Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

Отчёт

по лабораторной работе №4

на тему:

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ (ООП)**

Проверил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ф.В. Усенко

(подпись)

Выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Предченко В.М.

(подпись) гр. 214301

Минск, 2024

**Цель:** Разработать систему для учета сотрудников крупной компании с поддержкой иерархии (например, Employee, Manager, Director). Реализовать методы для расчета заработной платы с учетом бонусов, вычетов, и налогов. Включить функциональность для прогнозирования затрат компании на основе текущих данных.

**КОД**

fun main() {  
 val employee1 = Employee("Alice", 50000.0, bonus = 5000.0)  
 val employee2 = Employee("Bob", 60000.0, deductions = 2000.0)  
 val manager1 = Manager("Charlie", 80000.0, team = *listOf*(employee1, employee2))  
 val director = Director("Diana", 120000.0, managers = *listOf*(manager1))  
  
 val company = Company(*listOf*(employee1, employee2, manager1, director))  
  
 val totalExpenses = company.calculateTotalExpenses()  
 *println*("Затраты компании: $totalExpenses")  
  
 val forecast = company.forecastExpenses(12)  
 *println*("Прогнозирование затрат на 12 месяцев: $forecast")  
}

class Director(  
 name: String,  
 baseSalary: Double,  
 bonus: Double = 0.0,  
 deductions: Double = 0.0,  
 val managers: List<Manager> = *emptyList*()  
) : Employee(name, baseSalary, bonus, deductions) {  
  
 override fun calculateSalary(): Double {  
 val teamBonus = managers.*sumOf* **{ it**.calculateSalary() **}** \* 0.2  
 return super.calculateSalary() + teamBonus  
 }  
}

open class Employee(  
 val name: String,  
 val baseSalary: Double,  
 val bonus: Double = 0.0,  
 val deductions: Double = 0.0  
) {  
 open fun calculateSalary(): Double {  
 return baseSalary + bonus - deductions  
 }  
}

fun main() {  
 val employee1 = Employee("Alice", 50000.0, bonus = 5000.0)  
 val employee2 = Employee("Bob", 60000.0, deductions = 2000.0)  
 val manager1 = Manager("Charlie", 80000.0, team = *listOf*(employee1, employee2))  
 val director = Director("Diana", 120000.0, managers = *listOf*(manager1))  
  
 val company = Company(*listOf*(employee1, employee2, manager1, director))  
  
 val totalExpenses = company.calculateTotalExpenses()  
 *println*("Затраты компании: $totalExpenses")  
  
 val forecast = company.forecastExpenses(12)  
 *println*("Прогнозирование затрат на 12 месяцев: $forecast")  
}

class Manager(  
 name: String,  
 baseSalary: Double,  
 bonus: Double = 0.0,  
 deductions: Double = 0.0,  
 val team: List<Employee> = *emptyList*()  
) : Employee(name, baseSalary, bonus, deductions) {  
  
 override fun calculateSalary(): Double {  
 val teamBonus = team.*sumOf* **{ it**.calculateSalary() **}** \* 0.1  
 return super.calculateSalary() + teamBonus  
 }  
}

**КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. **Что такое первичный конструктор, и как он используется для инициализации свойств класса?**

В Kotlin первичный конструктор объявляется прямо в заголовке класса и используется для инициализации свойств класса.

Пример:

class Car(val brand: String, val model: String, var year: Int)

1. **Как в Kotlin создать вторичный конструктор, и зачем он может понадобиться?**

Вторичные конструкторы используются для предоставления дополнительных способов инициализации объектов. Они объявляются с использованием ключевого слова constructor.

Пример:

class Car {

val brand: String

val model: String

var year: Int

constructor(brand: String, model: String) {

this.brand = brand

this.model = model

this.year = 2021 // Значение по умолчанию

}

}

1. **Что такое наследование, и как его реализовать в Kotlin? Приведите пример.**

В Kotlin классы могут наследоваться с использованием ключевого слова open перед базовым классом и : для указания наследования.

Пример:

open class Vehicle(val brand: String) {

fun honk() {

println("Vehicle is honking")

}

}

class Car(brand: String, val model: String) : Vehicle(brand) {

fun drive() {

println("Driving a $brand $model")

}

}

**ВЫВОД**

В данной лабораторной работе были изучены основные принципы ООП, такие как инкапсуляция, наследование и полиморфизм. Были созданы несколько классов для выполнения цели. Задачи, выполненные в ходе работы, позволили глубже понять, как использовать возможности Kotlin для создания гибкой и расширяемой архитектуры программного обеспечения, а также улучшили навыки работы с объектно-ориентированным подходом в разработке.