# Отчёт об изучение языка программирования Kotlin

Выполнил Загребельный Алесандр

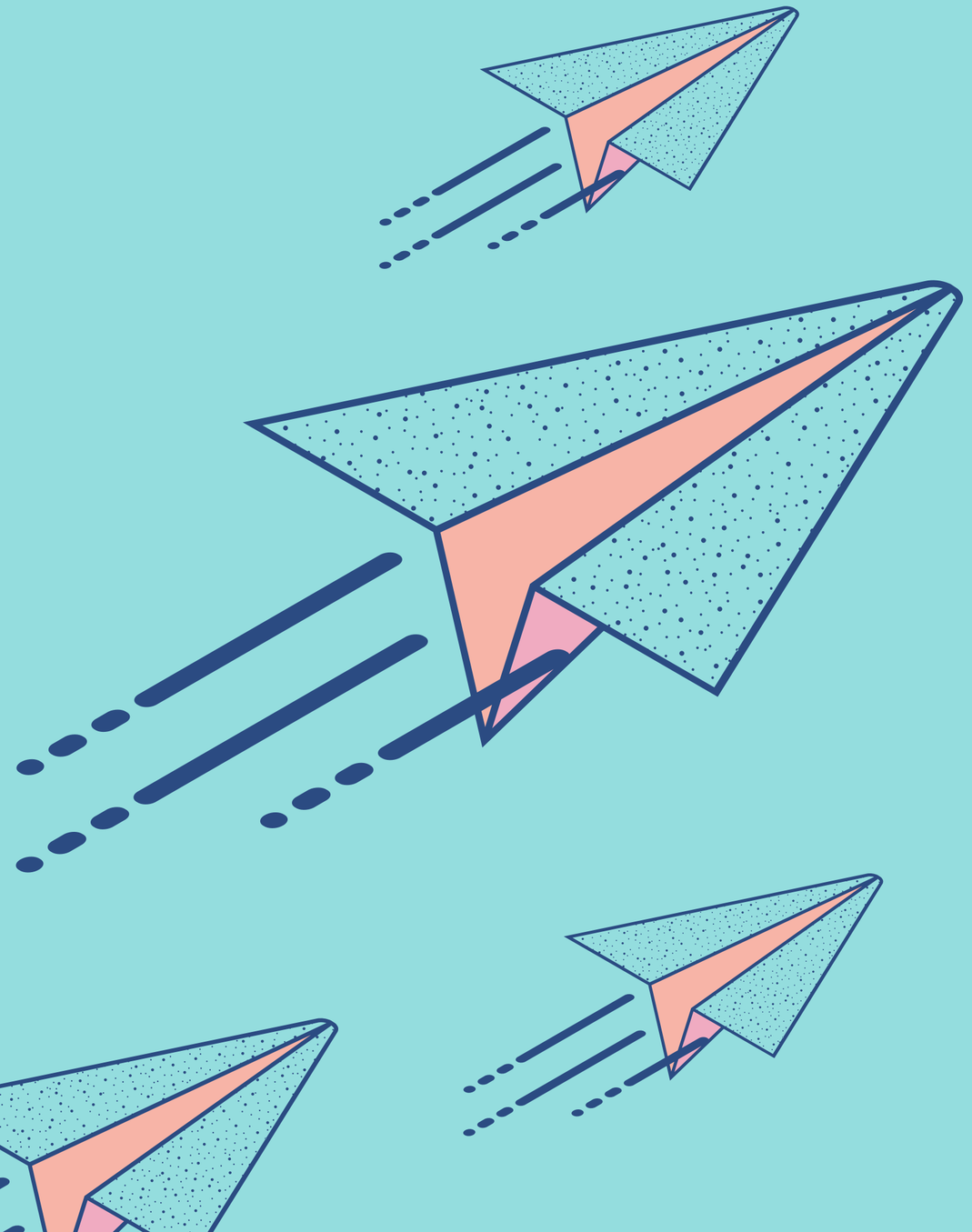
# Знакомство с Kotlin

# Знакомство с языком Kotlin

Kotlin – это статически типизированный, объектно-ориентированный язык программирования, работающий поверх Java Virtual Machine.

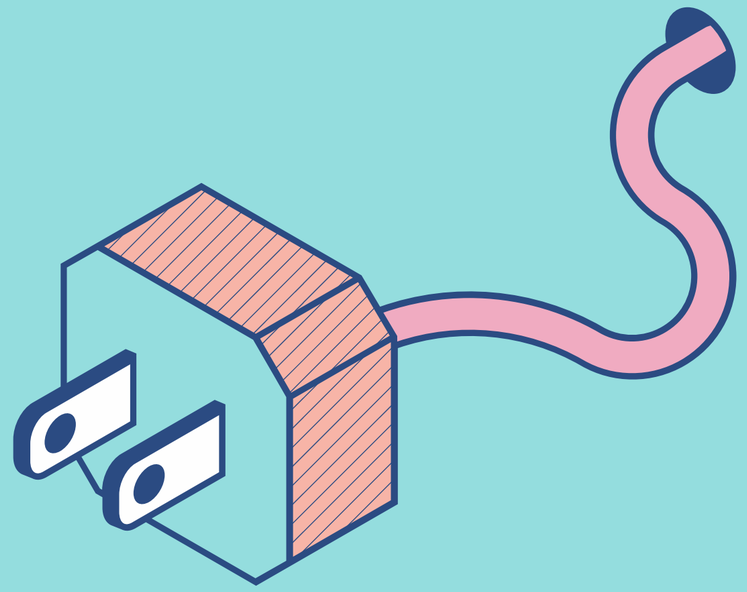
# Знакомство с языком Kotlin

**Основная цель языка Kotlin**- предложить более **компактную**, **производительную** и **безопасную** альтернативу языку Java, пригодную для использования везде, где сегодня применяется Java.



# Знакомство с языком Kotlin

**Kotlin** – это статически типизированный, объектно-ориентированный язык программирования, работающий поверх Java Virtual Machine.



# Знакомство с языком Kotlin

## Основные области применения язы Kotlin

- разработка кода, работающего на стороне **сервера** (как правило, серверной части веб-приложений)
- создание приложений, работающих на устройствах **Android**

# Знакомство с языком Kotlin

Kotlin стал рекомендуемым языком пограммирования для разработки под Android в 2019 году компанией Google

# Основы Kotlin

# Основы Kotlin

## Первая программа на языке Kotlin

```
fun main(){
    println("Hello Kotlin")
}
```

В языке Kotlin обявление функции начинается с ключевого слова **fun.**
Точкой входа в программу является функция **main**.
Вместо **System.out.println** как в Java, испльзуется функция **println**.
Так же пропал оператор **; (точка с запятой)** после завершения строки.

# Основы Kotlin

## Переменные

```
var a = "String"
val b: Int = 1
```

Для объявления переменной используется одно из ключевых слов:

- **val -** для создания неизменяемой переменной от англ. value;
- **var** - для создания изменяемой переменной от англ. variable;

Также, с помощью оператора **:** (доветочие) можно указать тип переменной явно. Если этого не сделать компилятор присвоит тип переменной самостоятльно. Значение другого типа этой переменной уже присвоить будет нельзя.

# Основы Kotlin

## Оператор if

```
fun max(a: Int , b: Int) = if (a > b) a else b
```

В Kotlin, в отличие от многих других языков программирования, значение возвращается из выражения **if.**
В данном примере функция max вернёт наибольшее значение из двух входящих чисел без использования оператора **return**.

# Основы Kotlin

## Оператор when

```
val number = 10
when (number) {
    0 -> println("Ноль")
    else -> println("Не-ноль")
}
```

Опертор when определяет условное выражение с несколькими "ветвями". Оно похоже на оператор switch, присутствующий в C-подобных языках.

# Основы Kotlin

## Диапазоны

```
val oneToTen = 1 .. 10

val oneToNine = 1 until 10
```

Диапазон представляет собой интервал между двумя значениями, обычно числовыми: началом и концом. Диапазоны определяются с помощью оператора **..**

Когда последнее значение диапазона не должно учитываться нужно использовать функцию **until**.

# Основы Kotlin

## Цикл for

```
val range = 1..5
for(i in range){
    println("Элемент $i")
}
```

Данное использование цикла является аналогом цикла **for-each** в Java.

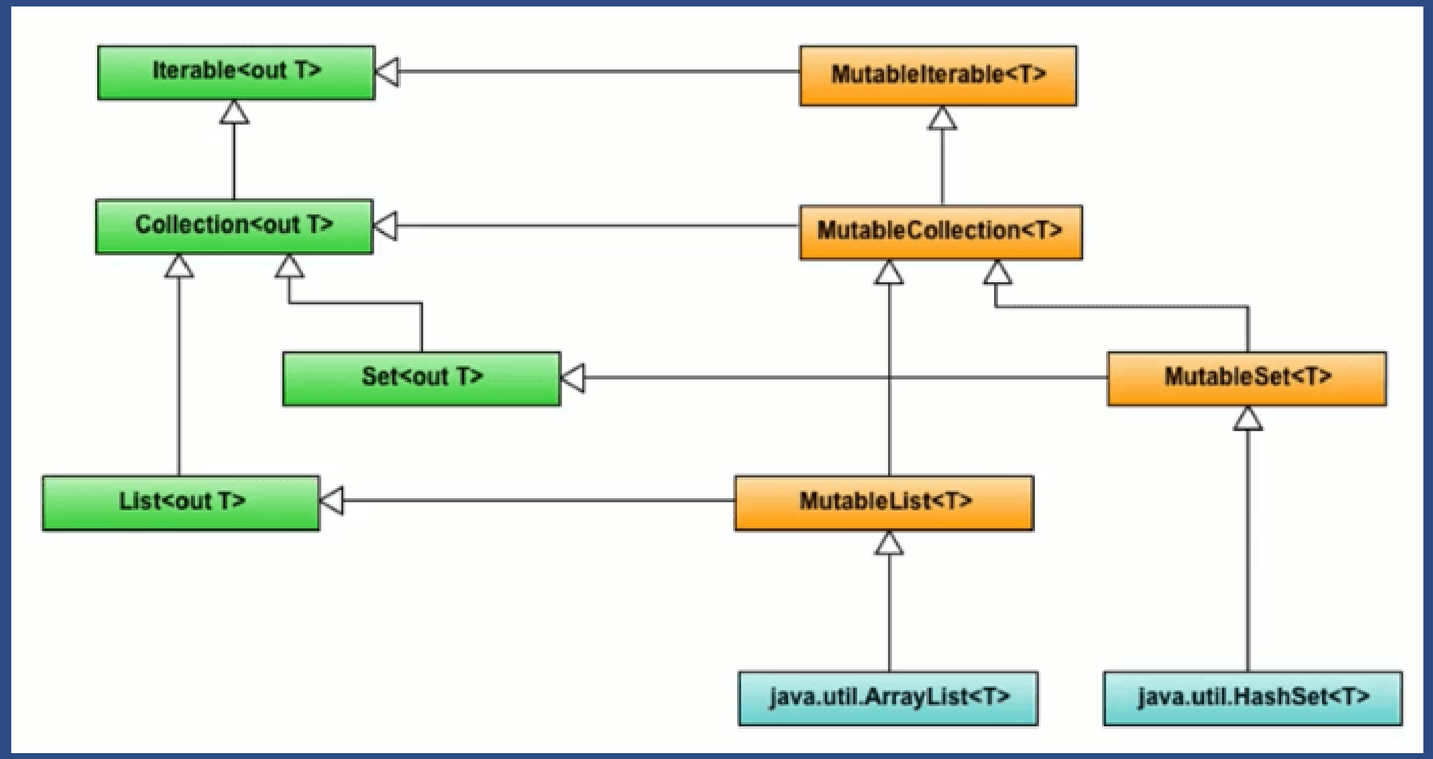
```
for (i in 0 until 8) {
   println("Элемент $i")
}
```

Более удобное использование аналогичного цикла for в Java

# Коллекции

# Основы Kotlin

## Коллекции




В Kotlin коллекции разделяются на **изменяемые (mutable)** и **неизменяемые (immutable)** коллекции.

**Mutable**-коллекция может изменяться, в нее можно добавлять, в ней можно изменять, удалять элементы. **Immutable**-коллекция также поддерживает добавление, замену и удаление данных, однако в процессе подобных операций коллекция будет заново пересоздаваться.

# Основы Kotlin

## Коллекции. List

```
val list = listOf("one", "two", "three")
println("Number of elements: ${numbers.size}") // 3
```

**List<T>** хранит элементы в определённом порядке и обеспечивает к ним доступ по индексу. Индексы начинаются с нуля (0 - индекс первого элемента) и идут до lastIndex, который равен (list.size - 1).

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
numbers.removeAt(1)
numbers[0] = 0
numbers.shuffle()
println(numbers) // [4, 0, 3, 5]
```

**MutableList<T>** - это List с операциями записи, специфичными для списка, например, для добавления или удаления элемента в определённой позиции.

# Основы Kotlin

## Коллекции. Set

```
val numbers = setOf(1, 2, 3)
numbers.add(4) //error
```

**Set<T>** хранит уникальные элементы; их порядок обычно не определён. null также является уникальным элементом: Set может содержать только один null. Два множества равны, если они имеют одинаковый размер и для каждого элемента множества есть равный элемент в другом множестве.

```
val numbers = mutableSetOf(1, 2, 3)
numbers.add(1) // проигнорируется
```

**MutableSet** - изменяемый Set, предоставляет всё, что есть в Set + операции и функции для изменения элементов.

# Основы Kotlin

## Коллекции. Map

```
val numbers = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
println("All keys: ${numbersMap.keys}") // [key1, key2, key3, key4]
```

**Map<K, V>** не является наследником интерфейса Collection; однако это один из типов коллекций в Kotlin. Map хранит пары "ключ-значение" (или entries); ключи уникальны, но разные ключи могут иметь одинаковые значения.

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)

numbersMap.put("three", 3)

numbersMap["one"] = 11
```

**MutableMap** - это Map с операциями записи, например, можно добавить новую пару "ключ-значение" или обновить значение, связанное с указанным ключом.

# Объектно-ориентированное программирование

# Основы Kotlin

## ООП. Классы

```
fun main(){
    var person = Person()
    person.sayHello()
}
class Person {
    var name: String = "Joe"
    var age: Int = 18

    fun sayHello(){
        println("Hello")
    }

}
```

Классы в Kotlin объявляются также как и в Java.

То, что называют в Java полями класса теперь называется **свойствами**. При создании объекта ключевое слово **new** писать больше не нужно.

# Основы Kotlin

## ООП. Конструкторы

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

Конструкторы в Kotlin делятся на **первичные** и **вторичные**.

Для определения конструкторов применяется ключевое слово **constructor**. Если первичный конструктор не имеет никаких аннотаций или модификаторов доступа, как в данном случае, то ключевое слово **constructor** можно опустить.

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

# Основы Kotlin

## ООП. Конструкторы

```
class Person(_name: String){
    val name: String = _name
    var age: Int = 0
    constructor(_name: String, _age: Int) : this(_name){
        age = _age
    }
}
```

Класс также может определять вторичные конструкторы. Они применяются в основном, чтобы определить дополнительные параметры, через которые можно передавать данные для инициализации объекта. Если для класса определен первичный конструктор, то вторичный конструктор должен вызывать первичный с помощью ключевого слова this:

# Основы Kotlin

## ООП. Блоки инициализации

```
class Person(_name: String){
    init {
      require(name.isNotBlank(), {"У человека должно быть имя!"})
      require(age > -1, {"Возраст не может быть отрицательным."})
    }
}
```

Основной конструктор не может в себе содержать какую-либо логику по инициализации свойств (исполняемый код). Он предназначен исключительно для объявления свойств и присвоения им полученных значений. Поэтому вся логика может быть помещена в блок инициализации - блок кода, обязательно выполняемый при создании объекта.

# Основы Kotlin

## ООП. Модификаторы доступа

### public

Является модификатором по умолчанию.
Классы, функции, свойства, объекты, интерфейсы с этим модификатором видны в любой части программы.

### protected

Видимость:

- внутри класса, в котором объявлены
- в дочерних классах

Данным модификатором нельзя пометить классы, переменные или функции, находящиеся вне класса.

### private

Данные будут доступны только в пределах конкретного класса или файла.

### internal

Классы, объекты, интерфейсы, функции, свойства, конструкторы с этим модификатором видны в любой части модуля, в котором они определены. Модуль представляет набор файлов Kotlin, скомпилированных вместе в одну структурную единицу.

# Основы Kotlin

## ООП. Абстрактные классы

```
abstract class Polygon {
    abstract fun draw()
}
class Rectangle : Polygon() {
    override fun draw() { // рисование прямоугольника }
}
```

Класс может быть объявлен как **abstract** со всеми или некоторыми его членами. Абстрактный член не имеет реализации в своём классе. Отличительной особенностью абстрактных классов является то, что мы не можем создать объект подобного класса.

# Основы Kotlin

## ООП. Интерфейсы

```
interface MyInterface {
    val prop: Int // абстрактное свойство
    fun bar() { // необязательное тело }
}
class Child : MyInterface {
    override val prop: Int = 29
    override fun bar() { // тело }
}
```

**Интерфейсы** в Kotlin могут содержать объявления абстрактных методов, а также методы с реализацией.

Класс должен реализовывать все абстрактные свойства и функции, определённые в интерфейсе. Интерфейсы похожи на абстрактные классы тем, что нельзя создать их экземпляры и они могут определять абстрактные или конкретные функции и свойства. Отличие в том, что интерфейсу не важна связь “родитель-наследник”, он задаёт лишь правила поведения.

# Лямбда-выражения

# Основы Kotlin

## Лямбда-выражения

```
val sum = { x: Int, y: Int -> x + y }
println(sum(1 , 2)) // Вызов лямбда-выражения, хранящегося в переменной
```

**Лямбда-выражения** - это небольшие фрагменты кода, которые можно передавать другим функциям. Лямбда-выражение можно сохранить в переменной, а затем обращаться к ней как к обычной функции (вызывая с соответствующими аргументами).

Лямбда-выражение всегда заключено в скобки **{...}**, а тело функции начинается после знака **->**.

# Функции высшего порядка: лямбда-выражения как параметры и возвращаемые значения

# Функции высшего порядка

## Объявление ункции высшего порядка

```
vat sum = { x: Int , y: Int -> x + y }
```

**Функциями высшего порядка** называют функции, которые принимают другие функции в аргументах и/или возвращают их. В данном случае компилятор определит, что переменная  sum имеет тип функции.

# Спасибо за внимание!

Загребельный Александр