 1
}
fun main() {
        foo()
        println(a + 1)
}
```

Для наглядности изменим переменную при вызове `foo()`.
Подобное и означает «ниже по иерархии».
Функция, что использует переменную `a` наследует область видимости (aka scope) файла, в котором мы её создали.
И так работает с любым местом, где переменную создают. Даже в функции:

```
var c = 2
fun foo() {
        var a = 2 // создаём переменную на уровне функции.
        c = 4 // переменная на уровне файла видна в функции `foo`
        fun bar() {
                val b = a.pow(2) // переменная `a` видна в функции `bar` из-
за наследования области видимости
                a = b
        }
```

```
        bar() // функция доступна в месте её создания (декларирования)
        println(a) // переменная доступна в месте её создания
}
```

Да, да, в котлине можно даже создавать функции в функциях, не удивляйтесь! Хотя сейчас не об этом.
В этом случае, функция `bar` наследует область видимости файла, функции `foo` и так может продолжаться до бесконечности. Вообще, фигурные скобки `{}` можно рассматривать как оператор, который создаёт новую область видимости.
Давайте визуализируем то, как строится наша область видимости:



Т.е: каждая новая область видимости наследует «родителя», в котором она создаётся.
Родитель и всё, что выше по иерархии, не видит что-то созданное ниже по иерархии.
P.S: Вообще подобные переменные, что создаются в функциях, называют — **локальной переменной**.
А переменная, которую мы создавали вне функции — **глобальной**. Она видна везде, начиная с того же файла, заканчивая другими.
Заканчивая другими? Так же, как мы создавали котлин-файл под именем «Main», мы можем создать любой другой файл. Как минимум, для того чтобы не держать весь код в одном файле. Это упростит навигацию по коду в проектах немножко сложнее тех, которые мы делали ранее.
А что будет если создать ещё один файл, в котором мы создадим такие функции и переменные? Что ж, давайте проверим:

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

Перейдя в файл «Main» и попробовав вызвать эти функции нас ждёт успех:

```kotlin
var c = 2
fun foo() {
        var a = 2
        c = 4
        fun bar() {
                val b = a.pow(2)
                a = b
        }
        bar()
        someFunction() // вызываем функцию с файла `another.kt`
        println(a + abc) // получаем переменную с файла `another.kt`
}
```

Что же это значит? А это значит то, что у файла так же, как и например у функции, есть дочерний scope (область видимости) и это некоторые другие файлы.
Некоторые другие файлы? Не все?
Дело в том, что файлы идентифицируются не только по их названию, но и по их **пакету**.
Пакет? Логически будет предположить, что никто не имел ввиду полиэтиленовый или какой либо другой пакет, а какой-то уникальный идентификатор.
Что за уникальный идентификатор и зачем он?
Всё для того же, для чего и создаются другие файлы: для удобства. Нужно же разделять и сортировать написанный код.
Из реальных примеров, вы можете взять системные папки по-типу Music, Videos, Images и прочие, что содержат информацию только определённой категории.
В Котлине подобная же система категоризации кода, единственное, что отличается, — это термин (**пакет**).
Собственно, так же как и с системными папками, мы можем делать структуру нашего проекта разделяя на какие-то осмысленные части.
Например, для всяких математический вычислений мы можем создать такой пакет: `math.calculations`.
В файловой структуре мы просто создаём соответствующие частям пакета (разделены точкой) папки:

Т.е папку `math`, а в ней ещё одну папку `calculations`. После чего, можно уже создавать наши файлы с кодом.
Для примера создадим файл с функцией, которая будет решать следующее уравнение:

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

```
// файл Function.kt
package math.calculations // автоматически добавилось нашей IDE
(идентификатор нашего файла)
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
        }
}
```

Как вы уже заметили, сверху у нас добавилась строка кода с местом нашего файла. Он обязателен, даже если вы поместили его в соответствующую папку. Это потому, что Kotlin допускает указание пакета свободно (т.е, вы можете не создавать файловую структуру, что будет соответствовать пакету).
Это делается в несложных проектах, где 8-10 файлов и проблем с навигацией нет, но я вам рекомендую всегда создавать соответствующую файловую структуру.
Что ж, перейдём к вызову нашей функции:

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

По-идее так, но вызвав мы получим следующую ошибку:

```
Unresolved reference: foo
```

Дело в том, что по-умолчанию, область видимости ограничивается текущим пакетом (в нашем случае хоть он и отсутствует, но он всё такой же идентификатор,

даже если он и пустой).
Для того чтобы получить что-то из другой области видимости (aka пакета) нужно для начала «импортировать» идентификатор.
«Импорт» делают с помощью ключевого слова `import`. Он всегда должен указываться сверху, сразу после пакета (ну или при его отсутствии, просто сверху).
Схема импорта такая:

```
import [пакет].[идентификатор]
```

Т.е, чтобы вызвать функцию `f(x: Double)`, нам нужно сделать следующее:

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

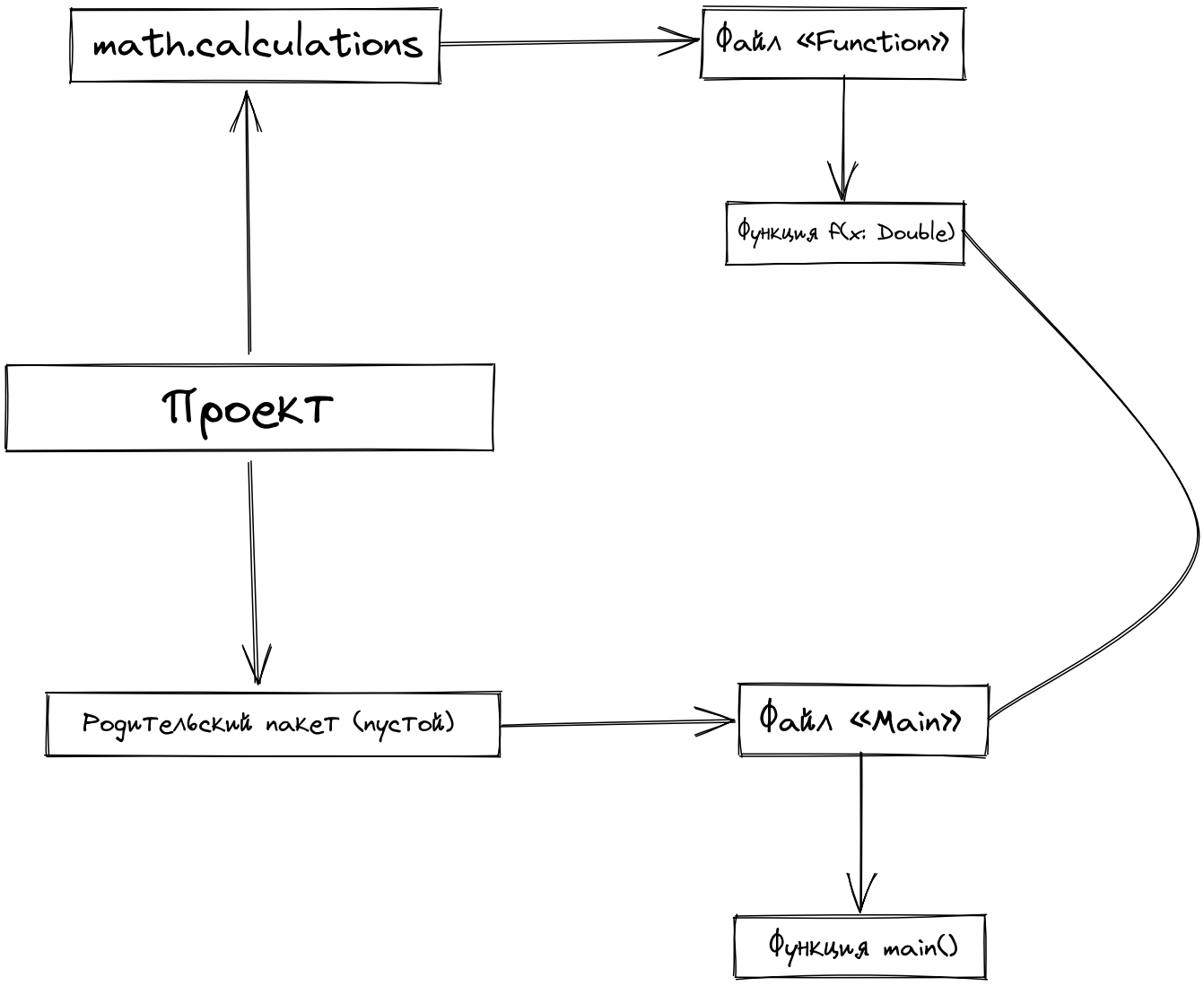
И у нас всё запуститься успешно!
Но, если без импортирования идентификаторы других пакетов не видны, то можно создавать дубликаты названий?
Да, вы можете создавать дубликаты имён за исключением ситуаций, когда вы пытаетесь создать одинаковый идентификатор в одном конкретном скоупе (области видимости).
Т.е, следующее запрещено:

```
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()
}
```

Однако, разрешено следующее:

```
val a = 1
fun main() {
        println(a)
        val a = 2
        println(a)
}
```

Дело в том, что приоритетным пространством имён (с нашими идентификаторами) является текущая область видимости (aka скоуп).
Это всё потому, что как таковая функция (или любое другое место) — новый независимый скоуп (область видимости). Мы не можем быть уверены, что рано или поздно мы не импортируем какую-то переменную или же не объявим такую же в этом файле. А выдумывать бесконечно новые имена не сделает код проще, а только усложнит его.
Кстати, стоит отметить, что создания дубликатов в одном пакете — невозможны. Дело в том, что со стороны котлина файл не независимая структурная единица и она существует только в исходном коде.
Вспомните пример с функцией в пакете `math.calculations`, мы же не указываем конкретный файл при вызове функции или её импорте? Потому дубликаты в одном пакете и невозможны, так как определить конкретный идентификатор с конкретного файла невозможно.
Что ж, для закрепления, давайте визуализируем всё то, что мы обговорили выше:



У нас есть проект с двумя уникальными пакетами: `math.calculations` и родительским (ну или пустым). Файл «Main» завязан на функцию `f(x: Double)` в пакете `math.calculations` (мы это выделили линией для визуализации).
Что ж, подведём промежуточный итог:

- Программа делится на разные области видимости (скоупы), которые имеют четкую иерархию в зависимости от того, где и что вы создаёте.
- Иерархия обычно следующая: область видимости на уровне пакета → область видимости на уровне декларации (функции, например) → и так далее (например вложенные функции).
- Родительской областью видимости является пакет, в котором существует наш идентификатор (функция, переменная). Идентификаторы с других пакетов не видны по-умолчанию.
- При необходимости, можно расширить пространство имён (идентификаторы, что видны в другой области видимости) с помощью импорта (`import [пакет].[идентификатор]`).

Модификаторы видимости

Кстати, говоря о том, что файл — это не независимая структура, я немножко соврал и сейчас объясню почему.
Давайте решим следующий пример:

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

Функция $a(x)$ у нас такая:

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

На Kotlin нам нужно написать следующее (в файле 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
}
```

И теперь вызовём это в Main:

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

И на этом наша программа, условно, закончена.
Посмотрев на функцию `a(x: Double)` мы можем подумать о том, что она используется только в функции `f(x: Double)` и в принципе, она нигде кроме в файле 'Function.kt' не нужна.
Можно эту функцию просто игнорировать в подсказках и не импортировать, однако, если таких функций много? Это очевидно, захламляет глобальное пространство имён, даже если оно не импортировано.
На выручку к нам приходят модификаторы видимости! **Модификаторы видимости** — ключевые слова, что описывают то, где виден идентификатор.
Для нашего случая, существует модификатор `private`. Он указывает, что переменная видна только там, где её создали и ниже по иерархии.
На самом деле, формула создания той же функции выглядит так:

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

По-умолчанию, ко всем декларациям (функциям, переменным и прочему) неявно применяется модификатор `public` (т.е публичный, видимый изнутри).
`fun main()` → `public fun main()`.
В нашем же случае, мы делаем следующее:

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

Кстати, в этом случае, при указании одинаковых идентификаторов в одном пакете, но разных файлах допускается, так как конфликт попросту невозможен.
С переменной будет так же:

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

Вывод

Изначальное рассмотрение уникальности имён, создание переменных и функций оказалось не таким простым, как вы поняли.
Как я уже упоминал ранее, идентификатор функции строится на следующих его свойствах — это имя и параметры.
С учетом рассмотренных тем: *область видимости* и *модификаторов видимости*, мы их так же добавляем в уникальность идентификатора (обычно это называют сигнатурой).
Ну и тоже самое мы делаем с переменной.
Итоговым вариантом идентификаторов у нас будут:

- Функция — модификатор видимости + область видимости + имя + набор параметров (различие в их количестве или типе).
- Переменная — модификатор видимости + область видимости + имя.

Желательно самому поиграться с этим для большего понимания!

Циклы и Рекурсии

Теперь же, перейдём к так же довольно интересной, но немножко непростой теме — циклы.
Чтобы больше понять, что такое циклы давайте создадим какую-то задачу.
Например, возьмём задачу, которую мы решали в прошлой теме. Для того, чтобы решить уравнение с использованием ввода, мы каждый раз запускаем нашу программу. А что, если сделать в нашей программе бесконечный ввод чтобы каждый раз не перезапускать нашу программу?

Вообще, без нашей темы циклов, это вполне можно было решить следующим образом:

```
fun main() {
        printlt("Введите число:")
        val input: Double = readln().toDouble()
        println("Результат: " + input.toString())
        return main() // в конце функции просто вызываем её ещё раз
}
```

И вот, решение найдено!

Подобное называют **рекурсией**. Простыми словами — это понятие объявления (написания, описания) кода функции через саму себя. Это как матрёшка, но, которая в нашем случае не имеет конца.

Что ж, а как теперь завершить нашу программу? Можно конечно это сделать закрыть принудительно процесс программы через инструменты системы или IDE, но давайте будем людьми и сделаем какой-то механизм выхода.

Чтобы сильно не заморачиваться, давайте введём условие, что для выхода из программы нам нужно написать «:q».

```
fun main() {
        printlt("Введите число (или воспользуйтесь :q для выхода):")
        val input: String = readln() // создаём переменную с текстом так как
нам нужно проверять ввод пользователя
        if(input != ":q") {
                val input: Double = input.toDouble() // сила областей
видимости!
                println("Результат: " + f(input).toString())
                main()
        }
}
```

Это всё так же останётся рекурсией, только уже не бесконечной (у нас появилось условие).

Что ж, рассмотрев довольно простой пример рекурсии, к которому можно было прийти самому при попытке решить задачу с перезапуском нашего решения уравнений.

Что же тогда такое циклы? **Циклы** — это средства языка, которые воссоздают рекурсию. Их так же относят к операторам, называя *циклическими* операторами.

Так что, теперь, давайте рассмотрим как это можно решить другими средствами языка. Не всегда же вы будете создавать отдельно функции для 'повторения чего-то', да?

While

Для облегчения вам жизни придумали довольно полезную конструкцию — `while`. Записывается следующим образом:

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

Подобная конструкция выполняет своё содержимое в `{}`, но перед каждым выполнением смотрит в условие (aka boolean-выражение) и если там `true`, то содержание выполняется, а если `false` — нет.

Наш предыдущий код можно выразить через `while` следующим образом:

```
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 следующем выполнении цикл
увидит, что в условии `false`
                } else {
                        val input: Double = input.toDouble()
                        println("Результат: " + f(input).toString())
                }
        }
}
```

Вот и наш первый цикл! Но, какой-то он сложный, вам не кажется?
Всё это можно упростить воспользовавшись специальными дополнительными операторами: `break` и `continue`.

Что делают эти два оператора? Давайте разберёмся.

- `break` (можно перевести как разорвать, оборвать) — принудительно заканчивает цикл. Т.е даже если условие будет `true` цикл всё равно закончится.
- `continue` (переводится как продолжить) — заканчивает выполнение текущего *повторения*. В отличии от `break`, `continue`, грубо говоря, выходит из кода (код после него не выполняется) и переходит сразу к следующему повторению (к проверке условия и дальнейшего повторения в случае, если там `true`).

Давайте перепишем наш код:

```
fun main() {
        while(true) { // можно забить на условие
                printlt("Введите число (или воспользуйтесь :q для выхода):")
                val input: String = readln()
                if(input == ":q") {
                        break // выходим из цикла
                } else {
```

```
                        val input: Double = input.toDouble()
                        println("Результат: " + f(input).toString())
                        continue // вообще, он необязателен в нашем случае,
но для наглядности добавим
                        println("А меня не будет!") // IDE нам подскажет, что
к этому участку кода мы никогда не дойдём из-за continue
                }
        }
}
```

Из-за ненадобности мы выкидывает переменную `shouldRun`, так как есть куда удобней способ с `break`.

Do-while

Одним из подвидов цикла while является `do-while`. Кроме названия, он отличается тем, что в **do while** сначала выполняется тело цикла, а затем проверяется условие продолжения цикла. Из-за такой особенности **do while** называют циклом с *постусловием*. В свою же очередь, обычный **while** называют циклом с *предусловием*.

Записывается следующим образом:

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

В подобном цикле так же существует `break` и `continue`, которые никак не отличаются. Однако, нашу задачу можно решить через **do-while** и без них следующим образом:

```
// создадим переменную с сообщением чтобы потом её переиспользовать
val numberInputMessage = "Введите число (или :q для выхода):"

// создадим отдельную функцию для удобства
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) // записываем
следующий ввод чтобы проверить после повторения, что было введено
        } while(input != ":q") // если ввод не ":q" программа будет
продолжать работать
}
```

Мы создали для удобства функцию и переменную, что объединяла похожий код. Так же сделали переменную вне цикла и запись в неё в конце цикла (для того чтобы проверять после повторения ввод пользователя).

Это альтернативное решение, хоть и не совсем хорошее.

For

И теперь перейдём к не менее важному виду циклов — `for`.
Отличие этого вида циклов в том, что он не строится на условии, а на итераторе.
Что такое итератор? Итератор — это встроенная утилита в язык, которая перемещается между какой-то суммой элементов. Т.е каждое *повторение* будет соответствовать одному элементу в этой сумме.
В нашем случае, эта сумма элементов будет соответствовать диапазону, а элемент — единице исчисления этого диапазона.

Что такое диапазон? Простыми словами — интервал значений какой-либо величины. Примером диапазона может быть [0; 5] (описывает интервал чисел от 0 до 5, включительно).
Бывают разные виды диапазонов, но пока мы рассмотрим самый простой вариант с диапазоном целых чисел.

Как создать подобный цикл? Для начала рассмотрим прогрессию с целыми числами:

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

Тут мы лицезреем оператор `in`, который работает с итератором (в нашем случае, с тем, что его выражает — диапазоном).

Данный код выведет следующее:

```
0
1
```

```
2
3
4
5
```

Довольно очевидно работает, не так ли?

Давайте решим следующую задачу:

> Воссоздайте функцию степеня для положительных чисел.
> Эквивалентно функции Int.pow(x: Double).

```
fun pow(number: Int, times: Int): Int {
        var output = number // создаём переменную, где будет хранится
умноженное значение
        for(i in 0..times) { // через диапазон указываем, сколько раз
повториться
                output *= number // умножаем то, что уже есть на параметр
number
        }
        return output // возвращаем число в степени
}
```

Тут нам IDE подскажет, что идентификатор `i` не используется и его желательно заменить на `_`. Дело в том, что в котлине по код-стилю принято, что идентификаторы, что не используются называют именно так.

Что же относительно задачи, тут несложный императивный вариант решения.

Давайте решим ещё одну задачу:

> Напишите программу, где пользователь вводит **любое целое положительное число.** А программа суммирует все числа от 1 до введенного пользователем числа.
> Т.е, если введут число 4, мы должны суммировать следующие числа: 1 + 2 + 3 + 4.

В этом нам как раз очень помогут диапазоны!

```
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
}
```

Мы создали всё так же временную переменную и всё так же использовали диапазоны с переменной `i`, что содержит элемент интервала этого диапазона на каждую итерацию (повторение) цикла (который и соответствует тому, что мы по-сути и делаем).

И на последок, решим ещё одну задачу:

> Даны натуральные числа от 1 до 50. Найти сумму тех из них, которые делятся на 5 или на 7.

Перед решением данной задачи, вспомним один из арифметических операторов — `%` (остаток от деления).

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

Выведет `2` и `0`, так как будет такой остаток после деления (в первом не делится нацело, во втором — делится).

Наша задача состоит в том, чтобы найти числа, что делятся нацело на 5 и 7.
Это будет эквивалентно следующему:

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

Это условие будет нам подходить. Теперь же, остаётся только сделать цикл, временнную переменную в которую мы будем добавлять результат.

```
fun main() {
        var temp: Int = 0
```

```
        for(i in 1..50)
                if(i % 5 == 0 || i % 7 == 0)
                        temp += i
        println("Сумма: " + temp)
}
```

Ответом у нас должно получится: `436`.

Заключение

В данной часте курса мы рассмотрели довольно много базовых тем, которые являются уже довольно весомым фактором в изучении как и котлина, так и других языков (так как, обычно языки не сильно различаются в этом плане, кроме, естественно другого синтаксиса и идиом).

Перед тем, как перейти к изучении следующих тем, советую вам попрактиковаться в решении некоторых задач, которые связаны с темами, что мы рассмотрели в этот раз.

> TODO задачи

Очень важно практиковаться, так что советую вам занятся этим сразу же после просмотра.