План лекции

- Переменные
- Система типов
  - Примитивные типы
  - Null-безопасность
  - Тип Unit
  - Функциональные типы и Лямбда-выражения
  - Обобщенные типы
  - Механизм вывода типов
- Классы в Kotlin
  - Объявление и инициализация класса
  - Свойства (Поля) класса
- Расширение класса
  - Наследование
  - Интерфейсы
  - Функции-расширения и свойства расширения
- Особые классы и объекты (Объекты-компаньоны, Классы данных, Классы одиночки, Классы перечислений)

Источник

Переменные

Есть два способа объявить переменную: с помощью ключевых слов val и var.

```
val button1: Button =
findViewById<Button>(R.id.button1)
```

```
var button: Button =
findViewById<Button>(R.id.button1)
button = findViewById<Button>(R.id.button2)
```

```
val button1 = findViewById<Button>(R.id.button1)
```

```
var button = findViewById<Button>(R.id.button1)
button = findViewById<Button>(R.id.button2)
```

Тип следует за именем поля и отделяется от него двоеточием (:). Тип в большинстве случаев можно опустить.

В большинстве случаев предпочтение следует отдавать ссылкам val.

Хотя переменную с ключевым словом val нельзя переназначить, она определенно может изменить значение!

```
class HelloActivity : Activity() {
    val surprising: Double // пример свойства класса без поля, где хранится его значение.
        get() = Math.random() // Каждый раз при обращении к переменной surprising будет
возвращаться другое случайное значение.
```

Система типов

Kotlin — это статически типизированный язык.
Проверка типов выполняется во время компиляции
В Kotlin существует механизм вывода типов.
Особенности системы типов:

- Нет примитивных типов
- У всего есть тип
- Тип Unit
- Функциональные типы
- Null-безопасность
- Обобщенные типы

Особенность	Java	Kotlin
Суперкласс	Object	Any, Any?

Источник

Система типов: Примитивные типы

Java	Kotlin
В Java есть типы int, float, boolean и т. д., которые не наследуются от Object	В Kotlin нет примитивных типов, которые бы загромождали его систему типов, по крайней мере, на уровне программиста.
У каждого примитивного типа Java есть эквивалент — класс-обертка (boxed type).	Отсутствие элементарных типов упрощает программирование, так как не требуется использовать специальные функции, и поддерживаются коллекции числовых и логических значений, не требующие преобразовывать элементарные значения в экземпляры классов и обратно.
	Внутреннее представление простых типов в Kotlin не связано с системой типов этого языка.

Источник

Система типов: Null-безопасность

Kotlin различает типы с поддержкой и без поддержки значения null.
В Kotlin необходимо явно обрабатывать пустые ссылки.
Kotlin использует такое разделение для всех типов: на поддерживающие null (с окончанием ?) и не поддерживающие null: например, String и String?, Person и Person?.

```
var a: Int = 100; // Может иметь значение из [–2 147 483 648 ... 2 147 483 647]
var a_null: Int? = null; // Может иметь значение из [–2 147 483 648 ... 2 147 483 647] и null
```

Any — это родительский класс для всех классов, не поддерживающих значения null, а Any? — это родительский класс для всех классов с поддержкой этого значения
любой тип, не поддерживающий null, является дочерним типом соответствующего типа, поддерживающего null

```
// Рабочий код
val x: Int = 3
val y: Int? = x
// А этот – нет:
// val x: Int? = 3
// val y: Int = x
```

Система типов: Null-безопасность

Kotlin различает типы с поддержкой и без поддержки значения null.
В Kotlin необходимо явно обрабатывать пустые ссылки.
Kotlin использует такое разделение для всех типов: на поддерживающие null (с окончанием ?) и не поддерживающие null: например, String и String?, Person и Person?.

```
var a: Int = 100; // Может иметь значение из [–2 147 483 648 ... 2 147 483 647]
var a_null: Int? = null; // Может иметь значение из [–2 147 483 648 ... 2 147 483 647] и null
```

Any — это родительский класс для всех классов, не поддерживающих значения null, а Any? — это родительский класс для всех классов с поддержкой этого значения
любой тип, не поддерживающий null, является дочерним типом соответствующего типа, поддерживающего null

```
// Рабочий код
val x: Int = 3
val y: Int? = x
// А этот – нет:
// val x: Int? = 3
// val y: Int = x
```

Система типов: Null-безопасность

Оператор точки (.), который также называют оператором разыменования, нельзя использовать с поддерживающими null-типами, потому что он может вызвать NPE (NullPointerException).

```
val s: String? = canReturnNull()
val l = s.length
```

```
val s: String? = canReturnNull()
val l = if (s != null) s.length else null
```

```
val s: String? = canReturnNull()
val l = s?.length
```

Kotlin определит тип значения l как Int?. Этот синтаксис удобно использовать в цепочках вызовов:

Java	Kotlin
City city = Optional.ofNullable(map.get(companyName)) .flatMap(Company::getManager) .flatMap(Employee::getAddress) .flatMap(Address::getCity) .getOrElse(null);	**val city: City? = map[companyName]?.manager?.address?.city**

Система типов: Null-безопасность

Kotlin позволяет также принять на себя всю ответственность за происходящее, т. е. за возможные исключения NPE

val city: City? = map[companyName]!!.manager!!.address!!.city // В этом фрагменте, если какой-то элемент будет иметь значение null (кроме city), будет возбуждено исключение NPE.

Иногда вместо null желательно использовать определенное значение по умолчанию.

Java	Kotlin
City city = map.getOrElse(City.UNKNOWN) .flatMap(Company::getManager) .flatMap(Employee::getAddress) .flatMap(Address::getCity) .getOrElse(City.UNKNOWN);	val city: City = map[company]?.manager?.address?.city ?: City.UNKOWN

Здесь, если какой-либо элемент имеет значение null, вместо него будет использовано специальное значение по умолчанию, определяемое оператором ?:, который иногда называют оператором Элвис.
Это эквивалентно: if (name?.length == null) 0 else name.length

Система типов: Тип Unit

В Kotlin у всего всегда есть значение.
Даже метод, который ничего специально не возвращает, имеет значение по умолчанию.
Это значение называется Unit.
Unit — это имя ровно одного объекта, значение, которое присваивается, если никакого иного значения нет.
Тип объекта Unit называется Unit.
Если код функции не возвращает значение явно, то функция имеет значение типа Unit.

Система типов: Функциональные типы

Система типов Kotlin поддерживает функциональные типы.
С помощью функциональных типов функции могут получать другие функции в качестве параметров или возвращать их в виде значений. Такие функции называются - функциями высшего порядка.
Возможность передавать функции в качестве аргументов в другие функции является основой функциональных языков.

```
fun getCurve(
    surface: (Double, Double) -> Int,
    x: Double
): (Double) -> Int {
    return { y -> surface(x, y) }
}
```

Kotlin поддерживает функциональные литералы: лямбда-выражения.
В Koltin лямбда-выражения всегда окружены фигурными скобками. В этих скобках список аргументов находится слева от стрелки ->, а выражение, представляющее собой значение выполнения, лямбда-выражения, располагается справа.

```
var la: (Int, Int) -> String = { x: Int, y: Int -> x * y; "down on the corner" }
```

Java	Kotlin
Function<Double, Double> func = x -> Math.pow(x, 2.0);	**val func: (Double) -> Double = { x -> x.pow(2.0) }**

Система типов: Функциональные типы

Если последним аргументом функции является другая функция (функция высшего порядка), то можно вынести лямбда-выражение, переданное в качестве параметра, за скобки, которые обычно ограничивают фактический список параметров

```
fun doSMTH(param: Int, smth:() -> Unit) {} // Последний аргумент,  - это функция


...
// Use
doSMTH(1, { println("smth")})
// Or
doSMTH(1) {
    println("smth")
}
```

Система типов: Обобщенные типы

Как и в Java, система типов Kotlin поддерживает переменные типов.
Обобщенные типы (generic types) представляют типы, в которых типы объектов параметризированы.
Обобщенными могут быть:

- Функции
- Типы

```
fun <T> simplePair(x: T, y: T) = Pair(x, y)
class Person<T, K>(val id: T, val name: K)
```

https://metanit.com/kotlin/tutorial/6.1.php

Система типов: Механизм вывода типов

Компилятор может определить тип идентификатора, располагая уже имеющейся у него информацией
Если компилятор сам может определить тип, то разработчику не нужно его указывать.

Иногда автоматическое определение типа невозможно, например, когда тип значения нельзя определить однозначно или когда поле не инициализируется. В таких ситуациях требуется явно объявлять тип.

Классы

Kotlin поддерживает объектно-ориентированную парадигму программирования.
Представлением объекта является **класс**. Класс фактически представляет определение объекта.
А объект является конкретным воплощением класса.

Для создания объекта необходимо вызвать конструктор класса.
По умолчанию компилятор создает конструктор, который не принимает параметров и который мы можем использовать.
Но также мы можем определять свои собственные конструкторы.
Классы в Kotlin могут иметь один первичный конструктор (primary constructor) и один или несколько вторичных конструкторов (secondary constructor).

```
class Person
val alex: Person = Person()
```

Классы: Первичный конструктор

```
class Person constructor(name: String) {
    val name: String = name
}
```

Первичный конструктор является частью заголовка класса и определяется сразу после имени класса.
Конструкторы, как и обычные функции, могут иметь параметры.
Класс может иметь только один первичный конструктор.

```
class Person constructor(val name: String) {
}
```

Первичный конструктор также может использоваться для определения свойств. Свойства определяются как и параметры, при этом их определение начинается с ключевого слова val (если их не планируется изменять) и var (если свойства должны быть изменяемыми).

```
class Person (val name: String)
```

Если первичный конструктор не имеет никаких аннотаций или модификаторов доступа, как в данном случае, то ключевое слово constructor можно опустить

```
class Person (val name: String, val surname: String?, val age: Int)
```

https://metanit.com/kotlin/tutorial/4.3.php

Классы: Инициализатор

```
class Person (_name: String, _age: Int){
    val name: String
    var age: Int = 1

    init{
        name = _name
        if(_age >0 && _age < 110) age = _age
    }
}
```

Для инициализации объектов можно использовать блоки инициализаторов

Можно определить любую логику, которая должна выполняться при инициализации объекта.

Вклассе может быть определено одновременно несколько блоков инициализатора.

https://metanit.com/kotlin/tutorial/4.3.php

Классы: Вторичный конструктор

Вторичные конструкторы определяются в теле класса с помощью ключевого слова **constructor.**
В классе могут быть одновременно и первичный, и вторичные конструкторы.
Можно определять больше одного вторичного конструктора
Можно делегировать из одного конструктора установку значений другому конструктору. Для этого применяется ключевое слово this:
Если для класса определен первичный конструктор, то вторичный конструктор должен вызывать первичный с помощью ключевого слова this

```
class Person{
    val name: String
    var age: Int = 0

    constructor(_name: String){
        name = _name
    }
    constructor(_name: String, _age: Int){
        name = _name
        age = _age
    }}
```

```
class Person{
    val name: String
    var age: Int = 0

    constructor(_name: String){
        name = _name
    }
    constructor(_name: String, _age: Int): this(_name){
        age = _age
    }
}
```

Классы: Вторичный конструктор

Вторичные конструкторы определяются в теле класса с помощью ключевого слова **constructor.**
В классе могут быть одновременно и первичный, и вторичные конструкторы.
Можно определять больше одного вторичного конструктора
Можно делегировать из одного конструктора установку значений другому конструктору. Для этого применяется ключевое слово this:
Если для класса определен первичный конструктор, то вторичный конструктор должен вызывать первичный с помощью ключевого слова this

```
class Person{
    val name: String
    var age: Int = 0

    constructor(_name: String){
        name = _name
    }
    constructor(_name: String, _age: Int){
        name = _name
        age = _age
    }}
```

```
class Person{
    val name: String
    var age: Int = 0

    constructor(_name: String){
        name = _name
    }
    constructor(_name: String, _age: Int): this(_name){
        age = _age
    }
}
```

Классы: Приватный конструктор

```
class Person private constructor(val name: String)
```

Классы: Свойства (Поля)

Каждый класс может хранить некоторые данные или состояние в виде свойств.
Свойства представляют переменные, определенные на уровне класса с ключевыми словами val и var.
Свойство в Kotlin напоминает сочетание поля Java и его метода чтения (если свойство доступно только для чтения и определено с помощью ключевого слова val) или его методов чтения и записи (если оно определено с помощью ключевого слова var).
Если свойство определено с помощью **val**, то значение такого свойства можно установить только один раз, то есть оно immutable.
Если свойство определено с помощью **var**, то значение этого свойства можно многократно изменять.
Объявлять методы доступа к свойствам не требуется. Они генерируются автоматически во время компиляции
Свойство должно быть инициализировано, то есть обязательно должно иметь начальное значение.

```
class Person{
    var name: String = “No Name"
    var age: Int = 18
}
```

```
class Person(val name: String, var age: Int)
```

```
class Person{
    val name: String
    var age: Int

    constructor(_name: String, _age: Int){
        name = _name
        age = _age } }
```

https://metanit.com/kotlin/tutorial/4.1.php

Классы: Свойства (Поля)

```
val person = Person("Light Yagami", 17)
println(person.name)
```

Код выглядит так, будто он обращается к полю name напрямую, на самом деле используется сгенерированный метод чтения. Он имеет то же имя, что и поле, и его вызов не должен сопровождаться круглыми скобками.

```
class Person(val name: String){
    var age: Int = 18
        set(value){
            if((value>0) and (value < 110))
                field = value}}
```

Kotlin поддерживает настройку методов доступа и записи для свойства.

Синтаксис

var имя_свойства[: тип_свойства] [= инициализатор_свойства]
    [getter]
    [setter]

```
class Person(val name: String, var age: Int){
    val isAdult: Boolean
        get() = age >= 18
}
```

```
val person = Person("Light Yagami", 17)
println(person.isAdult)
```

Идентификатор **field** представляет автоматически генерируемое поле, которое непосредственно хранит значение свойства. То есть свойства фактически представляют надстройку над полями, но напрямую в классе мы не можем определять поля, мы можем работать только со свойствами. Стоит отметить, что к полю через идентификатор field можно обратиться только в геттере или в сеттере, и в каждом конкретном свойстве можно обращаться только к своему полю.

https://metanit.com/kotlin/tutorial/4.2.php

Классы: Функции класса

Класс также может содержать функции.
Функции определяют поведение объектов данного класса.
Функции класса определяется также как и обычные функции.
В функциях, которые определены внутри класса, доступны свойства этого класса.

```
class Person(val name: String, var age: Int){
    fun sayHello(){
        println("Hello, my name is $name")
    }
}
```

https://metanit.com/kotlin/tutorial/4.1.php

Классы: Наследование

Базовый класс (класс-родитель, родительский класс, суперкласс), который определяет базовую функциональность.
Производный класс (класс-наследник, подкласс), который наследует функциональность базового класса и может расширять или модифицировать ее.

По умолчанию классы закрыты для расширения. Чтобы разрешить расширение нужно использовать ключевое слово **open**

При наследовании производный класс должен вызывать первичный конструктор (а если такого нет, то конструктор по умолчанию) базового класса.
Вызвать конструктор базового класса в производном классе можно двумя способами.

- после двоеточия сразу указать вызов конструктора базового класса
- вызвать конструктор базового класса - определить в производном классе вторичный конструктор и в нем вызвать конструктор базового класса с помощью ключевого слова **super**

```
open class Person{
    var name: String = "No Name"
    fun printName() = println(name)
}

class Programmer: Person()

class Manager: Person{
    constructor() : super()
}
```

```
open class Person(val name: String){
    fun printName() = println(name)
}
class Employee(empName: String):
Person(empName)
```

https://metanit.com/kotlin/tutorial/4.9.php

Классы: Переопределение методов и свойств

В производном классе для переопределения свойства перед ним указывается аннотация **override**. Если свойство определяется через первичный конструктор, то также перед его определением ставится аннотация open

При переопределении в производном классе к этим функциям применяется аннотация override Чтобы функции базового класа можно было переопределить, к ним применяется аннотация open.

Запретить дальнейшее переопределение функции в классах-наследниках можно с применением ключевоего слово final

```
open class Person(val name: String, open var age: Int = 1){
//    open var age: Int = 1
    open fun display() = println("Name: $name")
}
open class Adult(name: String, override var age: Int = 18): Person(name, age)

class Employee(name: String, val company: String): Person(name){
    final  override fun display() = println("Name: $name    Company: $company")
}
```

https://metanit.com/kotlin/tutorial/4.10.php

Классы: Интерфейсы

Интерфейсы представляют контракт - набор функциональности, который должен реализовать класс.
Интерфейсы могут содержать объявления свойств и функций, а также могут содержать их реализацию по умолчанию.
Интерфейсы позволяют реализовать в программе концепцию полиморфизма и решить проблему множественного наследования, поскольку класс может унаследовать только один класс, зато интерфейсов он может реализовать множество.

```
interface Workable {
    fun work() }

interface Student {
    fun study() }

class Person(): Workable, Student {
    override fun work() {
        println("I work now!")}

    override fun study() {
        println("I study now!")  }
}
```

https://metanit.com/kotlin/tutorial/4.8.php

Классы: Функции и свойства - расширения

Функции расширения (**extension function**) позволяют добавить функционал к уже определенным типам. При этом типы могут быть определены где-то в другом месте, например, в стандартной библиотеке.

Для атрибутов класса можно объявить свойства-расширения. Идея таже — на самом деле вы не расширяете класс, но можете создавать новые атрибуты, доступные с помощью точечной нотации

```
class Rectangle(val width: Double, val height: Double)

val Rectangle.area: Double
    get() = width * height

fun Rectangle.scaleWidth(coef: Int): Double {
    return this.width * coef
}
```

https://metanit.com/kotlin/tutorial/5.5.php

Классы: Объекты-компаньены

Классы в Kotlin не имеют статических членов.
Объект-компаньон— это объект-одиночка, всегда связанный с классом Kotlin.
Хотя это и не обязательно, чаще всего определение объекта-компаньона размещается в нижней части связанного с ним класса.
Объекты-компаньоны могут иметь имена, расширять классы и наследовать от интерфейсов.
При ссылке на член объекта-компаньона из класса, который содержит его, нужно указать ссылку с именем вмещающего класса

```
data class Person(val name: String,
                  val registered: Instant = Instant.now()) {
    companion object {
        fun create(xml: String): Person {
            TODO("Реализовать позже" )
        }
    }
}
```

Источник

Классы: Классы данных

Иногда классы бывают необходимы только для хранения некоторых данных.
В Kotlin такие классы называются data-классы.
Они определяются с модификатором **data**

При компиляции такого класса компилятор автоматически добавляет в класс функции с определенной реализацией, которая учитывает свойства класса, которые определены в первичном конструкторе:

- **equals()**: сравнивает два объекта на равенство
- **hashCode()**: возвращает хеш-код объекта
- **toString()**: возвращает строковое представление объекта
- **copy()**: копирует данные объекта в другой объект

При этом чтобы класс определить как data-класс, он должен соответствовать ряду условий:

- Первичный конструктор должен иметь как минимум один параметр
- Все параметры первичного конструктора должны предваряться ключевыми словами **val** или **var**, то есть определять свойства
- Свойства, которые определяются вне первичного конструктора, не используются в функциях toString, equals и hashCode
- Класс не должен определяться с модификаторами **open**, **abstract**, **sealed** или **inner**.

```
data class Person(val name: String, val age: Int)
val p = Person("User");
println(p.component1() + " " + p.component2())
```

https://metanit.com/kotlin/tutorial/4.12.php

Классы: Классы одиночки

Часто бывает необходимо создать единственный экземпляр данного класса. Такой экземпляр называется синглтоном (одиночкой).
Шаблон проектирования «Синглтон» – это метод, гарантирующий невозможность создания более одного экземпляра класса.
В Java этот шаблон имеет сложную реализацию, потому что трудно гарантировать создание не более одного экземпляра.
В Kotlin синглтон легко создать, заменив слово class словом object

```
object Hermit: Person("hermit", 100)
```

Классы: Перечисления

Классы перечислений в Kotlin очень похожи на перечисления в Java.
Они создают класс, который не может быть подклассом и имеет фиксированный набор экземпляров.
Как и в Java, перечисления не могут быть подклассами других типов, но могут реализовывать интерфейсы и иметь конструкторы, свойства и методы.

```
enum class GymActivity {
    BARRE, PILATES, YOGA, FLOOR, SPIN, WEIGHTS
}
```

Перечисления очень хорошо работают с оператором when.

```
fun requiresEquipment(activity: GymActivity) = when (activity) {
    GymActivity.BARRE -> true
    GymActivity.PILATES -> true
    GymActivity.YOGA -> false
    GymActivity.FLOOR -> false
    GymActivity.SPIN -> true
    GymActivity.WEIGHTS -> true
}
```

Спасибо за внимание!