**Дневник изучения ЯП Kotlin.**

Введение

Язык kotlin является компилируемым языком, я бы назвал его более совершенной версией java. Он собой замещает java в той же мобильной разработке: в 2017 году именно kotlin занял место java в мобильной разработке, теперь именно kotlin является основным языком мобильной разработки.

Я пытался сделать мобильное приложение на kotlin, но безуспешно. Я столкнулся с тем, что сама по себе Android разработка мне была на тот момент не по зубам, да и знаний в kotlin явно не хватало, я его знал на околонулевом уровне. Поэтому я решил погрузиться в изучение kotlin, записав свои знания и впечатления здесь.

**Важным** считаю сказать, что здесь я отображаю свои впечатления, то, что меня удивило или заинтересовало. То, что мне не очень интересно, я упоминаю вскользь или не упоминаю совсем. Если я не буду вкладывать свое мнение, выделять то, что мне понравилось, то чем эта работа будет отличаться от генерации нейросетью или копии статьи из интернета?

Переменные и их типы

Начнем с переменных. Переменные бывают разных типов, в kotlin статическая типизация, то есть просто так тип переменной после инициализации нельзя изменять.

Создадим 3 переменные:

val myInt = 15  
var myString = "Hello"  
var myNewString : String? = "world"

Теперь обсудим каждую строку.

1. Первая строка содержит в себе инициализацию целочисленной переменной. Тип её явно не указан, компилятор сам определяет его, для определения других числовых типов могут использоваться литералы. Для инициализации мы используем ключевое слово **val**, оно обозначает, что значение переменной нельзя изменять в будущем.
2. Во второй строке мы делаем то же самое, что и в первой, но тип будет String, а переменная будет изменяемой в силу ключевого слова **var**.
3. В третьей строке мы четко указываем тип переменной. Как я понял так правильно, да и после С++ не хотелось бы смотреть на переменные без явного указания типа при инициализации. Обратим внимание на вопросительный знак. Он указывает на то, что это не просто String, а Nullable String, то есть объект может быть равен Null.

**Nullable**

На Nullable нам необходимо задержаться. Это нужно для того, чтобы программа знала, что внутри переменной может быть Null и предотвращала ошибки в написании кода: kotlin подскажет разработчику, где переменная имеет Nullable тип (следовательно, где может попасться Null), плюсом ко все код даже не скомпилируется.

Есть способы, как все же работать с Nullable типами там, где это, казалось бы, делать нельзя.   
В качестве примера поработаем с нашей третьей строкой:

myNewString.length // в данном случае компилятор выдаст ошибку

Теперь же напишем:

myNewString?.length

Что изменилось? Оператор «**?**» в данном случае осуществляет проверку на Null перед выполнением данного кода, если myNewString не равен Null, то мы получим длину строки myNewString.

Также стоит уделить внимание «элвис» оператору, по факту в случае использования оператора «**?**» мы пишем древо условий, где в случае если myNewString != null, то выполняется код, а иначе… иначе выдается ошибка компиляции.

Элвис решает эту проблему, добавляя «else» в древо условий:

myNewString?.length ?: 0

Здесь мы указываем, что если строка равно Null, то вместо ошибки компиляции нам выдастся length = 0.

Также можно использовать оператор утверждения, что это не Null «**!!**».

Попробуем использовать его:

myNewString!!.length // Утверждаем, что строка точно не равна Null

Если все же она равно Null, то получим ошибку компиляции.

Зачем нам этот Nullable? Всё просто. Зачастую он используется в работе с вводимыми (получаемыми) значениями. Например:

var readerString : String? = *readlnOrNull*() // после «=» можно поставить readline

Получая данные из консоли, запроса, файла, нужно быть готовым получить Null. Для этого и нужны Nullable типы. Если здесь вместо Nullable String поставить просто String, то код не скомпилируется (даже если поставить readline).

Перед концом раздела хотелось бы сказать о том, что еще есть ключевое слово **const val**, он нужен для объявления констант, которые можно использовать в разных файлах.

Условные операторы

Тут я не хочу задерживаться надолго, в kotlin операторы **if**, **else**, **else if** не отличаются по синтаксису и функционалу от того же C++, но меня заинтересовал другой оператор, а именно **when**. Приведу пример работы с when на примере нашей переменной myInt.

when(myInt){  
 10 -> *println*("Не 15, а 10")  
 15 -> *println*("Ого, 15!")  
 20 -> *println*("Не 15, а 20")  
 else -> {  
 *println*("не 15, не 10, не 20")  
 *println*("Обидно получилось")  
 }  
}

Здесь мы проверяем значение myInt, если он равен 10, то вызываем функцию, находящуюся после стрелочки рядом с десяткой, и так далее… Присутствует **else**, он берет на себя все неописанные случаи. После стрелочки можно описывать не одно действие, а несколько, обособив их фигурными скобками.

Стоит отметить, что when может не только исполнять функции, но и возвращать значения. Тогда после стрелки будет стоять просто значение, которое нужно вернуть.

Диапазоны

Диапазоны в Kotlin — это мощный инструмент для работы с последовательностями чисел или символов. Они позволяют легко создавать и манипулировать диапазонами значений.

Рассмотрим пример:

val range1: IntRange = 1..10 // содержит числа от 1 до 10 включительно

val range2: IntRange = 1 until 10 // содержит числа от 1 до 10 не включая 10

val range3: CharRange = ‘a’..’z’ // содержит буквы от a до z включительно

val range4 : ClosedRange<Double> = 41.1..442.1

val range5 : IntProgression = 1..10 step 2 // Устанавливаем шаг 2

val range6 : IntProgression = 10..1 downTo // Обратный перебор

Диапазоны привлекли меня тем, что мы можем создавать диапазоны разных типов, и по некоторым можем итерироваться. По типам ClosedRange, IntProgression итерировать нельзя, зато к ним тоже можно применять **in**.

Например:

*println*(50.1 in range4) // выводит true

Массивы

Массивы в kotlin являются однотипными, также у них фиксирована длина. Тип массива можно указать при его инициализации, так и сделаем:

val arr1 : Array<Int> = *arrayOf*(1, 2, 3)

А еще можно вот так:

val arr2 : IntArray = *intArrayOf*(4, 5, 6)

Явная разница лежит в том, что intArrayOf можно проинициализировать пустым без ошибки. Есть и другие расхождения, но они лежат в java, туда мы не полезем.

Важно заметить, что при использовании **var** можно пересоздать массив, удалив первичный безвозвратно. При этом и при использовании **var**, и при использовании **val** мы имеем возможность изменять отдельные элементы массива, не создавая массив заново.

Не менее важно отметить, что массив можно перебирать циклом for, то есть он итерируем.

for (elem in arr2){

println(elem)

}

Коллекции

В kotlin есть 3 основных массива: **List**, **Set**, **Map**.

**List**

Список (List) – это, грубо говоря, то же самое, что и массив, только в список можно добавлять элементы, да и удалять тоже можно. Список является по сути является интерфейсом, в отличие от массива. Список менее эффективен по производительности, чем массив.

Создадим список:

val list1 : List<Int> = *listOf*(7, 8, 9)

В данном случае он является неизменяемым, в том числе нельзя изменять его элементы без пересоздания (в массивах так было можно).

Для того чтобы список был изменяемым, надо изменить его инициализацию:

val list2 : MutableList<Int> = *mutableListOf*(10, 11, 12)

Для добавления объекта в список используется метод **add(index: Int, element : T)**, где T – тип элементов списка. Этот метод я счёл нужным описать, так как тут можно указать индекс, куда вставляется элемент. Остальные методы описывать я не буду, мне они показались не столь интересными.

Я до этого говорил про итерируемость массива, и хочу сказать, что список тоже итерируемый, что очевидно. Но List пошел дальше, у него есть свой метод для перебора значений **forEach**. Итератор **it** автоматически пройдется по каждому элементу массива. Пример:

list2.*forEach*{*println*(**it**)  
 *println*(**it** \* 2)  
}

Имя **it** является системным названием итератора, но … . Внутри forEach можно выполнить несколько действий одновременно.

В случаях, когда мы работаем с коллекцией, а не интервалом, предпочтительней использовать forEach, а не for. Но в for можно использовать continue и break, это является преимуществом.

Также для List существуют методы map и filter. Если говорить коротко, то при обращении к этому методу списка в фигурных скобках пишется:

* Действие, если мы используем map. Тогда это действие применится ко всем элементам списка.
* Условие, возвращающее bool. Тогда в списке останутся лишь те элементы, которые удовлетворяют условию.

**Map**

Map по факту несет в себе ассоциации «ключ – значение». Ключи уникальны, значения – нет.

Приведем пример:

val map1 : Map<Int, String> = *mapOf*(  
 1 *to* "Январь",  
 2 *to* "Февраль",  
 3 *to* "Март"  
)

Map сам по себе тоже интерфейс, и мы реализовываем сценарий Int, String. Int – ключи, String – значения.

Я бы провел аналогию с словарем из ЯП Python. Механика та же.

Также можно не только получать и менять ключи/значения, но и добавлять/удалять записи, но для этого надо немного изменить инициализацию:

val map2 : Map<Int, String> = *mutableMapOf*(  
 4 *to* "апрель",  
 5 *to* "май",  
 6 *to* "июнь"  
)

**Set**

По большому счету это то же самое, что и List, но без повторений.

Функции:

В целом, ничего особенного я для себя не подметил. Единственное, что хочу сказать, это то, что мне эстетически приятен синтаксис обозначения типов входных и выходных значений. То есть:

fun func(arg : Type) : Type { … }

Очень приятно глазу и интуитивно понятно, вот пример:

fun summ(a : Int, b : Int) : Int{  
 return a + b  
}

Разве не красиво?

Классы

**Инициализация**

Наконец-то, классы.  
Сначала я ничего особенного в сигнатуре не заметил, выпишем её:

class className(arg: Type…){…}

Внутри мы создаем функции, атрибуты и прочее, ничего особенного. Также к аргументу можно добавить **val** или **var**, чтобы сразу проинициализировать аргументы внутри класса.

Потом увидел, что можно добавить ключевое слово **constructor**, вот пример:

class University constructor(var name : String, var age : Int, var rating : Int ){}

Ничего от этого не изменится, но с этим ключевым словом мы еще встретимся.

В ходе изучения выяснилось, что вторичные и первичные конструкторы прописываются как в примере:

class University(var name : String, var age : Int, var rating : Int ){  
  
 var isOld = false  
  
 constructor(  
 name : String,  
 age : Int,  
 rating : Int,  
 isOld: Boolean  
 ) : this(name, age, rating){  
  
 this.isOld = isOld  
  
 }  
}

Таким образом мы можем прописывать свою логику для разных инициализаций.

Также можно прописывать блоки инициализации. Это блоки кода, которые выполняются по порядку при после инициализации объекта.

Пример:

class University(var name : String, var age : Int, var rating : Int ){  
  
 var isOld = false  
  
 constructor(  
 name : String,  
 age : Int,  
 rating : Int,  
 isOld: Boolean  
 ) : this(name, age, rating){  
  
 this.isOld = isOld  
  
 }  
  
 init {  
 *println*("init")  
 }  
}

Объект класса University можно двумя способами:

val un1 : University = University("Bmstu", 300, 100, true)  
val un2 : University = University("Bmstu", 300, 100)

И будет реализована разная логика.

Инициализация работает в таком порядке:

Вторичный => первичный => блоки инициализации

**Наследование**

В наследовании меня удивило то, что в kotlin нельзя просто так наследовать классы. Для этого перед ключевым словом class надо писать ключевое слово **open**, чтобы показать, что мы хотим расширить этот класс.

Также открытием для меня было то, что в kotlin нельзя наследоваться от нескольких классов одновременно. При этом возможность наследоваться от нескольких интерфейсов присутствует.

Пример наследования:

open class University(var name : String, var age : Int, var rating : Int ){  
  
 var isOld = false  
  
 constructor(  
 name : String,  
 age : Int,  
 rating : Int,  
 isOld: Boolean  
 ) : this(name, age, rating){  
  
 this.isOld = isOld  
  
 }  
  
 init {  
 *println*("Состояние isOld:")  
 *println*(isOld)  
 }  
}  
  
class Univer(name: String, age: Int, var obshaga : String) : University(name, age, rating=1000){  
  
}

Когда мы описываем класс-наследник, мы записываем его аргументы, которые нужны для инициализации класса-родителя, именно через такой синтаксис. Допустимы значения по умолчанию.

Абстрактные классы создаются через написание ключевого слова **abstract** вместо слова open.

Переопределение функций также реализовано через open. Перед переопределяемой функцией в классе-родителе пишем open, а в наследнике – **override**.

Пример:

//TIP To <b>Run</b> code, press <shortcut actionId="Run"/> or  
// click the <icon src="AllIcons.Actions.Execute"/> icon in the gutter.  
  
open class University(var name : String, var age : Int, var rating : Int ){  
  
 var isOld = false  
  
 open fun hello(){  
 *println*("Привет, я $name! ")  
 }  
  
 constructor(  
 name : String,  
 age : Int,  
 rating : Int,  
 isOld: Boolean  
 ) : this(name, age, rating){  
  
 this.isOld = isOld  
  
 }  
  
 init {  
 *println*("Состояние isOld:")  
 *println*(isOld)  
 }  
}  
  
class Univer(name: String, age: Int, var obshaga : String) : University(name, age, rating=1000){  
 override fun hello(){  
 *println*("Ну-ну, кто пришел? Я $name! Учись и не дергайся...")  
 }  
}

Здесь объект класса Univer будет при вызове использовать определение из наследника.

Для вызова родительского уже переопределенного метода используется ключевое слово **super**.

Теперь вернемся к абстрактным классам. Это уже более интересно, задержимся тут.

При наследовании дочерний класс переопределяет абстрактные методы и поля под себя. Синтаксически все то же самое, только вместо open будет abstract.

Теперь о интерфейсах.

Сами по себе интерфейсы нам не так интересны. Интересно то, что я уже сказал выше. Можно наследовать несколько интерфейсов, получая их функционал, при этом есть возможность наследовать не весь функционал интерфейса. Таким образом есть возможность собрать свой набор функций из разных интерфейсов.

При наследовании абстрактных методов интерфейса (они абстрактны по умолчанию) следует использовать override для переопределения.

**Инкапсуляция**

Модификаторов доступа в kotlin 4 штуки, 3 стандартных: **public**, **private**, **protected**; и 1 для меня необычный – **internal**, он отвечает за то что объект будет виден во всем модуле.

**Полиморфизм**

В первую очередь хочется сказать, что в kotlin доступна перегрузка функций. (ad hoc – по случаю)

Также реализуем принцип subtyping (полиморфизм включения). То есть родительский класс может быть использован вместо дочернего, если и родителя переопределены нужные методы.

**Анонимные и lambda функции**

Начнем с анонимных переменных. В kotlin они реализованы через ключевое слово **fun**, при желании в типе функции можно явно указать типы и количество аргументов и выдаваемых значений.  
  
Пример:

val myFunc1 : (Int) -> Int = fun (num : Int): Int = num \* 5

В аргументе к fun мы указали аргумент анонимной функции. Если функция ничего не возвращает, в качестве возвращаемого значения указываем Unit.

Приведем пример lambda функции:

val myFunc2 : (Int) -> String = **{** num: Int **->** "Num is $num"  
**}**

**Extensions**

В kotlin можно расширять функционал существующих классов вне самого класса, и эти функции будут самые что ни на есть функции класса. Это я посчитал нужным записать, довольно интересно.

**Датаклассы**

Датаклассы – это классы, которые специализируются на хранении данных. В низ нельзя инициализировать свои функции, зато у них есть встроенные методы, которые удобно используются для работы с датаклассами.

Из них:

* equals() – сравнение двух объектов (по значениям, а не ссылкам)
* hashCode() – хэширование по значению
* toString() – обращение в строку (лучше переопределять для своего датакласса
* copy()

**Выводы**

Я освоил основы ЯП kotlin и нашел в нем что-то новое для себя. Если будет время и желание, я постараюсь изучить Web технологии kotlin.