Московский государственный технический
университет им. Н.Э. Баумана

Факультет «Информатика и система управления»
Кафедра ИУ5 «Система обработки информации и управления»

Курс «**ПиКЯП**»

Отчёт по Домашнему заданию

Выполнил:
Студент группы ИУ5-36Б
Турсков Е. В.

Подпись и дата:

Проверил:
Преподаватель каф. ИУ5

Подпись и дата:

Реферат

Тема: Язык программирования Kotlin: особенности, парадигмы, конструкции и применение

Введение

Язык программирования Kotlin, разработанный компанией JetBrains, появился на сцене в 2011 году и за относительно короткий срок обрел широкую популярность. Изначально задуманный как более лаконичная и безопасная альтернатива языку Java, Kotlin развивался, постепенно становясь универсальным инструментом для разработки: от серверной логики и Android-приложений до кроссплатформенных проектов и веб-сервисов. В 2017 году Google официально объявила о поддержке Kotlin для разработки под Android, что способствовало значительному росту популярности языка среди мобильных разработчиков. Однако сфера его применения не ограничивается мобильными платформами. Благодаря мульти-платформенному подходу, Kotlin позволяет писать код, который можно использовать на разных целевых платформах – JVM, JavaScript, Native.

Цель данного реферата – всесторонне рассмотреть основные аспекты языка Kotlin, его синтаксис, ключевые парадигмы (объектно-ориентированное и функциональное программирование), типовые конструкции языка, а также продемонстрировать примеры кода, решающие конкретные задачи. Особое внимание будет уделено понятиям, связанным с безопасностью типов (null-безопасность), упрощенным синтаксисом, расширяемостью языка и его экосистеме.

1. История, мотивация и сфера применения Kotlin

Kotlin был создан как внутренняя разработка JetBrains, компании, наиболее известной своими интегрированными средами разработки (IntelliJ IDEA и другие). Главная идея состояла в создании языка, компилируемого в байт-код JVM, но имеющего более выразительный, лаконичный и безопасный синтаксис по сравнению с Java. Основной упор делался на следующие моменты:

• **Безопасность и надежность:** Избавление от целого класса ошибок, связанных с «NullPointerException» (вегызываемых использованием null-ссылок), а также улучшенные проверки типов.

• **Снижение количества шаблонного кода:** Уменьшение так называемого «бойлерплейта» (boilerplate code), характерного для Java.

• **Совместимость:** Полная совместимость с Java-кодом и существующей экосистемой, возможность плавной миграции и интеграции.

• **Поддержка функциональной парадигмы:** Возможность использовать лямбда-выражения, функции высшего порядка, каррирование и другие функциональные концепции.

Со временем Kotlin приобрел широкую поддержку и стал применяться в самых разных сценариях: серверная разработка (Ktor, Spring Boot), Android-приложения, создание DSL, разработка скриптов, кроссплатформенная разработка (Kotlin Multiplatform), а также front-end разработка под JavaScript (Kotlin/JS) и системное программирование под LLVM (Kotlin/Native).

2. Ключевые особенности и преимущества

2.1. **Null-безопасность**

Kotlin вводит четкую типовую систему для работы с nullable-переменными. В отличие от Java, где любой объект может оказаться null и при неосторожном доступе вызвать

NullPointerException, в Kotlin переменные по умолчанию не могут быть null. Для объявления переменной, которая может быть null, используется специальный синтаксис с вопросительным знаком ?. Это позволяет компилятору выполнять дополнительные проверки и предотвращать потенциальные ошибки.

Пример:

```
var  nullableString: String? =  null
_// Попытка обратиться к nullableString напрямую потребует дополнительных проверок:_
println(nullableString?.length) _// Безопасный вызов: если nullableString == null, вернется null, иначе длина строки_
```

2.2. **Короткий и выразительный синтаксис**

Kotlin предлагает богатый синтаксический сахар, позволяющий писать меньше кода для решения тех же задач. Например, при инициализации полей в конструкторе класса не требуется повторно указывать типы или создавать множество шаблонного кода.

Пример сравнения (условный псевдокод на Java и Kotlin):

Java:

```
public  class Person {
    private  String name;
    private  int  age;

    public  Person(String name,  int  age) {
        this.name = name;
        this.age = age;
    }
}
```

Kotlin:

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

2.3. **Совместимость с Java**

Kotlin может быть скомпилирован в байт-код JVM и легко интегрируется в существующие проекты на Java. Это дает возможность постепенно мигрировать кодовую базу на Kotlin или использовать его только в тех модулях, где его преимущества особенно ценны.

2.4. **Поддержка функциональных конструкций**

Функции высшего порядка, лямбды, неизменяемые коллекции, расширенные средства работы со стандартной библиотекой – все это делает Kotlin полноценным языком, сочетающим в себе принципы объектно-ориентированного и функционального программирования.

3. Парадигмы программирования в Kotlin

Kotlin позиционирует себя как мультпарадигменный язык, в котором гармонично сочетаются объектно-ориентированные (ООП) и функциональные (ФП) подходы.

3.1. **Объектно-ориентированное программирование (ООП)**

Kotlin сохраняет классические ООП-концепции: классы, интерфейсы, наследование, инкапсуляция. Однако он добавляет и некоторые упрощения. Например, классы по умолчанию

не являются открытыми для наследования – для этого явно требуется ключевое слово open. Это сделано для лучшей предсказуемости кода и снижения риска непреднамеренных переопределений.

Пример:

```
open class Animal(val name: String) {
    fun eat() = println("$name is eating.")
}

class Dog(name: String): Animal(name) {
    fun bark() = println("Woof! My name is $name")
}

fun main() {
    val dog = Dog("Rex")

    dog.eat()
    dog.bark()
}
```

3.2. **Функциональное программирование (ФП)**

ФП-стиль в Kotlin проявляется в возможности использовать лямбда-выражения, функции как параметры, расширения функций без изменения их исходного кода, функции без побочных эффектов, а также в доступе к богатой функциональной стандартной библиотеке.

Пример использования функций высшего порядка:

```
fun processNumbers(numbers: List<Int>, operation: (Int) -> Int): List<Int> {
    return numbers.map(operation)
}

fun main() {
    val list = listOf(1, 2, 3, 4)
    val doubled = processNumbers(list) { it * 2 }
    println(doubled) // [2, 4, 6, 8]
}
```

4. Основные конструктивные элементы языка Kotlin

4.1. **Объявление переменных**

В Kotlin используются ключевые слова val и var. val соответствует неизменяемой переменной (аналог final в Java), а var - изменяемой.

```
val immutableString = "Hello"
var mutableNumber = 10
mutableNumber = 20
```

4.2. **Функции**

Функции в Kotlin определяются с помощью ключевого слова fun. Тип возвращаемого значения можно указывать явно или пропустить, если он выводим по контексту. Есть поддержка функций-однострочников.

```kotlin
fun add(a: Int, b: Int): Int {
    return a + b
}

fun multiply(a: Int, b: Int) = a * b
```

4.3. Структуры данных

Kotlin предоставляет богатый набор коллекций: списки (List), наборы (Set), карты (Map). По умолчанию коллекции неизменяемы, но есть и их изменяемые аналоги (MutableList, MutableSet, MutableMap).

```kotlin
val numbers = listOf(1, 2, 3)
val mutableNumbers = mutableListOf(4, 5, 6)
mutableNumbers.add(7)
```

4.4. Условные конструкции и циклы

Kotlin поддерживает стандартные условные конструкции if, when, а также цикл for и while. Особенностью if в Kotlin является то, что он может возвращать значение, а when является более выразительной заменой switch в Java.

```kotlin
// if как выражение:
val a = 10
val b = 20
val max = if (a > b) a else b

// when:
val x = 3
val description = when(x) {
    1 -> "one"
    2 -> "two"
    in 3..5 -> "three to five"
    else -> "other"
}
```

4.5. Классы, интерфейсы, наследование

Kotlin имеет классы (по умолчанию final), интерфейсы, абстрактные классы, а также свойства, которые объединяют поле и методы доступа (геттеры/сеттеры).

```kotlin
interface Drivable {
    fun drive()
}

abstract class Vehicle(val brand: String) {
    abstract fun startEngine()
}

class Car(brand: String): Vehicle(brand), Drivable {
    override fun startEngine() {
        println("$brand engine started.")
    }
```

```
    override fun drive() {
        println("Car is moving.")
    }
}
```

4.6. **Data-классы и Sealed-классы**

Data-классы упрощают создание классов для хранения данных, автоматически генерируя toString(), equals(), hashCode() и copy()методы. Sealed-классы позволяют создавать ограниченные иерархии классов для использования в выражениях when.

```
data class User(val name: String, val age: Int)

sealed class Result {
    data class Success(val data: String): Result()
    data class Error(val message: String): Result()
}
```

5. Безопасность типов и обработка null

Одной из главных «фишек» Kotlin является строгий контроль за null-значениями. Переменные, которые могут быть null, должны быть объявлены с типом, помеченным ?. Обращение к методам таких переменных без соответствующей проверки приведет к ошибке компиляции.

Для безопасного обращения к nullable-переменным используется оператор безопасного вызова ?. или оператор «Элвис» ?:, позволяющий подставить значение по умолчанию, если переменная равна null.

```
val maybeString: String? = null
println(maybeString?.length) // null, не будет ошибки

val length = maybeString?.length ?: 0 // Если maybeString = null, вернется 0
```

6. Лямбда-выражения и функции высшего порядка

Лямбда-выражения позволяют определять анонимные функции, которые можно передавать как параметры другим функциям. Функции высшего порядка принимают или возвращают другие функции, что упрощает реализацию функциональных стилей программирования.

```
val numbers = (1..10).toList()
val filtered = numbers.filter { it % 2 == 0 } // оставляем только четные
println(filtered) // [2, 4, 6, 8, 10]
```

7. Расширения (Extension functions)

Extension-функции позволяют «расширять» возможности существующих классов, не изменяя их исходный код. Это особенно удобно при работе с библиотечными классами.

```
fun String.firstChar(): Char = this[0]

fun main() {
    val text = "Hello"
    println(text.firstChar()) // 'H'
}
```

8. Корутинный подход к асинхронному программированию

Одним из мощных инструментов Kotlin являются корутины – лёгкие потоки выполнения, упрощающие написание асинхронного и конкурентного кода. Корутины позволяют писать асинхронный код в процедурном стиле, без необходимости громоздких callback-цепочек или использования сложных конструкций потоков.

Пример простого использования корутин:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("Корутина: Привет из корутины!")
    }
    println("Главная функция: Привет!")
}
```

Вывод

```
Главная функция: Привет!
Корутина: Привет из корутины!
```

Тут корутина запускается параллельно с основным потоком, при этом код выглядит линейно и понятно.

9. Пример решения практической задачи на Kotlin

Рассмотрим задачу: необходимо обработать список сотрудников компании, отфильтровать их по определенным критериям, подсчитать средний возраст по отфильтрованной группе, а также представить результат в удобочитаемом виде.

Пример кода:

```
data class Employee(val name: String, val age: Int, val department: String)

fun main() {
    val employees = listOf(
        Employee("Alice", 30, "IT"),
        Employee("Bob", 25, "IT"),
        Employee("Charlie", 35, "Accounting"),
        Employee("Diana", 42, "Marketing"),
        Employee("Eve", 29, "IT")
    )

    // Отфильтруем сотрудников из IT-отдела
    val itEmployees = employees.filter { it.department == "IT" }

    // Подсчитаем средний возраст
    val averageAge = itEmployees.map { it.age }.average()

    // Сформируем читабельный вывод
    println("Сотрудники IT-отдела:")
```

```
    itEmployees.forEach { println("- ${it.name}, возраст: ${it.age}") }
    println("Средний возраст: $averageAge")
}
```

При выполнении этого кода будет получен следующий вывод:

```
Сотрудники IT-отдела:
- Alice, возраст: 30
- Bob, возраст: 25
- Eve, возраст: 29
Средний возраст: 28.0
```

Данный пример демонстрирует удобство использования функциональных конструкций в Kotlin, работу с коллекциями, а также простой и ясный синтаксис.

10. Экосистема и инструментарий

Помимо языка, Kotlin обладает развитой экосистемой:

• **Интеграция с IDE**: Поддержка IntelliJ IDEA, Android Studio и других IDE, автоматическая миграция Java-кода в Kotlin.

• **Библиотеки и фреймворки**: Ktor для создания асинхронных серверных приложений, kotlinx.coroutines для асинхронного программирования, многоплатформенные библиотеки.

• **Инструменты сборки и деплоймента**: Поддержка Gradle, Maven для удобной сборки проектов.

• **Тестирование**: Легкая интеграция с JUnit, TestNG, поддержка платформенных фреймворков тестирования.

Заключение

Kotlin – это современный, выразительный и безопасный язык программирования, элегантно сочетающий в себе принципы объектно-ориентированного и функционального подхода. Он был создан с учетом опыта и недостатков более ранних языков, таких как Java, и призван упростить жизнь разработчикам, сделать их код более читаемым, безопасным и лаконичным. Широкая поддержка со стороны индустрии, интеграция в Android-разработку, гибкость и кроссплатформенность делают Kotlin одним из наиболее перспективных и востребованных языков современности.

В данном реферате мы рассмотрели историю появления языка, его главные идеи и преимущества, синтаксис, парадигмы и ключевые конструкции. Кроме того, мы познакомились с использованием Kotlin для решения практической задачи. Изучение Kotlin и его особенностей открывает перед разработчиками новые возможности, позволяя писать качественный, устойчивый и надежный код, соответствующий требованиям современного рынка программного обеспечения.

Список литературы и ресурсов:

1. Официальный сайт Kotlin: https://kotlinlang.org/
2. Документация Kotlin: https://kotlinlang.org/docs/home.html
3. JetBrains Blog о Kotlin: https://blog.jetbrains.com/kotlin/
4. Книга «Kotlin in Action» (Дмитрий Жемеров, Светлана Исакова), Manning Publications.