Министерство образования Республики Беларусь

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: Суровый И. А.

Студент группы 310901

Преподаватель: Усенко Ф. В.

Минск 2024

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

**Вариант 27**: Добавить монстров Кабан и Оборотень. Организовать метод убийства монстра. Переделать работу классов так, чтобы монстры сообщали квесту о своём убийстве.

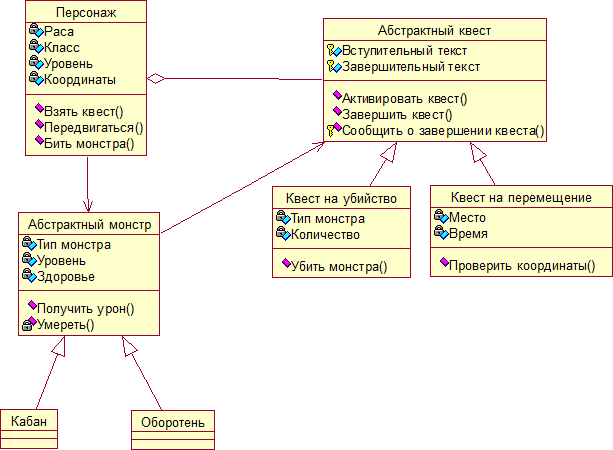


Рисунок 1 – Диаграмма классов

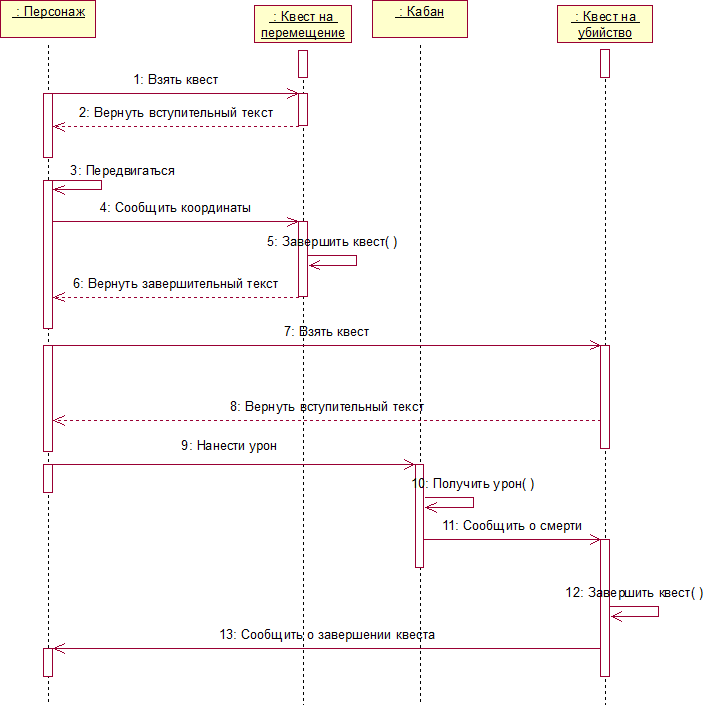


Рисунок 2 – Диаграмма последовательности

**Код программы:**

package main

import models.\*

fun main() {

println("Добро пожаловать в игру!\nПредставься путник: ")

val nameCharacter = readln()

println("В нашей игре 5 рас:\n1. Орк\n2. Полурослик\n3. Гном\n4. Эльф\n5. Тифлинг")

println("Выбери расу, $nameCharacter!")

var raceCharacter = "Гоблин"

var classCharacter = "Следопыт"

var choice = readln()

when (choice) {

"1" -> raceCharacter = "Орк"

"2" -> raceCharacter = "Полурослик"

"3" -> raceCharacter = "Гном"

"4" -> raceCharacter = "Эльф"

"5" -> raceCharacter = "Тифлинг"

else -> println("$nameCharacter, тот кто не играет по правилам, играет за гоблина!")

}

println("В нашей игре 5 классов:\n1. Воин\n2. Лучник\n3. Маг\n4. Чародей\n5. Плут")

println("Выбери класс, $nameCharacter!")

choice = readln()

when (choice) {

"1" -> classCharacter = "Воин"

"2" -> classCharacter = "Лучник"

"3" -> classCharacter = "Маг"

"4" -> classCharacter = "Чародей"

"5" -> classCharacter = "Плут"

else -> println("$nameCharacter, тот кто не играет по правилам, играет за следопыта!")

}

val character = Character(name = nameCharacter, race = raceCharacter, characterClass = classCharacter, level = 1, x = 0, y = 0, hp = 20)

character.reportInfo()

}

package models

class Character(

val name: String,

val race: String,

val characterClass: String,

var level: Int,

var x: Int,

var y: Int,

var hp: Int

) {

fun takeQuest(quest: Quest) {

println("Персонаж $race - $characterClass с $level LVL $hp HP взял квест: ${quest.introText}")

}

fun reportInfo(){

println("\n Имя - $name" +

"\n Раса - $race" +

"\n Класс - $characterClass" +

"\n $level LVL $hp HP" +

"\n Координаты - $x, $y"

)

}

fun move(newX: Int, newY: Int) {

x = newX

y = newY

println("Персонаж переместился на координаты ($x, $y)")

}

fun hitMonster(monster: Monster) {

println("Персонаж атакует монстра!")

monster.receiveDamage(10)

}

}

package models

class KillQuest(

introText: String,

completionText: String,

val monsterType: String,

val monsterCount: Int

) : Quest(introText, completionText) {

private var killedCount = 0

fun killMonster() {

if (killedCount < monsterCount) {

killedCount++

println("Убито монстров: $killedCount из $monsterCount")

}

}

override fun complete(): String {

return if (killedCount == monsterCount) {

completionText

} else {

"Квест не выполнен, нужно убить еще ${monsterCount - killedCount} монстров."

}

}

}

package models

abstract class Quest(

val introText: String,

val completionText: String

) {

abstract fun complete(): String

}

package models

abstract class Monster(

val type: String,

val level: Int,

var health: Int

) {

abstract fun receiveDamage(damage: Int)

fun die() {

println("$type умер.")

}

}

package models

class Boar(level: Int, health: Int) : Monster("Кабан", level, health) {

override fun receiveDamage(damage: Int) {

health -= damage

if (health <= 0) {

die()

} else {

println("Кабан получил урон, осталось здоровья: $health")

}

}

}

package models

class Werewolf(level: Int, health: Int) : Monster("Оборотень", level, health) {

override fun receiveDamage(damage: Int) {

health -= damage

if (health <= 0) {

die()

} else {

println("Оборотень получил урон, осталось здоровья: $health")

}

}

}

package actions

import models.Character

import models.Quest

object QuestActions {

fun startQuest(character: Character, quest: Quest) {

character.takeQuest(quest)

println("Квест: ${quest.introText}")

}

fun completeQuest(quest: Quest): String {

return quest.complete()

}

}

package actions

import models.Character

object MovementActions {

fun moveCharacter(character: Character, newX: Int, newY: Int) {

character.move(newX, newY)

}

fun reportCoordinates(character: Character) {

println("Текущие координаты персонажа: (${character.x}, ${character.y})")

}

}

package actions

import models.Character

import models.Monster

object BattleActions {

fun attackMonster(character: Character, monster: Monster) {

character.hitMonster(monster)

}

fun reportMonsterDeath(monster: Monster) {

if (monster.health <= 0) {

monster.die()

} else {

println("Монстр всё ещё жив с ${monster.health} HP.")

}

}

}

Результат работы программы представлен на рисунке 1

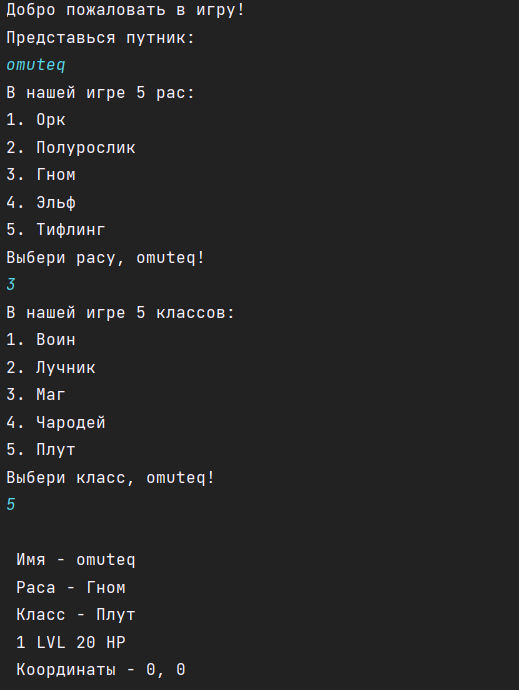


Рисунок 1 – Результат работы программы

**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 файла 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}. Предназначен для определения функциональных выражений.

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.