Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Кафедра инженерной психологии и эргономики

Основы алгоритмизации и программирования Отчет по лабораторной работе №1

«Использование языка программирования Kotlin»

Выполнил: Капуза А.И.

Студент группы 310901

Преподаватель: Василькова А. Н.

Минск 2024

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

**Вариант 10**: Реализовать функциональную часть приложения, представленная диаграммой последовательности. Реализовать объекты и взаимоотношения, представленные на диаграмме классов. Добавить класс Заведующий кафедрой, который управляет классом университет и отвечает за создание объектов класса экзамен. У класса экзамен есть два состояния сдан и не сдан.

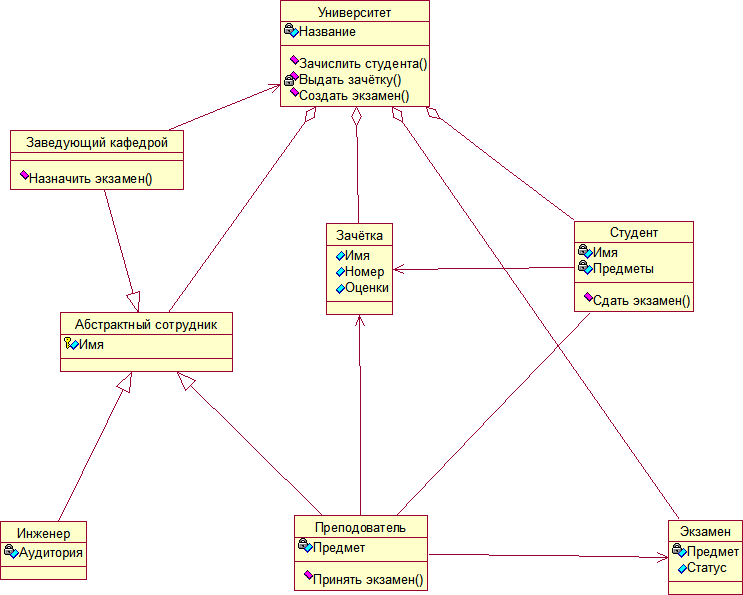


Рисунок 1 – Диаграмма классов

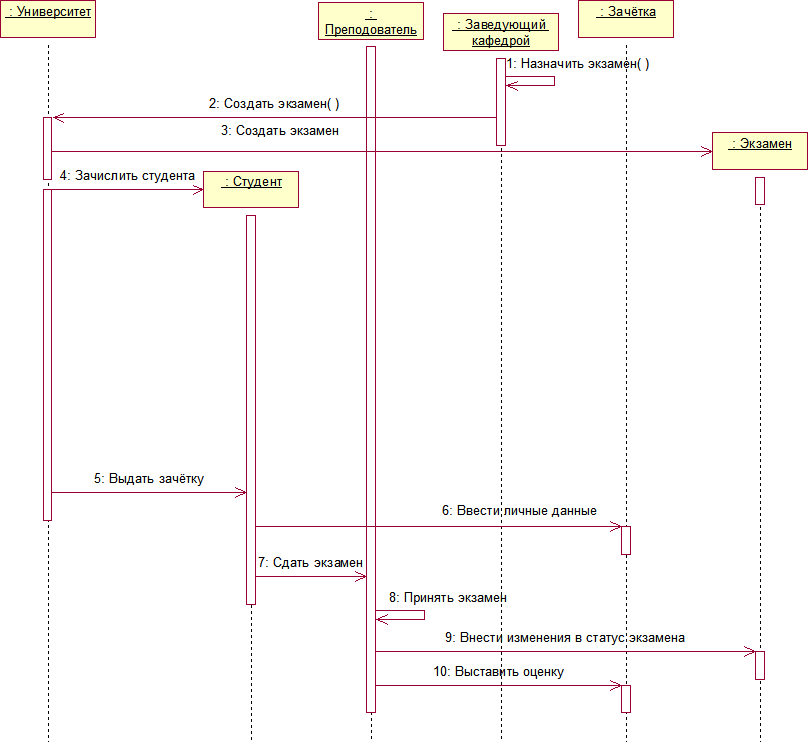


Рисунок 2 – Диаграмма последовательности

**Текст программы:**

class Subject(val name: String)

abstract class AbstractEmployee(val name: String)

class Engineer(name: String, val roomNumber: String) : AbstractEmployee(name)

class DepartmentHead (name: String) : AbstractEmployee(name)  
{ fun scheduleExam(exam: Exam, date: String) //метод отвечающий за назначения экзамена  
 { exam.date = date  
 exam.status = Exam.ExamStatus.*SCHEDULED  
 println*("Exam for ${exam.subject} scheduled on $date")  
 }  
 fun manageUniversity(university: University) //метод для управления университетом  
 { *println*("Managing university: ${university.name}")  
 university.students.*forEach*()  
 **{** student **->** *println*("Student: ${student.name}, Number: ${student.studentNumber}")  
 **}** university.exams.*forEach*()  
 **{** exam **->** *println*("Exam: ${exam.subject}, Status: ${exam.status}, Date: ${exam.date}")  
 **}** }  
}

class Exam(val subject: String)  
{ var status: ExamStatus = ExamStatus.*NOT\_PASSED* //первоначальный статус экзамена  
 var date: String = ""  
 fun passExam()  
 { status = ExamStatus.*PASSED  
 println*("Exam $subject passed.")  
 }  
 enum class ExamStatus  
 { *PASSED*,  
 *NOT\_PASSED*,  
 *SCHEDULED*,  
 *CONDUCTED* }  
 companion object { var Examstatus: String = "" }  
}

class GradeBook  
{ val records = *mutableListOf*<Record>()  
 data class Record(val studentName: String, val groupNumber: Int, val subject: String, val grade: Int)  
 fun addRecord(studentName: String, groupNumber: Int, subject: String, grade: Int)  
 { records.add(Record(studentName, groupNumber, subject, grade))  
 *println*("Record added: $studentName, $groupNumber, $subject, $grade")  
 }  
 fun getRecordsForStudent(studentName: String): List<Record>  
 { return records.*filter* { it.studentName == studentName }}  
 fun getInfo(): List<Record>  
 { return records }  
}

class Lecturer(name: String, val subject: String) : AbstractEmployee(name)  
{ fun conductExam(exam: Exam)  
 { if (exam.status == Exam.ExamStatus.*SCHEDULED*)  
 { exam.status = Exam.ExamStatus.*CONDUCTED  
 println*("Exam for ${exam.subject} has been conducted")  
 }  
 else  
 { *println*("Exam for ${exam.subject} cannot be conducted. Current status: ${exam.status}") }  
 }  
 fun manageRecordBook(group: String, students: List<Student>, subject: Subject, grade: Int)  
 { for (student in students)  
 { if (student.group == group)  
 { student.recordBook?.addRecord(student.name, student.group.hashCode(), subject.name, grade) }  
 }  
 }  
}

class University(val name: String)  
{ val students = *mutableListOf*<Student>() //список студентов  
 val exams = *mutableListOf*<Exam>() //список экзаменов  
 fun enrollStudent(student: Student) //метод для зачисления студентов  
 { students.add(student)  
 student.recordBook = GradeBook()  
 *println*("Student ${student.name} enrolled.") }  
 fun createExam(subject: String) : Exam //метод для создания экзаменов  
 { val exam = Exam(subject)  
 exams.add(exam)  
 *println*("Exam for subject $subject has been created.")  
 return exam }  
 fun listExams(): List<Exam> = exams }

class Student(val name: String, val group: String, val subjects: List<String>, val grades: MutableMap<String, Int>)  
{ val studentNumber: String = ""  
 var recordBook: GradeBook? = GradeBook()  
 private val exams = *mutableListOf*<Exam>()  
 fun takeExam(exam: Exam, grade: Int?) //Метод для сдачи экзамена  
 { if (subjects.contains(exam.subject))  
 { exam.status = Exam.ExamStatus.*CONDUCTED  
 println*("Student $name took the exam in ${exam.subject}.")  
 if (grade != null)  
 { grades[exam.subject] = grade  
 recordBook?.addRecord(name, group.hashCode(), exam.subject, grade)  
 *println*("Student $name took the exam in ${exam.subject} and received a grade of $grade.") }  
 else  
 { *println*("Student $name did not pass the exam in ${exam.subject}.") }  
 }  
 else  
 { *println*("Student $name is not enrolled in ${exam.subject}.") }  
 }  
 fun getExams(): List<Exam> = exams  
 fun viewRecordBook(): List<GradeBook.Record>?  
 { return recordBook?.getInfo() }  
}

fun main()  
{ val university = University("Белорусский государственный университет информатики и радиоэлектроники")  
 val student1 = Student("Lyi Grey", "12345", *listOf*("Math", "Phisics"), *mutableMapOf*())  
 university.enrollStudent(student1)  
 student1.viewRecordBook()  
 val lecturer = Lecturer("Dr. Smith", "Math")  
 val subjects = *listOf*("Math", "Physics")  
 val grades = *mutableMapOf*<String, Int>()  
 val student2 = Student("John Doe", "Group A34", subjects, grades)  
 val exam2 = Exam("Dark arts defence")  
 student2.takeExam(exam2, 3)  
 *println*(student2.viewRecordBook())  
 val exam1 = university.createExam("Math")  
 val departmenthead = DepartmentHead("Mr. Black")  
 departmenthead.scheduleExam(exam1, "2024-06-05")  
 lecturer.conductExam(exam1)  
 student1.takeExam(exam1, 10)  
 *println*(student1.viewRecordBook())  
 val engineer = Engineer("Fred Weasley", "101")  
 *println*("Engineer ${engineer.name} is assigned to room ${engineer.roomNumber}.")  
}

Результат работы программы представлен на рисунке 1

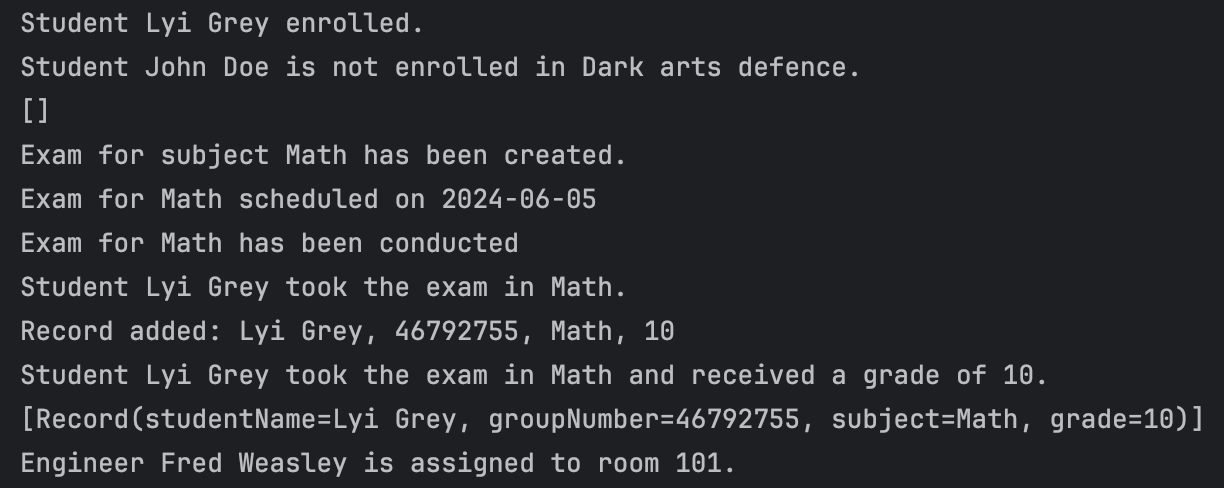


Рисунок 1 – Результат работы программы

**Ответы на вопросы к лабораторной работе:**

* 1. Что такое *JVM*?

JVM (Java Virtual Machine) – это виртуальная машина для запуска программ написанных на языках Java, Kotlin и др. Она управляет системной памятью и обеспечивает переносимую среду выполнения для приложения

1. Пакеты в *Kotlin*?

Используются для организации кода в логическую единицу и предотвращения конфликтов имен. В начале файда с помощью слова package.

1. Классы в *Kotlin*.

Объявляются с исп ключевого слова class. Класс может иметь первичный и вторичный конструкторы, свойства и методы.

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

Объявляются переменные с помощью var и val. Var исп для объявления изменяемых переменных. Val исп для объявления неизменяемых переменных.

1. Сколько конструкторов доступно в *Kotlin*?

Класс может иметь один первичный и один или несколько вторичных конструкторов.

1. Для чего служат абстрактные классы?

Абстрактный класс используется для определения общих характеристик, которые должны быть реализованы в подклассах.

1. Какой порядок вызова конструкторов при наследовании?

Сначала вызывается конструктор базового класса, затем конструктор производного класса.

1. Как переопределить метод родительского класса?

Передается с исп ключевого слова override.

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

Оператор «Elvis» (?:) исп для предоставления значения по умолчанию, если выражение слева от оператора равно null.

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

Лямбда-выражения объявляются с использованием синтаксиса {параметр -> тело}.

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

Они определяют видимость класса, методов и свойств. Public – доступен везде (по умолчанию), private – доступен только внутри файла или класса, protected – доступен в классе и его подклассах, internal – доступен внутри модуля.

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

Он позволяет сравнить значение переменной с набором возможных значений и выполнить данный блок кода. Может быть исп как выражение, возвращающее значение, или как оператор, выполняющий блоки кода. Может объединять несколько условий в одну ветку с помощью запятой. Поддерживает проверку значений на принадлежность диапазону или коллекции с помощью операторов in и !in. When не требует break для выхода из блока кода, т.к. выполнение автоматически прекращается. Может сравнить значение с результатом произвольных выражений.

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

Классы создаются с исп ключевого слова class, может содержать свойства (переменные) и методы. Объекты с исп ключевого слова var или val и вызова конструктора класса. Класс может иметь один первичный конструктор и один или несколько вторичных конструкторов. Вторичные конструкторы объявляются с исп клбчевого слова constructur. Инициализаторы исп для выполнения кода при. Создании объекта. Они объявляются с исп ключ слова init.

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

Inline-функция исп для оптимизации кода, уменьшая накладные расходы на вызов функций.

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

В блок try помещается код, который может вызвать исключение. Catch перехватывает и обрабатывает исключение если оно возникает.

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

Оператор in исп для проверки наличия элемента в коллекции. Возвращает true, если элемент присутствует в коллекции и false если нет.

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

Расширение позволяет добавлять новые функции и свойства к сущ кассам без необходимости наследования или исп паттернов. С помощью расширений (extensions).

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

Функциональные типы данных представляют собой функции и могут быть использованы как параметры или возвращаемые значения.

**Вывод**: Изучена работа языка *Kotlin* и выполнена разработка приложения.