**Функция высших порядков** — это функция, которая принимает другие функции в качестве аргументов или возвращают функцию в качестве результата. fun hOrderFunc(a:Int, f:(Int) -> Int):Int = f(a)

**Чистая функция** — это функция, которая не изменяет состояния программы вне себя. Это означает, что функция не изменяет значения глобальных переменных, не выполняет ввода/вывода, не изменяет аргументы, не работает с внешними состояниями и не взаимодействует с другими функциями или методами, которые изменяют состояние. Результат функции зависит только от ее входных параметров и не зависит от какого-либо внешнего состояния. fun add(a:Int, b: Int): Int{ return a + b}

**Лямбда-выражения** — это компактные, анонимные функции, которые можно создавать "на месте" и передавать как аргументы другим функциям или использовать в выражениях, являются основным инструментом для функционального программирования. val sum = { x: Int, y: Int -> x + y }

**Ассоциативные массивы** в Kotlin представлены интерфейсом Map, который представляет собой коллекцию пар "ключ-значение". Каждый ключ в Map уникален, и каждому ключу соответствует только одно значение. val mutableMap = mutableMapOf( "key1" to "value1", "key2" to "value2")

**Хвостовая рекурсия** — это форма рекурсии, при которой рекурсивный вызов является последним действием в функции. Это позволяет компилятору или интерпретатору оптимизировать вызовы, используя одну и ту же стековую рамку для каждого вызова функции. Преимущества: Эффективное использование памяти, предотвращение переполнения стека. fun factorialTailRec(n: Int): Int { return factorialHelper(n, 1)}tailrec fun factorialHelper(n: Int, accumulator: Int): Int {

return if (n == 0) accumulator else factorialHelper(n - 1, n \* accumulator)}

**Рекурсия вверх** — это форма рекурсии, где вычисления происходят после рекурсивного вызова. В отличие от хвостовой рекурсии, где результат накапливается и возвращается сразу, в рекурсии вверх вычисления происходят "на пути вверх" после завершения всех рекурсивных вызовов. fun factorial(n: Int): Int {return if (n <= 1) 1 else n \* factorial(n - 1)}

**Исключения в Kotlin** — это механизм, который позволяет программе реагировать на ошибки и необычные ситуации. Когда возникает исключение, выполнение программы прерывается, и управление передается в соответствующий обработчик исключений. В Kotlin, как и в Java, исключения обрабатываются с помощью блоков try, catch и finally.

**Массивы в Kotlin** представляют собой коллекцию элементов фиксированной длины одного типа. Для работы с массивами в Kotlin используется класс Array. Создание массива: val array = Array(5) { 0 }, Доступ к массиву: val element = array[0], array[0] = 10, Итерация по массиву: array.forEach { println(it) }

**Списки в Kotlin** представляют собой коллекцию элементов, длина которой может изменяться. В отличие от массивов, списки могут быть изменяемыми (mutable) или неизменяемыми (immutable). Неизменяемый список val list = listOf(1, 2, 3, 4, 5), изменяемый val mutableList = mutableListOf(1, 2, 3, 4, 5). Доступ к элементам val element = list[0], mutableList[0] = 10 Итерация list.forEach { println(it) }

Kotlin предоставляет богатую и мощную **библиотеку коллекций**, которая включает в себя несколько интерфейсов и классов для работы с различными типами коллекций. Основные коллекции включают списки (List), множества (Set), карты (Map), а также изменяемые версии этих коллекций. Интерфейсы: Iterable, Collection, List, Set, Map.

**Наследование** — это механизм ООП, который позволяет создавать новый класс на основе существующего. Класс по умолчанию в Kotlin является final (не может быть унаследован). Чтобы класс мог быть унаследован, его нужно объявить с помощью ключевого слова open. Производный класс наследует свойства и методы базового класса и может их переопределять. Для этого используется ключевое слово override.

**Делегация** позволяет одному объекту делегировать выполнение задачи другому объекту. Kotlin поддерживает делегацию двумя способами: делегация свойств (используя ключевое слово by) и делегирование реализации интерфейса.

**Метод filter** возвращает новую коллекцию, содержащую только те элементы исходной коллекции, которые соответствуют заданному условию (предикату). Метод **map** применяет заданную функцию к каждому элементу исходной коллекции и возвращает новую коллекцию, содержащую результаты этих применений. Метод **count** возвращает количество элементов в коллекции, которые соответствуют заданному условию (предикату). Метод **sortedBy** сортирует коллекцию на основе значения, возвращаемого функцией-предикатом.Метод **maxByOrNull** возвращает элемент коллекции, для которого значение функции-предиката максимально. Если коллекция пуста, возвращается null.

**упорядочивание** по количеству встречаемости элемента

fun main() {

val inputList = listOf(5, 6, 2, 2, 3, 3, 3, 5, 5, 5)

val sortedList = sortByFrequency(inputList)

println(sortedList) // Output: [5, 5, 5, 5, 3, 3, 3, 2, 2, 6]}

fun sortByFrequency(list: List<Int>): List<Int> {

// Подсчитываем количество вхождений каждого элемента

val frequencyMap = list.groupingBy { it }.eachCount()

// Сортируем элементы по количеству вхождений в порядке убывания

val sortedByFrequency = list.sortedByDescending { frequencyMap[it] }

return sortedByFrequency}