Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Кафедра инженерной психологии и эргономики

Современные языки программирования

Отчет по лабораторной работе №1

**«Использование языка программирования *Kotlin*»**

Выполнил: Еремейко А. А.

Студент группы 310901

Проверил: Усенко Ф.В.

Минск 2024

**Цель работы:** Выполнить разработку приложения с использованием языка программирования *Kotlin*.

**Вариант 8:** Добавить класс Руководитель Компании, управляющий классом Компания. Добавить класс Вип-клиент, наследующий от класса клиент и класс Срочный заказ, наследующий от класса Заказ.

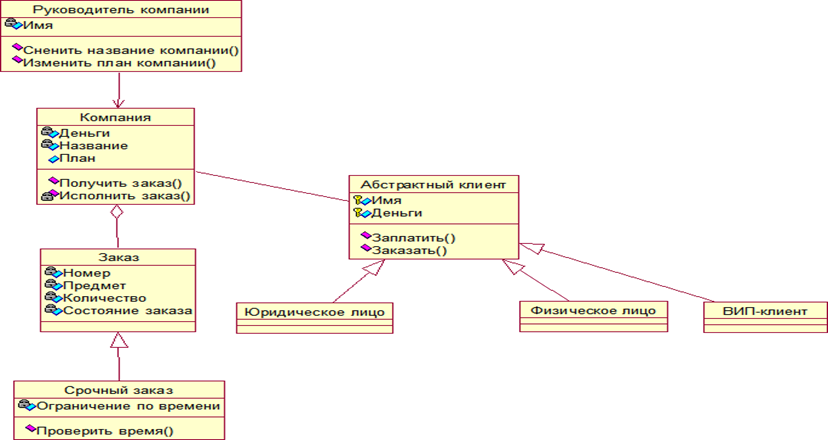


Рисунок 1 – Диаграмма классов

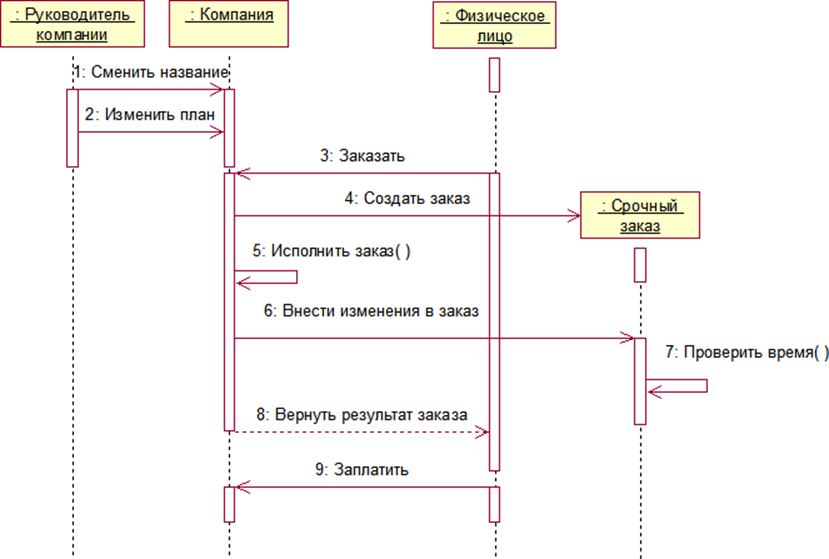


Рисунок 2 – Диаграмма последовательности

**Текст программы:**

package kotlinequivalent  
// Абстрактный клиент  
abstract class AbstractClient(  
 protected val name: String,  
 protected var money: Double  
) {  
 abstract fun pay(cost: Double, company: Company)  
 abstract fun order(company: Company, product: String, amount: Int, isUrgent: Boolean, timeLimit: String): Double  
}

package kotlinequivalent  
// Класс компании  
class Company(  
 private var money: Double,  
 private var name: String,  
 var plan: String  
) {  
 fun getMoney() : Double {return this.money}  
 fun addMoney(cost : Double) {this.money += cost}  
  
 fun getName() : String {return this.name}  
 fun setName(newName: String) {this.name = newName}  
  
 fun receiveOrder(product: String, amount: Int, isUrgent: Boolean, timeLimit: String): Double {  
 *println*("\nКомпания \"$name\" получила заказ.")  
 val order = if (isUrgent) {  
 UrgentOrder(1, product, amount, "ожидает выполнения", timeLimit).*apply* **{** *println*("\nКомпания \"$name\" создала срочный заказ №${getNumber()}: $product, " +  
 "количество - $amount шт., лимит времени: $timeLimit, состояние заказа - ${getStatus()}.")  
 // Вызов apply позволяет выполнить блок кода в контексте этого объекта  
 // Например, обратиться к свойствам без явного упоминания объекта (this используется неявно)  
 **}** } else {  
 Order(1, product, amount, "ожидает выполнения").*apply* **{** *println*("\nКомпания \"$name\" создала обычный заказ №${getNumber()}: $product, " +  
 "количество - $amount шт., состояние заказа - ${getStatus()}.")  
 **}** }  
 return executeOrder(order)  
 }  
  
 private fun executeOrder(order: Order): Double {  
 order.setStatus("в процессе выполнения")  
 *println*("\nКомпания \"$name\" приступила к выполнению заказа №${order.getNumber()}, " +  
 "состояние заказа - ${order.getStatus()}.")  
  
 if (order is UrgentOrder) {  
 order.setTimeLimit("12 часов")  
 *println*("\nКомпания \"$name\" внесла изменения в срочный заказ №${order.getNumber()}.")  
 order.checkTime()  
 }  
  
 order.setStatus("выполнен")  
 *println*("\nКомпания \"$name\" завершила выполнение заказа №${order.getNumber()}, состояние заказа - ${order.getStatus()}.")  
 return 500.0 \* order.getAmount() // возвращаем стоимость заказа  
 }  
}

package kotlinequivalent  
// Руководитель компании  
class CompanyManager(private val name: String) {

fun changeCompanyName(company: Company, newName: String) {  
 *println*("\nРуководитель $name изменил название компании с \"${company.getName()}\" на \"$newName\".")  
 company.setName(newName)  
 }

fun changeCompanyPlan(company: Company, newPlan: String) {  
 *println*("\nРуководитель $name изменил план компании с \"${company.plan}\" на \"$newPlan\".")  
 company.plan = newPlan  
 }  
}

package kotlinequivalent  
// Юридическое лицо  
class LegalEntity(name: String, money: Double) : AbstractClient(name, money) {  
 override fun pay(cost: Double, company: Company) {  
 if (money >= cost) {  
 money -= cost  
 company.addMoney(cost) // добавляем деньги компании  
 *println*("\nЮридическое лицо $name оплатило $cost$. Остаток: $money$.")  
 } else {  
 *println*("\nЮридическому лицу $name недостаточно средств для оплаты.")  
 }  
 }  
  
 override fun order(company: Company, product: String, amount: Int, isUrgent: Boolean, timeLimit: String): Double {  
 if (!isUrgent) {  
 *println*("\nЮридическое лицо $name сделало заказ \"$product\" количеством $amount шт.")  
 } else {  
 *println*("\nЮридическое лицо $name сделало срочный заказ \"$product\" количеством $amount шт., лимит времени - $timeLimit.")  
 }  
 return company.receiveOrder(product, amount, isUrgent, timeLimit)  
 }  
}

package kotlinequivalent  
// Класс заказа  
open class Order(  
 private val number: Int,  
 private val product: String,  
 private val amount: Int,  
 private var status: String  
) {  
 fun getNumber() : Int {return this.number}  
 fun getProduct() : String {return this.product}  
 fun getAmount() : Int {return this.amount}  
 fun getStatus() : String {return this.status}  
 fun setStatus(newStatus : String) {this.status = newStatus}  
}

package kotlinequivalent  
// Физическое лицо  
class PhysicalPerson(name: String, money: Double) : AbstractClient(name, money) {  
 override fun pay(cost: Double, company: Company) {  
 if (money >= cost) {  
 money -= cost  
 company.addMoney(cost) // добавляем деньги компании  
 *println*("\nФизическое лицо $name оплатило $cost$. Остаток: $money$.")  
 } else {  
 *println*("\nФизическому лицу $name недостаточно средств для оплаты.")  
 }  
 }  
  
 override fun order(company: Company, product: String, amount: Int, isUrgent: Boolean, timeLimit: String): Double {  
 if (!isUrgent) {  
 *println*("\nФизическое лицо $name сделало заказ \"$product\" количеством $amount шт.")  
 } else {  
 *println*("\nФизическое лицо $name сделало срочный заказ \"$product\" количеством $amount шт., лимит времени - $timeLimit.")  
 }  
 return company.receiveOrder(product, amount, isUrgent, timeLimit)  
 }  
}

package kotlinequivalent  
// Срочный заказ  
class UrgentOrder(  
 number: Int,  
 product: String,  
 amount: Int,  
 status: String,  
 private var timeLimit: String  
) : Order(number, product, amount, status) {  
 fun checkTime() {  
 *println*("\nПроверка времени для срочного заказа №${getNumber()}: лимит времени - $timeLimit.")  
 }  
 fun setTimeLimit(newTimeLimit : String) {this.timeLimit = newTimeLimit}  
}

package kotlinequivalent  
// ВИП-клиент  
class VipClient(name: String, money: Double) : AbstractClient(name, money) {  
 override fun pay(cost: Double, company: Company) {  
 val finalCost = cost - (cost \* 0.1)  
 if (money >= finalCost) {  
 money -= finalCost  
 company.addMoney(cost) // добавляем деньги компании  
 *println*("\nВИП-клиент $name оплатил $finalCost$ с учетом скидки. Остаток: $money$.")  
 } else {  
 *println*("\nВИП-клиенту $name недостаточно средств для оплаты.")  
 }  
 }  
  
 override fun order(company: Company, product: String, amount: Int, isUrgent: Boolean, timeLimit: String): Double {  
 if (!isUrgent) {  
 *println*("\nВИП-клиент $name сделал заказ \"$product\" количеством $amount шт.")  
 } else {  
 *println*("\nВИП-клиент $name сделал срочный заказ \"$product\" количеством $amount шт., лимит времени - $timeLimit.")  
 }  
 return company.receiveOrder(product, amount, isUrgent, timeLimit)  
 }  
}

package kotlinequivalent  
// Главная функция  
fun main() {  
 // Создаем компанию и руководителя  
 val company = Company(10000.0, "Технологии будущего", "Годовой план")  
 val manager = CompanyManager("Александр")  
  
 // Руководитель изменяет название и план компании  
 manager.changeCompanyName(company, "Инновационные технологии")  
 manager.changeCompanyPlan(company, "Ежеквартальный план")  
  
 // Создаем клиента  
 val client = PhysicalPerson("Иван", 1500.0)  
  
 // Клиент (физическое лицо) делает заказ "Ноутбук" количеством 2 штуки и получает стоимость заказа  
 val cost = client.order(company, "Ноутбук", 2, true, "24 часа")  
  
 // Клиент оплачивает заказ  
 client.pay(cost, company)  
}

Результат работы программы приведен на рисунке 3.

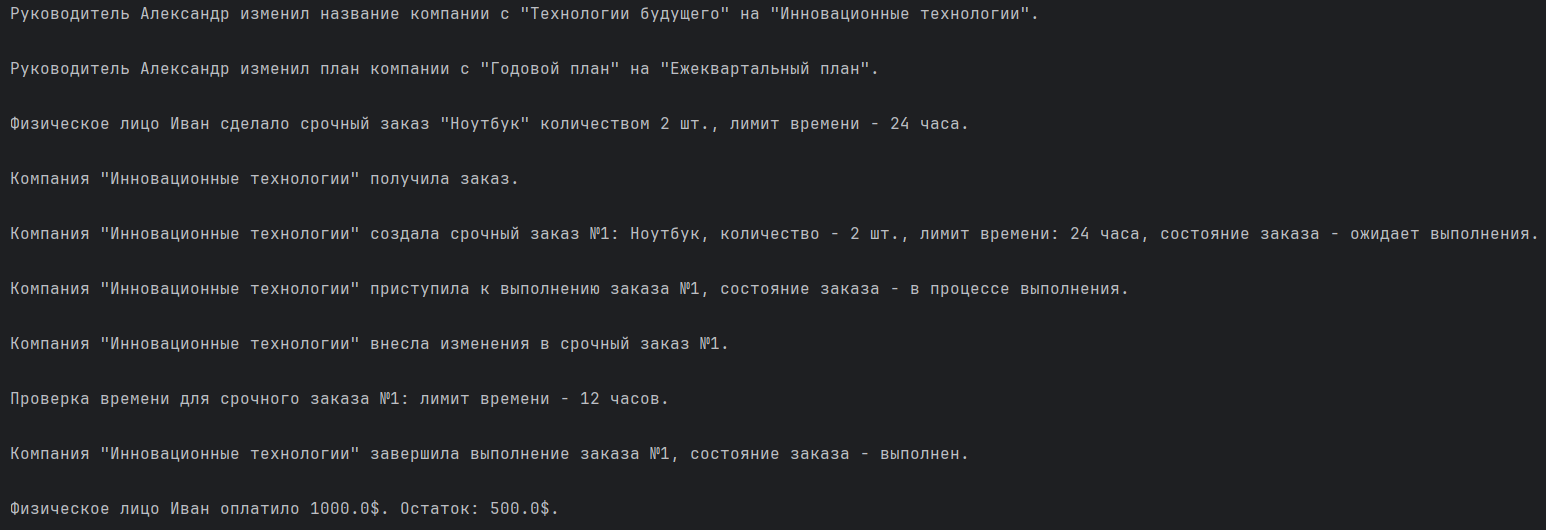


Рисунок 3 – Результат работы программы

**Ответы на вопросы к лабораторной работе:**

1. Что такое JVM?

JVM (Java Virtual Machine) — это виртуальная машина, которая выполняет программы, скомпилированные в байт-код. Она обеспечивает платформонезависимость, позволяя запускать одну и ту же программу на разных операционных системах. JVM интерпретирует байт-код и преобразует его в машинный код, который может выполняться конкретным процессором.

1. Пакеты в Kotlin?

Пакеты в Kotlin — это способ организовать классы и другие элементы программы в логические группы. Они помогают избежать конфликтов имен, особенно в больших проектах, где используются библиотеки с одинаковыми названиями классов. Пакеты также облегчают читаемость и поддержку кода. Каждый файл может быть связан с определенным пакетом, что делает программу структурированной.

1. Классы в Kotlin.

Классы являются основным строительным блоком объектно-ориентированного программирования. Они представляют собой шаблоны для создания объектов и содержат свойства (данные) и методы (функции). В Kotlin классы могут быть обычными и абстрактными.

1. Как объявить переменную в Kotlin? Различие между var и val.

Var обозначает изменяемую переменную, значение которой можно переопределить. Val используется для неизменяемых переменных, чье значение задается только один раз.

1. Сколько конструкторов доступно в Kotlin?

В Kotlin существуют два типа конструкторов: первичный и вторичный. Первичный конструктор объявляется в заголовке класса и используется для инициализации его свойств. Вторичные конструкторы нужны, когда требуется гибкость при создании объектов, например, для выполнения дополнительной логики перед инициализацией.

1. Для чего служат абстрактные классы?

Абстрактные классы используются для создания базовых шаблонов, которые должны быть конкретизированы в дочерних классах. Они содержат абстрактные методы (без реализации), которые обязаны переопределяться наследниками. Это позволяет задавать общие требования к классам, обеспечивая единообразие в проекте.

1. Какой порядок вызова конструкторов при наследовании?

При наследовании сначала вызывается конструктор базового класса, а затем выполняется конструктор производного класса.

1. Как переопределить метод родительского класса?

Чтобы переопределить метод родителя, в Kotlin используется ключевое слово override. Оно явно указывает, что метод в дочернем классе заменяет реализацию метода базового класса.

1. Что такое оператор «Elvis» в Kotlin и как он используется?

Оператор ?:, известный как «Elvis», используется для обработки значений, которые могут быть null. Он позволяет указать альтернативное значение, которое будет использовано, если исходное значение равно null. Этот оператор активно используется для сокращения проверок и предотвращения ошибок, связанных с NullPointerException.

1. Как объявить и использовать лямбда-выражения в Kotlin?

Лямбда-выражения — это функции без имени, которые используются для краткой записи кода. Они представляют собой функциональные блоки, которые можно передавать как аргументы другим функциям. Синтаксическая форма лямбда-выражений может быть представлена следующим образом:

val sum = { a: Int, b: Int -> a + b }

1. Что такое модификаторы доступа (access modifiers) в Kotlin?

Модификаторы доступа определяют уровень видимости классов, методов и свойств. В Kotlin существует четыре модификатора: public — доступен везде (по умолчанию); private — доступен только внутри того же класса или файла; protected — доступен в классе и его наследниках; internal — доступен только в рамках одного модуля.

1. Как работает оператор when в Kotlin и какие его особенности?

When — это многофункциональный оператор, заменяющий switch из других языков. Он позволяет проверить значение на соответствие множеству условий. Особенностью when является его гибкость: он может проверять диапазоны, типы данных и даже использовать логические выражения.

1. Как создать и использовать классы и объекты в Kotlin?

Классы определяются для описания объектов, а объекты — это конкретные экземпляры классов. Создание объектов производится с помощью ключевого слова new или без него, что упрощает синтаксис:

class Person(val name: String)

val person = Person("John")

1. Что такое inline-функции и зачем их применять?

Inline-функции в Kotlin используются для оптимизации производительности, особенно при работе с функциями высшего порядка. Они заменяют вызовы функций их телом, что сокращает накладные расходы на выполнение, но увеличивает размер бинарного файла.

1. Как обрабатывать исключения с помощью блока try-catch в Kotlin?

Блок try-catch обеспечивает безопасное выполнение кода, а блок finally используется для выполнения завершающих действий, независимо от того, произошло исключение или нет. Синтаксис блока:

try {

val result = 10 / 0

} catch (e: ArithmeticException) {

println("Error: ${e.message}")

} finally {

println("Cleanup")

}

1. Как работает оператор in для проверки наличия элемента в коллекции?

Оператор in проверяет, содержится ли элемент в коллекции или диапазоне.

1. Как объявить и использовать расширения для стандартных классов в Kotlin?

Расширения позволяют добавлять новые функции к существующим классам без изменения их исходного кода. Это делает код более читаемым и повторно используемым, особенно при работе со стандартными библиотеками. Пример:

fun String.printHello() = println("Hello, $this")

"World".printHello() // Hello, World

1. Что такое функциональные типы данных (function types) и как их применять?

Функциональные типы данных представляют собой типы, которые могут хранить функции. Они позволяют передавать функции как параметры или возвращать их в виде результата. Это открывает возможности для функционального программирования, такого как создание цепочек вызовов или обработки данных. Пример использования функционального типа:

val multiply: (Int, Int) -> Int = { a, b -> a \* b }

println(multiply(3, 4)) // 12

**Вывод:** Таким образом, в ходе лабораторной работы я изучил основы языка программирования *Kotlin* и выполнил разработку приложения с использованием данного языка.