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: Чиж А. В.

Студент группы 310901

Преподаватель: Усенко Ф. В.

Минск 2024

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

**Вариант 2**: Добавить класс Комплексное число в показательном виде, организовать взаимодействие с классом Комплексное число в алгебраическом виде. Реализовать действие умножения и деления. Реализовать класс Человек, управляющий классом Выражение.

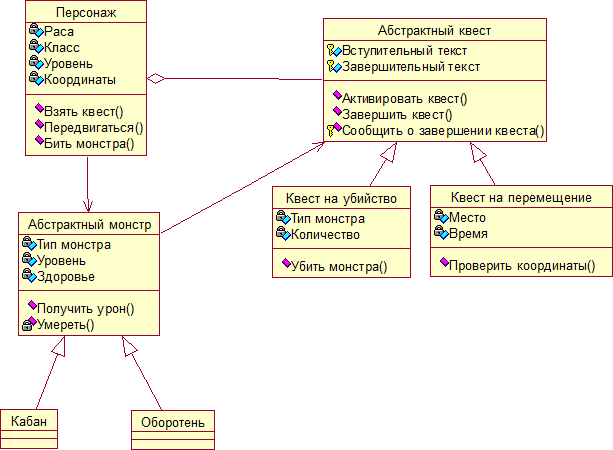


Рисунок 1 – Диаграмма классов

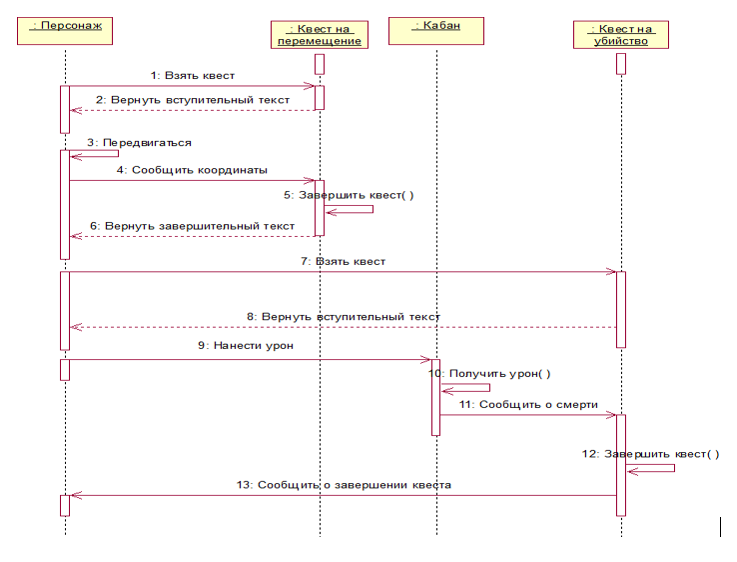


Рисунок 2 – Диаграмма последовательности

**Код программы:**

import service.User

fun main() {

val user: User = User();

println(user.doOp());

}

package service

import entity.Expression;

import entity.Number

class User() {

private var name: String;

init {

print("Ваше имя: ");

this.name = readLine()!!;

}

fun doOp(): Number? {

val exp: Expression = Expression();

exp.addNum();

exp.addNum();

val result: Number? = exp.operate();

return result;

}

}

package entity

class Expression {

var list: ArrayList<Number> = arrayListOf();

companion object {

enum class Operation {

Add,

Subtract,

Multiply,

Divide,

}

}

fun addNum() {

println("Выберите тип комплесного числа:\n1. В алгебраическом виде\n2. В показательном виде");

val i: Int = readLine()!!.toInt();

val type: Number.Companion.Type = Number.Companion.Type.entries[i - 1];

val num: Number;

if (type.equals(Number.Companion.Type.Algo)) {

print("Введите действительную часть: ");

val real: Double = readLine()!!.toDouble();

print("Введите мнимую часть: ");

val imaginary: Double = readLine()!!.toDouble();

num = Number(type, real, imaginary);

} else {

print("Введите модуль: ");

val mod: Double = readLine()!!.toDouble();

print("Введтье фазу: ");

val phase: Double = readLine()!!.toDouble();

num = Number(type, mod, phase);

}

list.add(num);

}

fun operate(): Number? {

println("Выберите операцию:\n1. Сложение\n2. Вычитание\n3. Умножение\n4. Деление");

val i: Int = readLine()!!.toInt();

val op: Operation = Operation.entries[i - 1];

val result: Number = list[0];

try {

for (i in 1..(list.size - 1)) {

when (op) {

Operation.Add -> result.value.add(list[i].value)

Operation.Subtract -> result.value.subtract(list[i].value)

Operation.Multiply -> result.value.multiply(list[i].value)

Operation.Divide -> result.value.divide(list[i].value)

}

}

} catch (e: IllegalArgumentException) {

println("Divide by zero!");

return null;

}

return result;

}

}

package entity

import complexInt.AlgoComplex

import complexInt.Complex

import complexInt.IndiComplex

class Number(type: Type, value1: Double, value2: Double) {

var value: Complex;

companion object {

enum class Type {

Algo,

Indi

}

}

init {

if (type.equals(Type.Algo)) {

value = AlgoComplex(value1, value2);

} else {

value = IndiComplex(value1, value2);

}

}

public override fun toString(): String {

return value.toString();

}

}

package complexInt

import kotlin.math.atan

import kotlin.math.pow

import kotlin.math.sqrt

class AlgoComplex(real: Double = 0.0, imaginary: Double = 0.0): Complex() {

private var real: Double;

private var imaginary: Double;

init {

this.real = real;

this.imaginary = imaginary;

}

public fun getReal(): Double {

return this.real;

}

public fun setReal(real: Double) {

this.real = real;

}

public fun getImaginary(): Double {

return this.imaginary;

}

public fun setImaginary(imaginary: Double) {

this.imaginary = imaginary;

}

internal fun toIndi(): IndiComplex {

val mod: Double = sqrt(this.real.pow(2) + this.imaginary.pow(2));

val phase: Double = atan(this.imaginary/this.real);

return IndiComplex(mod, phase);

}

public override fun add(num: Complex) {

if (num is AlgoComplex) {

this.real += num.getReal();

this.imaginary += num.getImaginary();

} else if (num is IndiComplex) {

add(num.toAlgo());

}

}

public override fun subtract(num: Complex) {

if (num is AlgoComplex) {

this.real -= num.getReal();

this.imaginary -= num.getImaginary();

} else if (num is IndiComplex) {

add(num.toAlgo());

}

}

public override fun multiply(num: Complex) {

if (num is AlgoComplex) {

val r: Double = num.getReal();

val i: Double = num.getImaginary();

this.real = (this.real \* r - this.imaginary \* i);

this.imaginary = (this.real \* i + this.imaginary \* r);

} else if (num is IndiComplex) {

multiply(num.toAlgo());

}

}

public override fun divide(num: Complex) {

if (num is AlgoComplex) {

val r: Double = num.getReal();

val i: Double = num.getImaginary();

if (r == 0.0 && i == 0.0) {

throw IllegalArgumentException();

}

this.real = (this.real \* r + this.imaginary \* i) / (r.pow(2) + i.pow(2));

this.imaginary = (r \* this.imaginary - this.real \* i) / (r.pow(2) + i.pow(2));

} else if (num is IndiComplex) {

divide(num.toAlgo());

}

}

public override fun toString(): String {

if (this.imaginary >= 0) {

return "${this.real} + ${this.imaginary}i";

} else {

return "${this.real} + (${this.imaginary})i";

}

}

}

package complexInt

import kotlin.math.cos

import kotlin.math.sin

class IndiComplex(mod: Double = 0.0, phase: Double = 0.0): Complex() {

private var mod: Double;

private var phase: Double;

init {

this.mod = mod;

this.phase = phase;

}

public fun getMod(): Double {

return this.mod;

}

public fun setMod(mod: Double) {

this.mod = mod;

}

public fun getPhase(): Double {

return this.phase;

}

public fun setPhase(phase: Double) {

this.phase = phase;

}

internal fun toAlgo(): AlgoComplex {

val real: Double = this.mod \* cos(this.phase);

val imaginary: Double = this.mod \* sin(this.phase);

return AlgoComplex(real, imaginary);

}

public override fun add(num: Complex) {

val algo: AlgoComplex = this.toAlgo();

algo.add(num);

val indi: IndiComplex = algo.toIndi();

this.mod = indi.getMod();

this.phase = indi.getPhase();

}

public override fun subtract(num: Complex) {

val algo: AlgoComplex = this.toAlgo();

algo.subtract(algo);

val indi: IndiComplex = algo.toIndi();

this.mod = indi.getMod();

this.phase = indi.getPhase();

}

public override fun multiply(num: Complex) {

if (num is IndiComplex) {

this.mod \*= num.getMod();

this.phase += num.getPhase();

} else if (num is AlgoComplex) {

val indi: IndiComplex = num.toIndi();

this.mod \*= indi.getMod();

this.phase += indi.getPhase();

}

}

public override fun divide(num: Complex) {

if (num is IndiComplex) {

this.mod /= num.getMod();

this.phase -= num.getPhase();

} else if (num is AlgoComplex) {

val indi: IndiComplex = num.toIndi();

this.mod /= indi.getMod();

this.phase -= indi.getPhase();

}

}

public override fun toString(): String {

if (this.phase >= 0) {

return "|${this.mod}|^i${this.phase}";

} else {

return "|${this.mod}|^i(${this.phase})";

}

}

}

package complexInt

public abstract class Complex {

abstract fun add(num: Complex);

abstract fun subtract(num: Complex);

abstract fun multiply(num: Complex);

abstract fun divide(num: Complex);

}

Результат работы программы представлен на рисунке 1

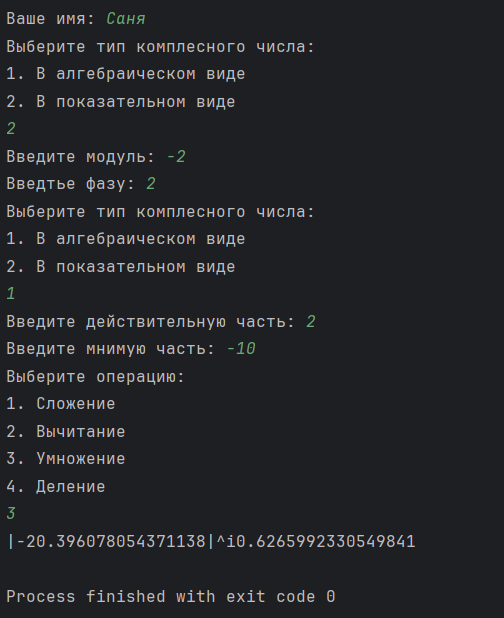


Рисунок 1 – Результат работы программы

**Ответы на вопросы к лабораторной работе:**

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

В Kotlin переменные объявляются с использованием ключевых слов var и val. Val используется для объявления неизменяемых переменных.

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

Оператор «Elvis» в Kotlin обозначается символом ?: и используется для обработки значений, которые могут быть равны null. Он позволяет задать значение по умолчанию, которое будет возвращено, если выражение слева от оператора Elvis равно null.

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

В Kotlin модификаторы доступа (access modifiers) определяют уровень видимости классов, объектов, функций и свойств. Они помогают контролировать, какие части кода могут взаимодействовать друг с другом.

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

Inline-функции в Kotlin — это функции, которые компилятор может встраивать (или «инлайнить») в место их вызова, вместо того чтобы создавать отдельный вызов функции в стеке. Это может привести к повышению производительности и уменьшению накладных расходов на вызов функции, особенно для небольших функций или функций высшего порядка.

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

Функциональные типы данных (function types) в Kotlin — это типы, которые представляют функции как объекты. Они позволяют передавать функции как аргументы, возвращать их из других функций и хранить в переменных. Это делает Kotlin языком с поддержкой функционального программирования.

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