Учреждение образования Федерации профсоюзов Беларуси
«Международный университет «МИТСО»

Факультет экономический
Кафедра информационных технологий

СОГЛАСОВАНО
Заведующий кафедрой
_____О.Б.Хорошко
27.02. 2026 г.

СОГЛАСОВАНО
Декан факультета
_____А.В.Ковтунов
13.04 2026 г.

ЭЛЕКТРОННЫЙ УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

для специальности 6-05-0414-04 «Управление информационными ресурсами»

Составитель: Пархимович А.В., старший преподаватель кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»

Рассмотрено и рекомендовано к утверждению на заседании кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
27.02. 2026 г., протокол № 7

Утверждено на заседании научно-методического совета учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
28.05. 2026 г., протокол № 5

РЕЦЕНЗЕНТЫ:
Кафедра микропроцессорных систем и сетей УО «Белорусский государственный университет информатики и радиоэлектроники»;
Т.Н.Беляцкая, заведующий кафедрой менеджмента УО «Белорусский государственный университет информатики и радиоэлектроники», доктор экономических наук, профессор.

Регистрационный № УД-064-26/7

Регистрационное свидетельство № 1202335558 от 15.08.2023
новый ЭУМК, как актуализ.

АКТУАЛИЗИРОВАН
Рассмотрено на заседании кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
_____202__ г., протокол № ___

Утверждено на заседании научно-методического совета учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
_____202__ г., протокол № ___

ОГЛАВЛЕНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Электронный учебно-методический комплекс (ЭУМК) подготовлен в соответствии с учебной программой Учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО» по дисциплине «Технологии программирования» для специальности 6-05-0414-04 «Управление информационными ресурсами».

Цель ЭУМК – обеспечение максимальной доступности и системности изучаемого материала по дисциплине, формирование у студентов специализированных компетенций в области современных технологий разработки программного обеспечения, включая проектирование, реализацию и сопровождение программных продуктов для платформы Android. Практическая значимость состоит в том, что данный ЭУМК позволяет структурировать учебную деятельность студентов, освоить методы организации работы коллективов программистов, принципы объектно-ориентированного и функционального программирования на языке Kotlin, а также подходы к тестированию и обеспечению качества ПО.

Основными структурными элементами ЭУМК являются:

Учебная программа по дисциплине, составленная на основе образовательного стандарта ОСВО 6-05-0414-04-2023.

Теоретический раздел, включающий конспект лекций для теоретического изучения. Содержание разделов охватывает основы разработки под Android, язык Kotlin, современные библиотеки (Jetpack Compose), архитектурные шаблоны (MVVM, Clean Architecture), а также вопросы работы с базами данных и сетевыми запросами.

Практический раздел, содержащий учебно-методические материалы для проведения лабораторных занятий. Темы работ включают освоение основ языка Kotlin, создание пользовательского интерфейса с помощью Jetpack Compose, работу с визуальными компонентами и контейнерами компоновки, а также реализацию взаимодействия с базами данных и сетью.

Раздел контроля знаний, в котором представлены вопросы для устных опросов (УО), темы докладов (УД), задания для решения практических задач (РПЗ), примерный перечень вопросов к экзамену и курсовой работе, а также темы рефератов (Р) и задания для управляемой самостоятельной работы (УСР).

Вспомогательный раздел, в котором приведены рекомендации по освоению дисциплины, перечень основной и дополнительной литературы (включая издания по языкам Java и C++), а также перечень необходимых инструментальных средств (Android Studio, Visual Studio Code).

Рекомендации по организации работы с ЭУМК.

Для лучшего усвоения материала студентам рекомендуется сначала ознакомиться с учебной программой по дисциплине «Технологии программирования». Усвоение материала по каждой из тем рекомендуется начинать с изучения соответствующей лекции («Теоретический раздел»), после чего следует ответить на вопросы для самоконтроля и выполнить задания лабораторных работ («Практический раздел»). Проверка знаний проводится путем решения практических задач, защиты докладов и ответов на вопросы из перечня для промежуточной аттестации («Раздел контроля знаний»). Студентам рекомендуется использовать материалы ЭУМК для осуществления активной самостоятельной работы при изучении дисциплины, подготовки к аудиторным занятиям и сдаче экзамена и курсовой работы.

Учреждение образования Федерации профсоюзов Беларуси
«Международный университет «МИТСО»

УТВЕРЖДАЮ
Проректор по учебной работе
учреждения образования
Федерации профсоюзов Беларуси
«Международный университет «МИТСО»
_____ М.А.Юрочкин
20.12. 2024 г.
Регистрационный №УД -082/02-24/уч.

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Учебная программа учреждения высшего образования по учебной дисциплине для специальности

6-05-0414-04 Управление информационными ресурсами

2024 г.

Контрольный экземпляр

Учебная программа составлена на основе образовательного стандарта ОСВО 6-05-0414-04-2023 для специальности 6-05-0414-04 «Управление информационными ресурсами», утвержденного Постановлением Министерства образования Республики Беларусь 07.08.2023 № 240 и учебного плана учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО» для специальности 6-05-0414-04 «Управление информационными ресурсами», утвержденного 01.03.2023 регистрац. № 6-05-0414-04/уч.

СОСТАВИТЕЛЬ:
М.А.Калинин, старший преподаватель кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»

РЕКОМЕНДОВАНА К УТВЕРЖДЕНИЮ:
Кафедрой информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
(протокол № 4 от 28.11.2024);

Научно-методическим советом учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
(протокол № 3 от 20.12.2024)

СОГЛАСОВАНО
Декан экономического факультета
_____А.В.Ковтунов

Нормоконтроль
ведущий специалист УМУ
_____Г.Д.Лагунович

Библиограф I категории
_____Н.А.Лёпа

I. ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Учебная дисциплина «Технологии программирования» входит составной частью в систему дисциплин, обеспечивающих подготовку специалистов по специальности 6-05-0414-04 «Управление информационными ресурсами», относится к циклу дисциплин модуля «Разработка программного обеспечения» и изучается в тесной взаимосвязи с учебной дисциплиной «Алгоритмизация и программирование».

Целью изучения учебной дисциплины «Технологии программирования» является теоретическая и практическая подготовка студентов в области технологий разработки программ в такой степени, чтобы при менеджменте программного проекта или в процессе участия в его реализации они могли выбирать необходимые технические, алгоритмические, программные и технологические решения, уметь объяснить принципы их функционирования и правильно их использовать. Иметь представление о каждом этапе жизненного цикла программы от проектирования до внедрения и сопровождения. Знать современные стандарты качества программного обеспечения (далее – ПО) и перспективные направления развития технологии разработки ПО.

Основными задачами учебной дисциплины являются:

изложение основных положений технологии разработки ПО, формулировка практических рекомендаций по организации работы коллективов программистов, руководства такими коллективами;

формирование у студентов знаний по дисциплине, связанных с процессом разработки ПО, включая связи с предметной областью, реализацию, организацию производства, контроль сроков исполнения и качества;

ознакомление с техническими программными и технологическими решениями, используемыми при разработке ПО;

приобретение практических навыков работы в коллективе программистов, умения находить правильные технологические решения по выбору структуры программного проекта, методов тестирования и контроля исполнения использование современных инструментальных и методологических средств.

Освоение учебной дисциплины «Технологии программирования» должно обеспечить формирование **специализированных компетенций**: разрабатывать алгоритмы и исходные коды программных приложений, используемых в сферах экономики и управления.

В результате изучения дисциплины, обучающиеся должны

знать:

классические и современные модели процесса разработки программных систем;

принципы, способы и методы разработки и поддержки программных продуктов;

методы организации и управления процессом разработки ПО;

основные способы и методы оценки качества продукта;

принципы работы с различными инструментами для поддержки процесса разработки программных систем;

уметь:

моделировать программные системы;

проводить анализ созданных моделей;

определять основные качественные характеристики программ;

получать качественные метрики программного продукта;

пользоваться современными CASE-средствами для поддержки процесса разработки и тестирования ПО;

иметь навыки:

применения основных методов анализа и проектирования ПО;

владения методами оценки качества и надежности ПО;

самообразования и способами использования приемов и методов тестирования в процессе разработки программных продуктов.

Распределение аудиторных часов по видам занятий и семестрам.
Формы промежуточной аттестации

Семестр	Количество академических часов							Самостоят. работа	Форма промежуточной аттестации
	Всего	Аудит.	Из них						
			Лекции	Лабор. занятия	Практ. занятия	Семи-нары	УСР[1]		
Очная форма получения высшего образования									
4	120	60	24	30			6	60	курс.р.; экз.
Всего	120	60	24	30			6	60	

[1] Управляемая самостоятельная работа

II. СОДЕРЖАНИЕ УЧЕБНОЙ ДИСЦИПЛИНЫ

Тема 1. Основы Android Development

Инструменты для разработки мобильных приложений. Нативные и кроссплатформенные инструменты разработки. Проектирование мобильного приложения, разработка документации. Операционная система Android и принципы взаимодействия с ней программ. Структура приложения Android. Стандартизированный графический интерфейс пользователя. Многозадачность. Аппаратная независимость. Установка среды разработки Android Studio.

Тема 2. Основы языка Kotlin

Основы языка Kotlin. Функциональное программирование. Объектно-ориентированное программирование. Обобщения. Дополнительные возможности ООП. Коллекции и последовательности. Корутины. Асинхронные потоки.

Тема 3. Библиотека Jetpack Compose

Jetpack Compose – современный набор инструментов Android для создания собственного пользовательского интерфейса на языке программирования Kotlin. Основные компоненты, которые предоставляет Jetpack Compose. Компонуемые функции – основные строительные блоки Jetpack Compose.

Тема 4. Модификаторы и визуальный интерфейс

Модификаторы. Установка цвета. Установка размеров. Установка отступов и смещения. Создание прокрутки. Обработка нажатий.

Тема 5. Контейнеры компоновки

Box. Column. Row.

Тема 6. Состояние компонентов

Введение в состояние компонентов.

Тема 7. Визуальные компоненты

Text. Кнопка Button. Ввод текста, TextField и OutlinedTextField. Модификатор Modifier.toggleable. Checkbox. Выбираемый компонент и модификатор selectable. RadioButton. Иконки и компоненты IconButton и IconToggleButton. FloatingActionButton и ExtendedFloatingActionButton. Панели приложения TopAppBar и BottomAppBar. Scaffold. Всплывающие сообщения и Snackbar. Выдвижная панель drawer в Scaffold. Slider. Переключатель Switch. Диалоговые окна AlertDialog. Меню DropdownMenu. Индикаторы прогресса.

Тема 8. Ресурсы в Jetpack Compose

Ресурсы строк. Ресурсы dimension. Ресурсы Color.

Тема 9. Работа с изображениями

Компонент Image. Ресурсы изображений и ImageBitmap. Векторная графика и ImageVector.

Тема 10. Архитектурные шаблоны и паттерны

Архитектурные шаблоны и паттерны (Model-View-ViewModel (MVVM), Clean Architecture) и их применение при разработке масштабируемых и поддерживаемых Android-приложений.

Тема 11. Работа с базами данных

Работа с базами данных на платформе Android. Использование SQLite для создания и управления локальными базами данных.

Тема 12. Работа с сетью

Методы взаимодействия с сетью на платформе Android. Отправка и получение данных посредством HTTP-запросов, работа с API, обработка данных в форматах JSON или XML.

Тема 13. Развертывание и публикация приложений

Процесс развертывания и публикации Android-приложений в Google Play Store. Подготовка приложения к релизу и создание подписи приложения.

III. ТРЕБОВАНИЯ К КУРСОВОЙ РАБОТЕ

В процессе изучения учебной дисциплины «Технологии программирования» предусмотрено выполнение курсовой работы.

Целью выполнения курсовой работы является закрепление полученных знаний в области технологий разработки программ, выборе необходимых технических, алгоритмических, программных и технологических решений.

На выполнение курсовой работы отводится 40 часов.

Выполнение курсовой работы осуществляется в соответствии с Методическими рекомендациями по подготовке, оформлению и защите курсовых, дипломных работ и магистерских диссертаций утвержденными Приказом ректора университета от 08.02.2024 № 59.

В интересах систематизации действий обучающихся по выполнению задания на курсовую работу, на кафедре разрабатываются методические рекомендации по выполнению курсовой работы по дисциплине «Технологии программирования».

Законченная и полностью оформленная работа в установленные сроки представляется руководителю курсовой работы для проверки и предварительной оценки.

Защита курсовой работы осуществляется согласно расписанию занятий.

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ТЕМ КУРСОВЫХ РАБОТ

1. Разработка приложения для учета финансов.
2. Создание приложения для планирования и управления задачами.
3. Разработка приложения для фото-редактирования.
4. Создание мобильного клиента для онлайн-магазина.
5. Разработка приложения для трекинга физической активности и здоровья.
6. Создание приложения для чтения и управления электронными книгами.
7. Разработка мобильного приложения для составления и сохранения списков покупок.
8. Создание приложения для поиска и бронирования гостиниц.
9. Разработка приложения для изучения иностранных языков.
10. Создание мобильного приложения для обмена сообщениями и чатов.
11. Разработка приложения для ведения дневника и личного органайзера.
12. Создание приложения для просмотра и потоковой передачи видео.
13. Разработка мобильного приложения для поиска и оценки ресторанов.
14. Создание приложения для воспроизведения музыки и создания плейлистов.
15. Разработка приложения для создания и редактирования заметок и записей.
16. Создание мобильного клиента для социальной сети.

17. Разработка приложения для управления и контроля финансовых инвестиций.
18. Создание приложения для просмотра новостей и статей.
19. Разработка мобильного приложения для подкастов и аудио-контента.
20. Создание приложения для планирования и организации путешествий.
21. Разработка приложения для создания и редактирования графических иллюстраций.
22. Создание мобильного клиента для музыкального стримингового сервиса.
23. Разработка приложения для учета и контроля калорийного питания.
24. Создание приложения для виртуальной реальности (VR).
25. Разработка мобильного приложения для поиска и бронирования билетов на мероприятия.
26. Создание приложения для проектирования и создания пользовательских интерфейсов.
27. Разработка приложения для отслеживания и контроля сна и сновидений.
28. Создание мобильного клиента для музыкального инструментария и создания музыки.
29. Разработка приложения для управления смарт-домом и умными устройствами.
30. Разработка приложения для организации и управления личными финансами.
31. Создание мобильного клиента для онлайн-рынка недвижимости.
32. Разработка приложения для тренировок и фитнеса дома.
33. Создание приложения для обмена и продажи товаров между пользователями.
34. Разработка мобильного приложения для поддержки ментального здоровья и благополучия.
35. Создание приложения для виртуальных экскурсий и путешествий.
36. Разработка приложения для управления и контроля времени работы и отдыха.
37. Создание мобильного клиента для сервиса онлайн-обучения и курсов.
38. Разработка приложения для организации и планирования свадеб и мероприятий.
39. Создание приложения для поиска и рекомендации культурных мероприятий (концерты, выставки, театры и т.д.).

IV. УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ

Очная (дневная) форма получения высшего образования

Номер раздела, темы	Название раздела, темы	Всего часов	Количество аудиторных часов					Самостоятельная работа	Литература	Форма контроля
			Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия	Количество часов УСР			
1	2	3	4	5	6	7	8	9	10	11
4 семестр										
1	Основы Android Development	7	2				1	4	[1, 2]	УО, Р
2	Основы языка Kotlin	12	2			4	-	6	[1, 2]	УО, РПЗ
3	Библиотека Jetpack Compose	9	2			2	1	4	[1, 2]	УО, РПЗ, Р(ТА)[2]
4	Модификаторы и визуальный интерфейс	8	2			2	-	4	[1, 2]	УО, РПЗ
5	Контейнеры компоновки	8	2			2	-	4	[1, 2]	УО, РПЗ
6	Состояние компонентов	6	-			2	-	4	[1, 2]	УО, РПЗ
7	Визуальные компоненты	14	2			6	-	6	[1, 2]	УО, РПЗ
8	Ресурсы в Jetpack Compose	8	2			2	-	4	[1, 2]	УО, РПЗ
9	Работа с изображениями	8	2			2	-	4	[1, 2]	УО, РПЗ
10	Архитектурные шаблоны и паттерны	9	2			2	1	4	[1, 2]	УО, РПЗ, Р(ТА)
11	Работа с базами данных	13	2			4	1	6	[1, 2]	УО, РПЗ, Р
12	Работа с сетью	11	2			2	1	6	[1, 2]	УО, Р
13	Развертывание и публикация приложений	7	2				1	4	[1, 2]	УО, Р
	Всего по дисциплине	**120**	**24**			**30**	**6**	**60**		**экз.**

[2] Текущая аттестация

V. ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

ОСНОВНАЯ ЛИТЕРАТУРА

1. Эккель, Б. Философия Java / Эккель Брюс. – 4-е полное изд. – Спб. : Питер, 2018 – 1168 с.: ил.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

2. Прата, С. Язык программирования С++. Лекции и упражнения : пер. с англ. / Стивен Прата. – 6-е изд. – М. : ООО “И.Д. Вильямс”, 2018. – 1248 с.

СРЕДСТВА ОБУЧЕНИЯ

Необходимыми средствами для реализации успешного формирования профессиональных компетенций у студентов являются:

оборудованный интернет-доступом компьютерный класс с установленным специальным программным обеспечением;

мультимедийная аппаратура;

электронные учебно-наглядные пособия (презентации, методические материалы).

СРЕДСТВА ДИАГНОСТИКИ РЕЗУЛЬТАТОВ УЧЕБНОЙ ДЕЯТЕЛЬНОСТИ

Диагностика результатов образовательной деятельности обучающихся осуществляется в ходе проведения всех видов занятий, самостоятельной работы и текущей аттестации по учебной дисциплине.

Основными формами контроля знаний по учебной дисциплине являются:

устный опрос	УО;
устный доклад	УД;
реферат	Р;
решение практических задач	РПЗ.

Текущая аттестация обучающихся проводится в течение семестра в целях периодического контроля и оценки результатов их учебной деятельности по учебной дисциплине с выставлением отметок.

Текущий контроль проводится строго в соответствии с учебно-методической картой дисциплины. Обучающийся должен в обязательном порядке участвовать во всех контрольных мероприятиях текущего контроля, предусмотренных учебной программой дисциплины.

Форма текущей аттестации по учебной дисциплине:

реферат	Р.

К промежуточной аттестации по учебной дисциплине, модулю обучающиеся допускаются при условии успешного прохождения текущей аттестации, предусмотренной учебной программой в текущем семестре.

Формами промежуточной аттестации являются:

курсовая работа	курс.р.;
экзамен	экз.

Оценка уровня знаний студента производится по десятибалльной шкале в соответствии с критериями, утвержденными Министерством образования Республики Беларусь.

ПЕРЕЧЕНЬ ТЕМ ЛАБОРАТОРНЫХ ЗАНЯТИЙ

Основная цель проведения лабораторных занятий состоит в закреплении теоретического материала курса, приобретении навыков выполнения эксперимента, обработки экспериментальных данных, анализа результатов, грамотного оформления отчетов.

Тема 2. Основы языка Kotlin
Лабораторная работа 1. Основы языка Kotlin
Обеспеченность занятия: ПК, Visual Studio Code.

Тема 3. Библиотека Jetpack Compose
Лабораторная работа 2. Библиотека Jetpack Compose
Обеспеченность занятия: ПК, Visual Studio Code.

Тема 4. Модификаторы и визуальный интерфейс
Лабораторная работа 3. Модификаторы и визуальный интерфейс
Обеспеченность занятия: ПК, Visual Studio Code.

Тема 5. Контейнеры компоновки
Лабораторная работа 4. Контейнеры компоновки
Обеспеченность занятия: ПК, Visual Studio Code.

Тема 6. Состояние компонентов
Лабораторная работа 5. Состояние компонентов
Обеспеченность занятия: ПК, Visual Studio Code.

Тема 7. Визуальные компоненты
Лабораторная работа 6. Визуальные компоненты
Обеспеченность занятия: ПК, Visual Studio Code.

Тема 8. Ресурсы в Jetpack Compose
Лабораторная работа 7. Ресурсы в Jetpack Compose
Обеспеченность занятия: ПК, Visual Studio Code.

Тема 9. Работа с изображениями
Лабораторная работа 8. Работа с изображениями
Обеспеченность занятия: ПК, Visual Studio Code.

Тема 10. Архитектурные шаблоны и паттерны
Лабораторная работа 9. Архитектурные шаблоны и паттерны
Обеспеченность занятия: ПК, Visual Studio Code.

Тема 11. Работа с базами данных
Лабораторная работа 10. Работа с базами данных
Обеспеченность занятия: ПК, Visual Studio Code.

Тема 12. Работа с сетью
Лабораторная работа 11. Работа с сетью
Обеспеченность занятия: ПК, Visual Studio Code.

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ЗАДАНИЙ И КОНТРОЛЬНЫХ МЕРОПРИЯТИЙ УСР

№ темы, (раздела)	Тема УСР	Кол-во часов	Метод. обеспечение	Форма контроля
2 курс, 4 семестр (6 часов)				
1.	Основы Android Development	1	Интернет-ресурсы	Реферат, доклад, сообщение
3.	Библиотека Jetpack Compose	1	Интернет-ресурсы	Реферат, доклад, сообщение
10.	Архитектурные шаблоны и паттерны	1	Интернет-ресурсы	Реферат, доклад, сообщение
11.	Работа с базами данных	1	Интернет-ресурсы	Реферат, доклад, сообщение
12.	Работа с сетью	1	Интернет-ресурсы	Реферат, доклад, сообщение
13.	Развертывание и публикация приложений	1	Интернет-ресурсы	Реферат, доклад, сообщение

Тематика рефератов

1. Основы работы с Compose: элементы управления, композиция, обработка событий и связывание данных.
2. Разработка интерфейса пользователя с использованием Compose: макеты, стили и темы.
3. Работа с данными и асинхронностью в Compose: использование Flow и State.
4. Разработка приложений с использованием сети, баз данных и медиаконтента.
5. Тестирование и отладка приложений на Compose.
6. Публикация и распространение приложений на Google Play.

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ВОПРОСОВ ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ (ЭКЗАМЕН)

1. Основные языки разработки под Android.
2. История развития и достоинства языка Kotlin.
3. Сборщики проектов. Gradle.
4. Основные элементы языка Kotlin: Типы данных. Переменные и функции.
5. Основные элементы языка Kotlin: Перечисления. Классы и свойства.
6. Основные элементы языка Kotlin: Управляющие конструкции.
7. Исключения.
8. Создание и использование коллекций.
9. Использование регулярных выражений.
10. Понятие анонимной функции и лямбда-выражения. Синтаксис лямбда выражений.
11. Лямбда-выражения и коллекции.
12. Перегрузка арифметических операторов.
13. Перегрузка логических операторов.
14. Перегрузка операторов для работы с коллекциями.
15. Понятие корутин. Преимущества использования корутин по сравнению с потоками. Жизненный цикл корутин.
16. Декларирование и применение аннотаций. Мета-аннотации. Классы в качестве параметров аннотации.
17. Классы для упрощения работы с базами данных.
18. Создание базы данных и таблиц.
19. Получение, модификация и запись данных в базу данных.
20. Структура Android проекта.
21. Структура Android проекта. Android Manifest.
22. Структура Android проекта. Ресурсы Android приложения.
23. Разработка UI Android приложения.
24. Меню Android приложения. Создание меню параметров.
25. Меню Android приложения. Создание контекстного меню.
26. Меню Android приложения. Создание всплывающего меню.
27. Меню Android приложения. Создание групп меню.
28. Жизненный цикл Activity.
29. Элементы экрана и их свойства.
30. Отладка Android приложений. LogCat.
31. Обработка исключений в Android.
32. База данных SQLite. SQLite OpenHelper.
33. База данных SQLite. Методы UPDATE и DELETE.
34. База данных SQLite. Метод QUERY.
35. База данных SQLite. Content provider (Контент-провайдер)
36. JSON в Android. Структура данных в JSON-формате.

VI. ПРОТОКОЛ СОГЛАСОВАНИЯ УЧЕБНОЙ ПРОГРАММЫ

Название учебной дисциплины, с которой требуется согласование	Название кафедры	Предложения об изменениях в содержании учебной программы учреждения высшего образования по учебной дисциплине	Решение, принятое кафедрой, разработавшей учебную программу (с указанием даты и номера протокола)
Алгоритмизация и программирование	информационных технологий	предложений нет	протокол заседания кафедры № 4 от 28.11.2024

I. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

ОСНОВЫ ANDROID DEVELOPMENT

Экосистема Android представляет собой многоуровневую структуру, включающую операционную систему, аппаратное обеспечение, инструменты разработки и каналы распространения приложений. Понимание взаимосвязей между этими элементами позволяет принимать обоснованные технические решения на этапе проектирования и избегать ошибок, связанных с несовместимостью или неоптимальным использованием ресурсов.

Под экосистемой Android в широком смысле понимается совокупность аппаратных устройств (смартфонов, планшетов, носимой электроники), операционной системы (AOSP − Android Open Source Project), сервисов Google (Google Mobile Services, GMS), а также инструментов разработки и магазинов приложений. Для разработчика экосистема − это границы возможного: то, какие функции устройства доступны через API, какие ограничения накладывает система и каковы ожидания пользователей.

Система Android основана на ядре linux, который предлагает основные функции управление памятью, безопасность и контроллеры. Над ним находится Уровень аппаратной абстракции (HAL), что позволяет приложению единообразно взаимодействовать с оборудованием тысяч различных устройств.

Над HAL живет рамки приложения с его API, где приложения взаимодействуют с камерой, GPS, датчики или уведомления. Менеджер пакетов управляет установкой, обновлением и разрешениями приложений, в то время как несколько системные службы (синхронизация, питание, уведомления и т.д.) работают в фоновом режиме. Эта модульная, открытая и расширяемая архитектура позволяет легко адаптировать Android к широкому выбор устройств, при этом умножая возможные сценарии, версии и комбинации.

Существует два способа разработки мобильных приложений: нативная и кроссплатформенная разработка. Выбор между нативной и кроссплатформенной разработкой − одно из самых важных решений в любом проекте по созданию мобильного приложения. От этого решения во многом зависит дизайн приложения, используемые для его создания технологии и, в конечном счете, пользователи, которые смогут им пользоваться.

Фундаментальное различие между нативной и кроссплатформенной разработкой заключается в том, для какой операционной системы вы разрабатываете приложение. Нативная мобильная разработка позволяет создавать приложения для конкретной операционной системы − Android или iOS.

Кроссплатформенная мобильная разработка, в свою очередь, позволяет создавать приложения для нескольких операционных систем.

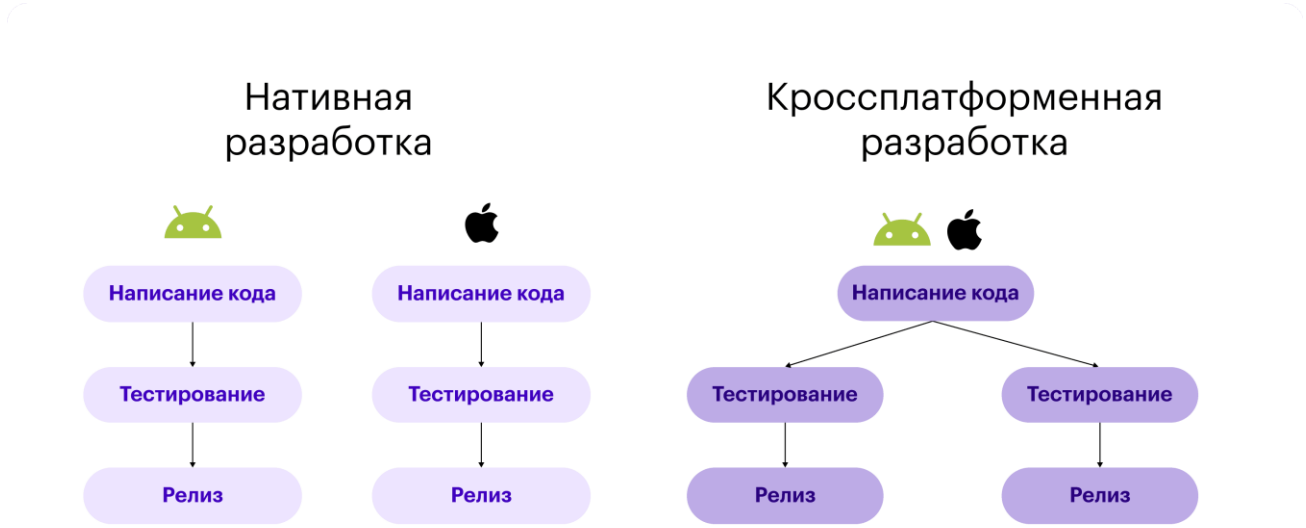


Рисунок 1.1 – Процесс разработки нативных и кроссплатформенных приложений

Рассмотрим различия между нативными и кроссплатформенными приложениями, способы их создания, а также их плюсы и минусы.

Нативные мобильные приложения – это приложения, разработанные для Android или iOS. В зависимости от того, для какой операционной системы вы разрабатываете приложение, оно обычно пишется на определенном языке программирования.

Если разрабатывается нативные приложения для iOS, то используется языки программирования Objective-C или Swift. Objective-C – это надмножество языка программирования C. Изначально он использовался для написания программного обеспечения для iOS. В 2014 году на Всемирной конференции разработчиков Apple представила Swift. Это мощный высокоуровневый язык программирования общего назначения для экосистемы Apple. По заявлениям Apple, Swift в 2,6 раза быстрее Objective-C, а его синтаксис проще для изучения.

Вот несколько известных примеров нативных мобильных приложений:

Google Карты;

Pinterest;

Spotify;

WhatsApp.

Инструменты для нативных приложений

Помимо комплектов для разработки программного обеспечения (SDK) для конкретных операционных систем, для нативной мобильной разработки требуется интегрированная среда разработки (IDE).

Для приложений на Android вам понадобится Android Studio или IntelliJ IDEA. Эти инструменты работают на Windows, macOS и Linux. Для приложений на iOS понадобится Xcode или AppCode в качестве IDE. Эти инструменты работают только на macOS.

Создание нативных мобильных приложений дает ряд преимуществ. К ним относятся:

Более высокая производительность: вы создаете и оптимизируете нативные приложения для конкретной платформы. Вы также компилируете их с использованием основного языка программирования и API платформы. Благодаря этому они работают очень быстро, эффективно и чутко реагируют на действия пользователей.

Высокий уровень безопасности: нативные приложения повышают уровень безопасности данных пользователей. Они имеют доступ к встроенным функциям безопасности, характерным для конкретной платформы.

Качественный пользовательский опыт: нативные приложения обеспечивают более удобное взаимодействие с пользователем. Внешний вид и оформление единообразны, поскольку они наследуют интерфейсы операционной системы устройства. Они соответствуют руководствам по дизайну для конкретной ОС, благодаря чему взаимодействие с приложением кажется более естественным.

Полный доступ к набору функций: Нативные мобильные приложения могут использовать все функции устройства, обеспечивая более богатый и целостный пользовательский опыт. Они также получают доступ к таким сервисам, как push-уведомления, которые играют ключевую роль в повышении вовлеченности пользователей.

Минимум ошибок: разработчики нативных приложений получают доступ к обновленным SDK сразу после их выпуска. В этих обновлениях всегда появляются улучшения и новые функции.

У нативных мобильных приложений много преимуществ, но есть и несколько недостатков. К ним относятся:

Стоимость: Разработка нативных мобильных приложений обычно требует больших затрат. Для работы с разными платформами нужны отдельные команды разработчиков. Например, для создания нативного приложения для Android нужны разработчики, владеющие Java или Kotlin. Для нативных приложений для iOS потребуются дополнительные разработчики Swift/Objective-C.

Время разработки: Работа отдельных команд над схожими приложениями для разных платформ требует времени. Поддержка и доработка также отнимают много сил.

Отсутствие возможности повторного использования кода: для работы с разными мобильными операционными системами приходится создавать и хранить код в отдельных проектах.

Цель кроссплатформенной разработки приложений − создание одного проекта для разных операционных систем. Такие приложения создаются с помощью кроссплатформенных фреймворков, которые используют SDK для конкретных платформ (Android SDK и iOS SDK) на основе унифицированного API. Это позволяет легко получать доступ к SDK и библиотекам для разных платформ.

Такие фреймворки создаются частными компаниями. Среди популярных кроссплатформенных фреймворков можно назвать следующие:

React Native от Meta. В качестве языка программирования используется JavaScript.

Flutter от Google. В качестве языка программирования используется Dart.

Xamarin от Microsoft (который переводится на MAUI). В качестве языка программирования используются C# и XAML.

Кроссплатформенные мобильные приложения обычно компилируются с использованием нативных элементов пользовательского интерфейса, благодаря чему создается впечатление, что приложение разработано для конкретной платформы. Как уже упоминалось, они представляют собой абстракцию для базовых SDK платформы. К доступным датчикам относятся GPS, уровень заряда батареи, камера и микрофон.

К известным примерам кроссплатформенных мобильных приложений относятся:

Instagram, Skype, Walmart и Airbnb (React Native);

Google Ads, приложение My BMW, eBay Motors и The New York Times (Flutter);

Всемирный банк, Fox Sports, Alaska Airlines и BBC Good Food (Xamarin).

Для развертывания приложения на Android потребуется Android SDK и компьютер с Windows, macOS или Linux. Для iOS потребуется iOS SDK, Xcode и исключительно компьютер с macOS. Ниже приведен список интегрированных сред разработки, поддерживающих кроссплатформенные фреймворки:

Мобильные приложения React Native: VS Code, Android Studio, WebStorm, Xcode и Atom;

Мобильные приложения Flutter: VS Code, Android Studio и IntelliJ;

Мобильные приложения Xamarin: Visual Studio.

Преимущества кроссплатформенных мобильных приложений

Создание кроссплатформенных мобильных приложений может принести следующие преимущества:

Низкие затраты: для кроссплатформенной мобильной разработки нужна всего одна команда разработчиков. Они должны хорошо разбираться в выбранном фреймворке. Приложение выходит на более широкую аудиторию, что позволяет ему набирать популярность и тестировать рынок.

Возможность повторного использования кода: кроссплатформенные фреймворки позволяют использовать единую кодовую базу. Это обеспечивает единообразие используемой бизнес-логики.

Быстрая разработка: благодаря повторному использованию кода и повышению производительности вы быстрее выходите на рынок. Кроссплатформенные фреймворки также оптимизируют процесс тестирования приложений в процессе разработки.

Упрощенное обслуживание: обновлять кроссплатформенное приложение проще. Это связано с тем, что вы работаете с единой кодовой базой. Во всех сборках для разных платформ используется один и тот же код, что обеспечивает единообразие.

Кроссплатформенная мобильная разработка устраняет некоторые недостатки, присущие разработке нативных приложений. Однако у нее есть и свои недостатки:

Увеличенный цифровой след: кроссплатформенные приложения обычно занимают больше места. Вам нужно обратить внимание на доступные варианты оптимизации для мобильных устройств.

Сложности с интеграцией: кроссплатформенные фреймворки не интегрируют все функции, специфичные для конкретной платформы. Для некоторых интеграций, связанных с аппаратным обеспечением, например с использованием графических процессоров, могут потребоваться навыки разработки нативных приложений.

Более низкая производительность: кроссплатформенные фреймворки обычно включают в себя собственную среду выполнения для запуска вашего приложения. Она отвечает за взаимодействие с сервисами, специфичными для конкретной платформы. Это добавляет еще один уровень вычислений, что приводит к снижению производительности.

Отложенные функции платформы: в новых выпусках SDK обычно появляются новые функции и обновления. В кроссплатформенных фреймворках для доступа к новым функциям нужно дождаться отдельного обновления.

Решение о разработке нативного или кроссплатформенного приложения имеет решающее значение. В некоторых случаях нативные приложения выигрывают у кроссплатформенных, и наоборот.

Освоение инструментов разработки − это важный, но только первый шаг. Настоящее мастерство начинается там, где мы переходим от простого написания кода к проектированию. Проектирование − это этап, на котором мы определяем структуру будущего приложения, продумываем логику взаимодействия и фиксируем все ключевые решения в документации.

Главная цель этого этапа − минимизировать риски. Согласитесь, гораздо дешевле и быстрее исправить ошибку в прототипе или в тексте технического

задания, чем переписывать тысячи строк кода готового приложения. Грамотно выстроенный процесс проектирования позволяет избежать дорогостоящих переделок и закладывает фундамент для высокого качества конечного продукта.

Этапы проектирования

1. Сбор требований

Прежде чем писать код, нужно чётко понять, что именно создается и каким оно должно быть. Этот этап фиксирует два типа требований: функциональные и нефункциональные.

Функциональные требования

Это конкретные возможности, которые приложение предоставляет пользователю.

Таблица 1.1 − Функциональные требования

Тип	Пример для приложения «Менеджер задач» (TODO-лист)
Видимые пользователю	Добавление новой задачи, отметка задачи как выполненной
Пользовательские	Регистрация, вход в аккаунт, синхронизация между устройствами
Админские	Просмотр статистики по задачам всех пользователей, управление аккаунтами

Нефункциональные требования

Это критерии качества, которые определяют, насколько хорошо система выполняет свои функции.

Таблица 1.2 − Критерии качества

Критерий	Как это понять	Метрика для примера
Производительность	Насколько быстро система реагирует	Время ответа на любое действие пользователя не превышает 2 секунд
Безопасность	Насколько надёжно защищены данные	Использование HTTPS для передачи данных и JWT-токенов для авторизации
Масштабируемость	Сможет ли система расти	Облачная инфраструктура
Доступность	Доступно ли приложение в любых условиях	Локальное хранение данных

После того как требования собраны, их необходимо визуализировать. Прототип – это интерактивная модель будущего приложения, которая позволяет проверить логику, навигацию и сценарии использования до начала разработки.

Современный инструментарий прототипирования базируется на Figma, которая остаётся стандартом индустрии. Встроенные AI-функции позволяют автоматически генерировать компоненты, адаптировать дизайн под разные платформы и создавать интерактивные переходы. В 2026 году появились решения, такие как Google Stitch, способные преобразовывать текстовое описание требований в готовые интерактивные прототипы высокой точности, сокращая время от концепции до прототипа с дней до минут.

В рамках прототипирования выделяют два аспекта дизайна.

UX-дизайн (User Experience) фокусируется на пользовательском опыте. На этом этапе создаётся карта пути пользователя (User Flow), описывающая последовательность шагов: например, авторизация → главный экран → добавление задачи → уведомление → возврат к списку. Анализируются возможные проблемные ситуации (забытый пароль, отсутствие сети) и прорабатываются сценарии их разрешения.

UI-дизайн (User Interface) отвечает за визуальное воплощение: цвета, шрифты, размеры элементов, соответствие гайдлайнам платформ (Material Design 3 для Android, Human Interface Guidelines для iOS).

Архитектура приложения − это способ организации кода, определяющий его тестируемость, масштабируемость и способность к развитию. Выбор архитектуры зависит от платформы (Android, iOS, кроссплатформа), сложности проекта и требований к долгосрочной поддержке.

В экосистеме Android официально рекомендованным подходом является использование Jetpack Compose для построения пользовательского интерфейса и архитектурных паттернов, основанных на одностороннем потоке данных (Unidirectional Data Flow, UDF). Наиболее распространённые варианты −MVVM (Model-View-ViewModel) и MVI (Model-View-Intent). Управление состоянием осуществляется через StateFlow в сочетании с `collectAsStateWithLifecycle`. Навигация реализуется через Navigation Compose. Традиционные подходы, основанные на XML-разметке и паттерне MVP, в новых проектах практически не используются.

В iOS-разработке стандартом стал SwiftUI, постепенно вытесняющий UIKit. Архитектура строится на MVVM или MVI с использованием Combine или Swift Observation для реактивного управления состоянием. Обязательным требованием является соблюдение строгой модели конкурентности Swift 6 strict concurrency.

Кроссплатформенная разработка занимает значительную долю коммерческих проектов − по оценкам, от 70 до 80% новых приложений создаются с использованием Flutter или React Native. Архитектурные подходы здесь

аналогичны нативным: MVVM и MVI адаптируются под особенности фреймворков, а Clean Architecture применяется в крупных проектах для разделения слоёв данных, домена и презентации.

Каждая архитектура имеет свои особенности. MVVM с односторонним потоком данных обеспечивает предсказуемость состояний и упрощает тестирование за счёт чёткого разделения логики и представления. MVI более строго определяет поток данных: все действия пользователя преобразуются в намерения (intents), которые обновляют единое состояние модели, что удобно для сложных экранов с большим количеством взаимосвязанных элементов. Clean Architecture добавляет слои (презентация, домен, данные) и строгие правила зависимостей, что даёт высокую независимость от фреймворков и упрощает замену технологий, но требует больших первоначальных затрат.

Пример реализации MVVM на Jetpack Compose может выглядеть так:

```
class TaskViewModel : ViewModel() {
    private val _uiState = MutableStateFlow(TaskUiState())
    val uiState: StateFlow<TaskUiState> = _uiState.asStateFlow()

    fun loadTasks() {
        viewModelScope.launch {
            _uiState.update { it.copy(isLoading = true) }
            val tasks = repository.getTasks()
            _uiState.update { it.copy(tasks = tasks, isLoading = 
false)}
        }
    }
}

data class TaskUiState(
    val tasks: List<Task> = emptyList(),
    val isLoading: Boolean = false
)
```

Во фрагменте или Compose-экране состояние наблюдается через `collectAsStateWithLifecycle()`, а UI обновляется реактивно при каждом изменении `uiState`.

Android представляет собой сложную программную платформу, построенную на основе ядра Linux. Понимание архитектуры операционной системы и принципов взаимодействия приложений с ней является необходимым условием для разработки надёжных, производительных и безопасных приложений. В этой главе рассматриваются ключевые уровни архитектуры Android, механизмы

безопасности, способы межпроцессного взаимодействия, а также принципы управления жизненным циклом приложений.

Архитектура Android организована по слоистому принципу, где каждый уровень выполняет строго определённые функции и предоставляет сервисы вышележащим уровням. Такое построение обеспечивает изоляцию компонентов, упрощает поддержку и позволяет абстрагировать разработчиков приложений от сложностей аппаратного обеспечения.

Таблица 1.3 − Уровни архитектуры Android

Уровень	**Описание**	**Ключевые компоненты (Android 15+)**
Linux Kernel	Базовое ядро для аппаратных драйверов, памяти, процессов. Версия 6.6+.	Binder драйвер, Ashmem, Low Memory Killer (LMK).
Hardware Abstraction Layer (HAL)	Абстракция аппаратного обеспечения для производителей (OEM).	Camera HAL 3.7, Neural Networks API (NNAPI 1.4).
Native Libraries	Библиотеки на C/C++ для медиа, графики.	OpenGL ES 3.2, Vulkan 1.3, WebView (Chromium 128).
Android Runtime (ART)	Виртуальная машина для выполнения Dalvik bytecode. AOT/Ahead-of-Time компиляция.	ART 15: улучшенный GC, R8/ProGuard оптимизации.
Framework	Java/Kotlin API для приложений (ActivityManager, LocationManager).	Jetpack (Room 2.6, Compose 1.7), Privacy Sandbox.
System Apps	Встроенные приложения (Settings, Phone). Могут быть заменены.	Google Mobile Services (GMS) или AOSP-версии.

Приложения в Android функционируют не изолированно, а в тесном взаимодействии с операционной системой. Это взаимодействие регулируется рядом принципов, определяющих безопасность, коммуникацию между компонентами и управление ресурсами.

Модель безопасности

Android реализует многоуровневую модель безопасности, основанную на изоляции процессов и системе разрешений.

Песочница (Sandbox). Каждое приложение выполняется в собственном процессе с уникальным идентификатором пользователя Linux (UID). Система назначает отдельный UID каждому приложению в момент установки и создаёт для него изолированную среду. Файлы, созданные одним приложением, недоступны

для чтения или записи другими приложениями по умолчанию. Эта изоляция на уровне ядра Linux составляет основу модели безопасности Android.

Разрешения (Permissions). Для доступа к защищённым данным и функциям устройства (контакты, камера, местоположение, SMS) приложение должно запросить соответствующие разрешения. Разрешения делятся на несколько категорий:

Нормальные разрешения − предоставляются автоматически во время установки или при запуске, не требуют явного подтверждения пользователя (например, доступ к интернету).

Опасные разрешения − требуют явного согласия пользователя во время работы приложения. Начиная с Android 6.0, такие разрешения запрашиваются в момент необходимости (runtime permissions).

Подписные разрешения − предоставляются только приложениям, подписанным тем же ключом, что и приложение, объявившее разрешение.

Специальные разрешения −требуют отдельной настройки пользователем в системных настройках (например, доступ к уведомлениям или установка других приложений).

Пользователь может в любой момент отозвать предоставленные разрешения через настройки системы, что требует от разработчика корректной обработки ситуаций, когда приложение лишается необходимых прав во время работы.

Подпись кода. Каждое приложение должно быть подписано цифровым сертификатом. Подпись используется для установления доверия между приложениями (например, для обмена данными через разрешения уровня подписи) и для обновлений: система позволяет обновить приложение только тем же ключом, которым оно было подписано изначально.

Межпроцессное взаимодействие (IPC)

В Android существует несколько механизмов межпроцессного взаимодействия, каждый из которых предназначен для определённых сценариев.

Intents являются основным способом коммуникации между компонентами разных приложений. Intent − это асинхронное сообщение, которое может быть явным (указывает конкретный компонент) или неявным (описывает действие, которое требуется выполнить). При отправке неявного Intent система определяет, какие приложения могут обработать запрос, и предлагает пользователю выбрать одно из них, если подходящих несколько.

Пример использования неявного Intent для открытия веб-страницы:

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("https://example.com"))
startActivity(intent)
```

Система находит установленные браузеры и либо открывает страницу в браузере по умолчанию, либо предлагает выбор.

Intent также используются для запуска служб (Services) и передачи широковещательных сообщений (Broadcast Receivers).

Binder представляет собой низкоуровневый механизм межпроцессного взаимодействия, лежащий в основе многих сервисов Android. Binder позволяет одному процессу вызывать методы другого процесса так, как если бы это был локальный вызов. Фреймворк Android предоставляет AIDL (Android Interface Definition Language) для определения интерфейсов, которые могут быть вызваны из разных процессов. Binder отличается высокой производительностью и эффективной передачей данных между процессами, поскольку использует единое пространство памяти для передачи объектов.

Content Providers являются специализированным механизмом для доступа к структурированным данным между приложениями. Content Provider инкапсулирует данные и предоставляет единый интерфейс доступа (CRUD операции: create, read, update, delete). Стандартные системные приложения (контакты, календарь, медиафайлы) предоставляют свои данные через Content Providers, что позволяет сторонним приложениям читать и изменять эти данные при наличии соответствующих разрешений.

Broadcast Receivers обеспечивают механизм распространения системных и пользовательских событий. Система отправляет широковещательные сообщения при наступлении определённых событий (завершение загрузки, подключение зарядного устройства, изменение настроек). Приложения могут регистрировать Broadcast Receivers для реагирования на такие события либо статически через манифест, либо динамически в коде.

Жизненный цикл

Android имеет уникальную модель управления процессами, ориентированную на сохранение ресурсов устройства и обеспечение отзывчивости интерфейса. В отличие от настольных операционных систем, Android не закрывает приложения при выходе пользователя, а сохраняет их в памяти для быстрого повторного запуска. Когда системе требуются ресурсы, она завершает наименее приоритетные процессы.

Жизненный цикл регулирует состояния Activity/Fragment/Service.

Система вызывает определённые методы при переходе активности из одного состояния в другое:

Таблица 1.4 − Ключевые методы

Метод	Действие
`onCreate()`	Инициализация (setContentView)
`onStart()`	Активность становится видимой для пользователя
`onResume()`	Активность получает фокус ввода и начинает взаимодействие с пользователем
`onPause()`	Частично скрыто (другой `app` сверху)
`onStop()`	Активность перестаёт быть видимой
`onDestroy()`	Активность уничтожается системой или после вызова `finish()`
`onRestart()`	Вызывается перед `onStart()`, если активность была остановлена и запускается вновь.

Важным аспектом является то, что система может завершить процесс приложения без вызова `onDestroy()`, если требуется освободить ресурсы. Поэтому разработчик должен полагаться на методы `onSaveInstanceState()` для сохранения временного состояния интерфейса, а для хранения критических данных использовать постоянное хранилище (SharedPreferences, базу данных или файлы).

Управление фоновыми задачами. Начиная с Android 8.0 (Oreo), платформа ввела строгие ограничения на выполнение фоновых работ. Службы, запущенные `startService()`, не могут выполняться в фоне более нескольких минут. Для длительных фоновых операций рекомендуется использовать WorkManager, который выбирает оптимальный способ выполнения задачи в зависимости от версии Android и состояния устройства. WorkManager поддерживает отложенные, периодические и связанные с условиями (наличие сети, зарядка) задачи.

Android-приложение строится вокруг четырёх базовых компонентов: Activity, Service, Broadcast Receiver и Content Provider. Каждый из них решает свою задачу, обладает собственным жизненным циклом и взаимодействует с операционной системой по установленным правилам. Именно эти компоненты формируют «точки входа» в приложение и определяют способы его интеграции с платформой.

Activity − экраны и их жизненный цикл.

Activity − это компонент, который обычно представляет один экран пользовательского интерфейса. В современных приложениях Activity часто выполняет роль контейнера для UI, а сама логика интерфейса строится на фрагментах, Compose или других представлениях. При этом Activity остаётся ключевой единицей жизненного цикла экрана и управления состоянием.

Практический смысл этого цикла состоит в том, что ресурсоёмкие действия должны запускаться и завершаться строго в подходящих состояниях. Например, подписки на события, запуск анимаций, работа с камерой или сохранение временного состояния требуют аккуратного управления именно через lifecycle-механизмы.

В современной архитектуре принято:

минимизировать объём логики внутри Activity, перенося бизнес-логику в отдельные слои (ViewModel, UseCase, Repository);

использовать ViewModel для сохранения состояния при пересоздании экрана (например, при повороте устройства);

избегать хранения в Activity тяжёлых объектов, привязанных к долгому времени жизни процесса.

Service предназначен для выполнения операций, которые не требуют пользовательского интерфейса. К таким операциям относятся воспроизведение музыки, синхронизация данных, загрузка файлов, длительные вычисления или выполнение периодических задач.

Платформа Android устанавливает строгие ограничения на фоновую работу. Поэтому Service используется осознанно, а для большинства длительных задач рекомендуются специализированные API:

WorkManager – для отложенных и гарантированных задач;

Foreground Service – для действий, которые действительно должны выполняться активно и о которых пользователь уведомлён;

JobScheduler – для планирования системных задач.

Основные виды Service:

Started Service – запускается методом `startService()` и работает до явной остановки (через `stopSelf()` или `stopService()`).

Bound Service – предоставляет интерфейс для связи с другими компонентами (через `bindService()`). Прекращает работу, когда все клиенты отвязываются.

Foreground Service – работает под постоянным уведомлением, имеет повышенный приоритет и не может быть остановлен системой так же легко, как обычный сервис.

Broadcast Receiver – реакция на системные события

Broadcast Receiver предназначен для реагирования на широковещательные сообщения системы или других приложений. Это механизм подписки на события: например, смена сетевого состояния, подключение питания, завершение загрузки, изменение времени или получение пользовательского события внутри приложения.

В современной версии Android использование широковещательных приёмников стало более ограниченным и адресным. Система сокращает возможность постоянного слежения за глобальными событиями ради экономии энергии и защиты приватности.

Broadcast Receiver может использоваться:

для реакции на системные изменения;

для обработки внутренних событий приложения;

для запуска логики, не требующей UI.

Важный момент: длинные операции внутри `onReceive()` выполнять нельзя, потому что метод должен завершаться быстро. Если нужна длительная обработка, приёмник должен только принять событие и передать задачу дальше – например, в WorkManager или Service.

Content Provider – общий доступ к данным

Content Provider предоставляет стандартизированный способ доступа к данным между приложениями. Он используется, когда данные должны быть доступны внешним компонентам или другим приложениям в контролируемом виде. На практике это может быть доступ к контактам, медиафайлам, календарю или собственным данным приложения, если нужно организовать безопасный обмен.

Content Provider работает через URI и набор операций:

`query()` – чтение данных;

`insert()` – добавление;

`update()` – изменение;

`delete()` – удаление;

`getType()` – определение типа данных.

Если приложение не предназначено для предоставления данных наружу, Content Provider часто вообще не требуется. Если же он используется, необходимо тщательно настраивать права доступа и экспозицию данных.

В Android приложения устроены так, что код и ресурсы разделены. Это удобно для адаптации интерфейса под разные устройства, языки и режимы экрана. Манифест, в свою очередь, играет роль декларативного описания приложения для системы.

AndroidManifest.xml: декларация компонентов, разрешений, точек входа

AndroidManifest.xml − центральный файл конфигурации Android-приложения. Он сообщает системе, что входит в состав приложения, какие компоненты доступны, какие разрешения нужны, какие экраны являются точками входа и какие особенности поддерживает программа.

В современных проектах манифест обычно содержит:

имя пакета и идентификатор приложения;

объявление `Activity`, `Service`, `Receiver`, `Provider`;

разрешения;

поддержку аппаратных и программных возможностей;

конфигурации для запуска, темы и совместимости;

сведения о `intent-filter`, если компонент должен быть доступен извне.

Папка `res/` содержит всё, что относится к ресурсам приложения, но не является исходным кодом: разметки экранов, изображения, строки, цвета, размеры, темы. Это одна из сильных сторон Android: интерфейс, тексты и конфигурации отделены от логики.

Основные типы ресурсов:

`Layout/` XML-файлы, описывающие структуру экранов (в классических приложениях) или элементы интерфейса.

`drawable/` растровые изображения, векторные ресурсы, формы, селекторы.

`mipmap/` иконки приложения (используются в лаунчере).

`values/` строки (strings.xml), цвета (colors.xml), размеры (dimens.xml), стили и темы (themes.xml), массивы.

`menu/` описания меню.

`xml/` дополнительные конфигурационные файлы.

При проектировании интерфейса рекомендуется:

использовать адаптивные размеры (dp, sp);

поддерживать разные ориентации без дублирования логики;

выносить все тексты в strings.xml;

подготавливать темы для светлого и тёмного режимов;

применять альтернативные ресурсы только там, где это действительно необходимо, чтобы не усложнять структуру проекта.

Код Android-проекта в современных условиях обычно организуется не по принципу «всё в одной папке», а по архитектурным слоям. Это необходимо для обеспечения тестируемости, масштабируемости и разделения ответственности.

Структура пакетов зависит от выбранной архитектуры, но часто включает разделение по функциональному признаку:

```
com.example.app
├── data
│   ├── api
│   ├── db
│   └── repository
├── domain
│   ├── model
│   └── usecase
├── presentation
│   ├── ui
│   ├── viewmodel
│   └── adapter
├── di
└── utils
```

Смысл такого разделения:

Data −работа с источниками данных: сетевые API, локальные базы данных, кэш, репозитории, реализующие интерфейсы доступа.

Domain − бизнес-логика: модели предметной области, сценарии использования (use cases), правила.

Presentation − интерфейс пользователя: экраны (Activity, Fragment, Compose), ViewModel, адаптеры.

Di − конфигурация внедрения зависимостей.

Utils − вспомогательные утилиты, расширения.

ОСНОВЫ ЯЗЫКА KOTLIN

Kotlin представляет современный, статически типизированный и один из самых быстроразвивающихся языков программирования, созданный и развиваемый компанией JetBrains. Kotlin можно использовать для создания самых разных приложений. Это и приложения для мобильных устройств − Android, iOS. Причем Kotlin позволяет писать кроссплатформенный код, который будет применяться на всех платформах. Это и веб-приложения, причем как серверные приложения, которые отрабатывают на стороне на стороне сервера − бекэнда, так и браузерные клиентские приложения − фронтенд. Kotlin также можно применять для создания десктопных приложений, для Data Science и так далее.

Как и Java, Kotlin −это статически типизированный язык. Проверка типов выполняется во время компиляции, и в Kotlin существует механизм вывода типов.

С его помощью Kotlin может выявлять ошибки, которые возникают, если в коде имеются противоречия. Проверка типов позволяет компилятору перехватывать и помечать большой класс ошибок программирования. В этом разделе освещаются некоторые наиболее интересные особенности системы типов

Kotlin, в том числе тип Unit, функциональные типы, null-безопасность и обобщенные типы.

В Kotlin нет примитивных типов, которые бы загромождали его систему типов. Суперкласс Any, аналог суперкласса Object в Java, лежит в основе иерархии типов Kotlin.

Все классы в Kotlin наследуют общий суперкласс, известный как Any. В Java этот суперкласс называется Object. Однако есть существенная разница: Any −это родительский класс для всех классов, не поддерживающих значения null, а Any − это родительский класс для всех классов с поддержкой этого значения. В этом и состоит основа null-безопасности. Другими словами, может быть полезно рассматривать

систему типов Kotlin как два идентичных дерева типов: все типы, не поддерживающие значения null, − это подтипы Any, а все нулевые типы с поддержкой null − это подтипы Any.

Переменные должны быть инициализированы. Для переменной нет значения по умолчанию. Например, следующий код вызовет ошибку компилятора:

```
val name: String // Ошибка! Типы, не поддерживающие значения null,
// должны быть инициализированы!
```

После определения переменной ей можно присвоить значение:

```
fun main() {
    var age: Int
    age = 23
    println(age)
}
```

Для присвоения значения переменной используется знак равно. Затем мы можем производить с переменной различные операции. Например, в данном случае с помощью функции println() значение переменной выводится на консоль. И при запуске этой программы на консоль будет выведено число 23.

Присвоение значения переменной должно производиться только после ее объявления. И также мы можем сразу присвоить переменной начальное значение при ее объявлении. Такой прием называется инициализацией:

```
fun main() {
    var age: Int = 23
    println(age)
}
```

С помощью ключевого слова val определяется неизменяемая переменная (immutable variable) или переменная только для чтения (read-only). То есть мы можем присвоить значение такой переменной только один раз, но изменить его после первого присвоения мы уже не сможем. Например, в следующем случае мы получим ошибку:

```
fun main() {
    val age: Int
    age = 23        // здесь норм – первое присвоение
    age  =  56         //  здесь  ошибка  –  переопределить  значение
переменной нельзя
    println(age)
}
```

А у переменной, которая определена с помощью ключевого слова var мы можем многократно менять значения (mutable variable):

```
fun main() {
    var age: Int
    age = 23
    println(age)
    age = 56
    println(age)
}
```

Поэтому если не планируется изменять значение переменной в программе, то лучше определять ее с ключевым словом val.

Определение констант

Также Kotlin поддерживает константы времени компиляции. Для их определения применяются ключевые слова const val:

```
const val maxAge = 120  // константа
fun main() {
    println(maxAge)
}
```

В данном случае maxAge является константой.

Отличительной особенностью констант является то, что они на стадии компиляции должны иметь некоторое значение, и это значение изменить нельзя. Это накладывает на использование констант ряд ограничений:

1. Естественно нельзя изменить значение константы:

```
const val maxAge = 120  // константа
fun main() {
    maxAge = 1500   // ошибка
    println(maxAge)
}
```

Здесь при попытке присвоения константе maxAge нового значения в функции main мы столкнемся с ошибкой на стадии компиляции.

2. Константа должна объявляться на самом верхнем уровне (вне класса/функции):

```
fun main() {
    const val maxAge = 120   // ошибка
    println(maxAge)
}
```

3. Тип данных константы должен соответствовать одному из примитивных (например, Int) или типу String.

Также стоит отметит отличие val-переменных от констант (const val): значение val-переменных устанавливается во время выполнения, а значение констант – во время компиляции. Значение val-переменной также нельзя изменить после установки, однако мы можем объявить val-переменную, а потом дальше в программе присвоить ей значение. Константе же необходимо присвоить значение сразу при определении.

В Kotlin все компоненты программы, в том числе переменные, представляют объекты, которые имеют определенный тип данных. Тип данных определяет, какой размер памяти может занимать объект данного типа и какие операции с ним можно

производить. В Kotlin есть несколько базовых типов данных: числа, символы, строки, логический тип и массивы.

Целочисленные типы

- **Byte**: хранит целое число от -128 до 127 и занимает 1 байт;
- **Short**: хранит целое число от -32 768 до 32 767 и занимает 2 байта;
- **Int**: хранит целое число от -2 147 483 648 (-2^{31}) до 2 147 483 647 ($2^{31} – 1$) и занимает 4 байта;
- **Long**: хранит целое число от –9 223 372 036 854 775 808 (-2^{63}) до 9 223 372 036 854 775 807 ($2^{63}-1$) и занимает 8 байт.

По умолчанию все числовые литералы расцениваются как значения типа Int. При присвоении числовых литералов переменным других типов данных происходит неявное преобразование. Например:

```
fun main(){

    val a: Byte = 127  // 127 – значение типа Int преобразуетсяв тип Byte
    println(a) // 127
}
```

Здесь числовой литерал, который по умолчанию представляет тип Int, преобразуется к типу Byte. И в данном случае все нормально, так как 127 укладывается в диапазон значений типа Byte.

Kotlin также поддерживает литералы типа Long – такие числовые литералы имеют суффикс L:

```
fun main(){

    val a: Long = 22    // 22 – Int, преобразование из Int в Long
    println(a) // 22
    val b: Long = 22L   // 22L – Long, преобразование не нужно
    println(b) // 22
}
```

Для передачи значений объектам, которые представляют беззнаковые целочисленные типы данных, после числа указывается суффикс U:

```
fun main(){

    val a: UByte = 10U
    val b: UShort = 45U
    val c: UInt = 250U
    val d: ULong = 30000U
    println(a) // 10
    println(b) // 45
    println(c) // 250
    println(d) // 30000
}
```

Кроме чисел в десятичной системе мы можем определять числа в двоичной и шестнадцатеричной системах.

Шестнадцатеричная запись числа начинается с 0x, затем идет набор символов от 0 до F, которые представляют число.

Кроме целочисленных типов в Kotlin есть два типа для чисел с плавающей точкой, которые позволяют хранить дробные числа:

- **Float**: хранит число с плавающей точкой от $-3.4*10^{38}$ до $3.4*10^{38}$ и занимает 4 байта
- **Double**: хранит число с плавающей точкой от $\pm5.0*10^{-324}$ до $\pm1.7*10^{308}$ и занимает 8 байта.

В качестве разделителя целой и дробной части применяется точка:

```
val height: Double = 1.78
val pi: Float = 3.14F
println(height)      // 1.78
println(pi)         // 3.14
```

Чтобы присвоить число объекту типа Float после числа указывается суффикс f или F.

Также тип Double поддерживает экспоненциальную запись:

```
val d: Double = 23e3
println(d)      // 23 000

val g: Double = 23e-3
println(g)      // 0.023
```

Тип Boolean может хранить одно из двух значений: true (истина) или false (ложь).

```
val a: Boolean = true
val b: Boolean = false
```

Символьные данные представлены типом Char. Он представляет отдельный символ, который заключается в одинарные кавычки.

```
val a: Char = 'A'
val b: Char = 'B'
val c: Char = 'T'
```

Также тип Char может представлять специальные последовательности, которые интерпретируются особым образом:

- \t: табуляция;
- \n: перевод строки;
- \r: возврат каретки;
- \': одинарная кавычка;
- \": двойная кавычка;
- \\: обратный слеш.

Строки представлены типом String. Строка представляет последовательность символов, заключенную в двойные кавычки, либо в тройные двойные кавычки.

```
fun main() {

    val name: String = "Eugene"

    println(name)
}
```

Строка может содержать специальные символы или эскейп-последовательности. Например, если необходимо вставить в текст перевод на другую строку, можно использовать эскейп-последовательность \n:

```
val text: String = "SALT II was a series of talks between United States \n and Soviet negotiators from 1972 to 1979"
```

Шаблоны строк (string templates) представляют удобный способ вставки в строку различных значений, в частности, значений переменных. Так, с помощью знака доллара $ мы можем вводить в строку значения различных переменных:

```
fun main() {

    val firstName = "Tom"
    val lastName = "Smith"
    val welcome = "Hello, $firstName $lastName"
    println(welcome)    // Hello, Tom Smith
}
```

Kotlin позволяет выводить тип переменной на основании данных, которыми переменная инициализируется. Поэтому при инициализации переменной тип можно опустить.

Kotlin предоставляет нам ряд встроенных функций для преобразования данных к определенному типу:

toByte;
toShort;
toInt;
toLong;
toFloat;
toDouble;
toChar.

Так, мы можем изменить пример выше, применив функцию toLong:

```
fun main(){

    var longN: Long = 2
    var intN: Int = 4
    longN = intN.toLong()   // Ошибки нет
    println(longN)
}
```

В Kotlin также есть тип Any, который позволяет присвоить переменной данного типа любое значение:

```
fun main() {

    var name: Any = "Tom"
    println(name)   // Tom
    name = 6758
    println(name)   // 6758
}
```

Для вывода информации на консоль в Kotlin есть две встроенные функции – `print()` и `println()`.

Обе эти функции принимают некоторый объект, который надо вывести на консоль, обычно это строка. Различие между ними состоит в том, что функция println() при выводе на консоль добавляет перевод на новую строку. Причем функция println() необязательно должна принимать некоторое значения.

Для ввода с консоли применяется встроенная функция `readLine()`. Она возвращает введенную строку. Стоит отметить, что результат этой функции всегда представляет объект типа String. Соответственно введеную строку мы можем передать в переменную типа String.

Одним из строительных блоков программы являются функции. Функция определяет некоторое действие. В Kotlin функция объявляется с помощью ключевого слова fun, после которого идет название функции. Затем после названия в скобках указывается список параметров. Если функция возвращает какое-либо значение, то после списка параметров через двоеточие можно указать тип возвращаемого значения. И далее в фигурных скобках идет тело функции.

Параметры необязательны. Возвращаемый тип можно не указывать, так как Kotlin обычно сам может его вывести на основании тела функции.

Например, определим и вызовем функцию, которая просто выводит некоторую строку на консоль:

```
fun main() {

    hello() // вызов функции hello
    hello() // вызов функции hello
    hello() // вызов функции hello
}
// определение функции hello
fun hello(){
    println("Hello")
}
```

Функции можно определять в файле вне других функций или классов, сами по себе, как например, определяется функция `main`. Такие функции еще называют функциями верхнего уровня (`top-level functions`).

Здесь кроме главной функции main также определена функция `hello`, которая не принимает никаких параметров и ничего не возвращает. Она просто выводит строку на консоль.

Функция `hello` (и любая другая определенная функция, кроме `main`) сама по себе не выполняется. Чтобы ее выполнить, ее надо вызвать. Для вызова функции указывается ее имя (в данном случае "hello"), после которого идут пустые скобки.

Таким образом, если необходимо в разных частях программы выполнить одни и те же действия, то можно эти действия вынести в функцию, и затем вызывать эту функцию.

Через параметры функция может получать некоторые значения извне. Параметры указываются после имени функции в скобках через запятую в формате `имя_параметра: тип_параметра`. Например, определим функцию, которая просто выводит сообщение на консоль:

```
fun main() {

    showMessage("Hello Kotlin")
    showMessage("Привет Kotlin")
    showMessage("Salut Kotlin")
}

fun showMessage(message: String){
    println(message)
}
```

Функция `showMessage()` принимает один параметр типа String. Поэтому при вызове функции в скобках необходимо передать значение для этого параметра: `showMessage("Hello Kotlin")`. Причем это значение должно представлять тип String, то есть строку. Значения, которые передаются параметрам функции, еще называют аргументами.

По умолчанию значения передаются параметрам по позиции: первое значение −первому параметру, второе значение −второму параметру и так далее. Однако, используя именованные аргументы, мы можем переопределить порядок их передачи параметрам

При вызове функции в скобках мы можем указать название параметра и с помощью знака равно передать ему нужное значение.

При этом, как видно из последнего случае, необязательно все аргументы передавать по имени. Часть аргументов могут передаваться параметрам по позиции. Но если какой-то аргумент передан по имени, то остальные аргументы после него также должны передаваться по имени соответствующих параметров.

Также если до обязательного параметра функции идут необязательные параметры, то для обязательного параметра значение передается по имени:

```
fun displayUser(age: Int = 18, name: String){
    println("Name: $name   Age: $age")
}
fun main() {

    displayUser(name="Tom", age=28)
    displayUser(name="Kate")
}
```

По умолчанию все параметры функции равносильны val-переменным, поэтому их значение нельзя изменить.

Однако если параметр предствляет какой-то сложный объект, то можно изменять отдельные значения в этом объекте. Например, возьмем функцию, которая в качестве параметра принимает массив:

```
fun double(numbers: IntArray){
    numbers[0] = numbers[0] * 2
    println("Значение в функции double: ${numbers[0]}")
}

fun main() {

    var nums = intArrayOf(4, 5, 6)
    double(nums)
    println("Значение в функции main: ${nums[0]}")
}
```

Здесь функция double принимает числовой массив и увеличивает значение его первого элемента в два раза. Причем изменение элемента массива внутри функции приведет к тому, что также будет изменено значение элемента в том массиве, который передается в качестве аргумента в функцию, так как этот один и тот же массив.

В Kotlin все является объектом, в том числе и функции. И функции, как и другие объекты, имеют определенный тип. Тип функции определяется следующим образом:

(типы_параметров) -> возвращаемый_тип

Возьмем функцию, которая не принимает никаких параметров и ничего не возвращает:

```
fun hello(){

    println("Hello Kotlin")
}
```

Она имеет тип () -> Unit

Если функция не принимает параметров, в определении типа указываются пустые скобки. Если не указан возвращаемый тип, то фактически, а в качестве типа возвращаемого значения применяется тип Unit.

Возьмем другую функцию:

```
fun sum(a: Int, b: Int): Int{
    return a + b
}
```

Эта функция принимает два параметра типа Int и возвращает значение типа Int, поэтому она имеет тип (Int, Int) -> Int.

Используя тип функции, мы можем определять переменные и параметры других функций, которые будут представлять функции.

Функции высокого порядка (high order function) – это функции, которые либо принимают функцию в качестве параметра, либо возвращают функцию, либо и то, и другое.

Чтобы функция могла принимать другую функцию через параметр, этот параметр должен представлять тип функции:

```
fun main() {

    displayMessage(::morning)
    displayMessage(::evening)
}
fun displayMessage(mes: () -> Unit){
    mes()
}
fun morning(){
    println("Good Morning")
}
fun evening(){
    println("Good Evening")
}
```

В данном случае функция `displayMessage()` через параметр `mes` принимает функцию типа () -> Unit, то есть такую функцию, которая не имеет параметров и ничего не возвращает.

В более редких случаях может потребоваться возвратить функцию из другой функции. В этом случае для функции в качестве возвращаемого типа устанавливается тип другой функции. А в теле функции возвращается лямбда выражение. Например:

```
fun main() {
    val action1 = selectAction(1)
    println(action1(8,5))    // 13

    val action2 = selectAction(2)
    println(action2(8,5))    // 3
}
fun selectAction(key: Int): (Int, Int) -> Int{
    // определение возвращаемого результата
    when(key){
        1 -> return ::sum
        2 -> return ::subtract
        3 -> return ::multiply
        else -> return ::empty
    }
}
fun empty (a: Int, b: Int): Int{
    return 0
}
fun sum(a: Int, b: Int): Int{
    return a + b
}
fun subtract(a: Int, b: Int): Int{
    return a - b
}
fun multiply(a: Int, b: Int): Int{
    return a * b
}
```

Здесь функция `selectAction` принимает один параметр – `key`, который представляет тип Int. В качестве возвращаемого типа у функции указан тип (Int, Int) -> Int. То есть `selectAction` будет возвращать некую функцию, которая принимает два параметра типа Int и возвращает объект типа Int.

В теле функции `selectAction` в зависимости от значения параметра `key` возвращается определенная функция, которая соответствует типу (Int, Int) -> Int.

Далее в функции main определяется переменная action1 хранит результат функции `selectAction`. Так как `selectAction()` возвращает функцию, то и переменная action1 будет хранить эту функцию. Затем через переменную action1 можно вызвать эту функцию.

Поскольку возвращаемая функция соответствует типу (Int, Int) -> Int, то при вызове в action1 необходимо передать два числа, и соответственно мы можем получить результат и вывести его на консоль.

Kotlin поддерживает объектно-ориентированную парадигму программирования, а это значит, что программу на данном языке можно представить в виде взаимодействующих между собой объектов.

Представлением объекта является класс. Класс фактически представляет определение объекта. А объект является конкретным воплощением класса. Например, у всех есть некоторое представление о машине, например, кузов, четыре колеса, руль и т.д. – некоторый общий набор характеристик, присущих каждой машине. Это представление фактически и является классом. При этом есть разные машины, у которых отличается форма кузова, какие-то другие детали, то есть конкретные воплощения этого класса, конкретные объекты или экземпляры класса.

Для определения класса применяется ключевое слово class, после которого идет имя класса. А после имени класса в фигурных скобках определяется тело класса. Если класс не имеет тела, то фигурные скобки можно опустить. Например, определим класс, который представляет человека:

```
class Person

// либо можно так
class Person { }
```

Класс фактически представляет новый тип данных, поэтому мы можем определять переменные этого типа:

```
fun main() {

    val tom: Person
    val bob: Person
    val alice: Person
}

class Person
```

В функции main определены три переменных типа Person. Стоит также отметить, что в отличие от других объектно-ориентированных языков (как C# или Java), функция main в Kotlin не помещается в отдельный класс, а всегда определяется вне какого-либо класса.

Для создания объекта класса необходимо вызвать конструктор данного класса. Конструктор фактически представляет функцию, которая называется по имени класса и которая выполняет инициализацию объекта. По умолчанию для класса компилятор генерирует пустой конструктор, который мы можем использовать.

Часть кода после знака равно Person() как раз и представляет вызов конструктора, который создает объект класса Person. До вызова конструктора переменная класса не указывает ни на какой объект.

Например, создадим три объекта класса Person:

```
fun main() {

    val tom: Person = Person()
    val bob: Person = Person()
    val alice: Person = Person()
}

class Person
```

Каждый класс может хранить некоторые данные или состояние в виде свойств. Свойства представляют переменные, определенные на уровне класса с ключевыми словами val и var. Если свойство определено с помощью val, то значение такого свойства можно установить только один раз, то есть оно immutable. Если свойство определено с помощью var, то значение этого свойства можно многократно изменять.

Свойство должно быть инициализировано, то есть обязательно должно иметь начальное значение.

Класс также может содержать функции. Функции определяют поведение объектов данного класса. Такие функции еще называют member functions или функции-члены класса. Например, определим класс с функциями:

```
class Person{
    var name: String = "Undefined"
    var age: Int = 18

    fun sayHello(){
        println("Hello, my name is $name")
    }

    fun go(location: String){
        println("$name goes to $location")
    }

    fun personToString() : String{
        return "Name: $name  Age: $age"
    }
}
```

Функции класса определяется также как и обычные функции. В частности, здесь в классе Person определена функция `sayHello()`, которая выводит на консоль строку "Hello" и эмулирует приветствие объекта Person. Вторая функция – `go()`

эмулирует движение объекта `Person` к определенному местоположению. Местоположение передается через параметр `location`. И третья функция `personToString()` возвращает информацию о текущем объекте в виде строки.

В функциях, которые определены внутри класса, доступны свойства этого класса. Так, в данном случае в функциях можно обратиться к свойствам `name` и `age`, которые определены в классе `Person`.

Для обращения к функциям класса необходимо использовать имя объекта, после которого идет название функции и в скобках значения для параметров этой функции:

```
fun main() {

    val tom = Person()
    tom.name = "Tom"
    tom.age = 37

    tom.sayHello()
    tom.go("the shop")
    println(tom.personToString())

}

class Person{
    var name: String = "Undefined"
    var age: Int = 18

    fun sayHello(){
        println("Hello, my name is $name")
    }

    fun go(location: String){
        println("$name goes to $location")
    }

    fun personToString() : String{
        return "Name: $name  Age: $age"
    }
}
```

Для создания объекта необходимо вызвать конструктор класса. По умолчанию компилятор создает конструктор, который не принимает параметров и который мы можем использовать. Но также мы можем определять свои собственные конструкторы. Классы в Kotlin могут иметь один первичный конструктор (primary constructor) и один или несколько вторичных конструкторов (secondary constructor).

Вторичные конструкторы определяются в теле класса с помощью ключевого слова constructor:

```
fun main() {

    val tom = Person("Tom", 39)
    val bob = Person("Bob", 45)

    println("Name: ${tom.name}  Age: ${tom.age}")
    println("Name: ${bob.name}  Age: ${bob.age}")
}

class Person{
    val name: String
    var age: Int

    constructor(_name: String, _age: Int){
        name = _name
        age = _age
    }
}
```

Конструкторы во многом похожи на функциии, как и обычные функции, могут иметь параметры. Здесь в классе `Person` определен конструктор, принимает два параметра: `_name` и `_age` −условно говоря имя и возраст человека. Внутри конструктора эти значения передаются переменным `name` и `age`.

Первичный конструктор является частью заголовка класса и определяется сразу после имени класса:

```
class Person constructor(_name: String, _age: Int){

}
```

Конструкторы, как и обычные функции, могут иметь параметры. Так, в данном случае конструктор имеет параметр `_name`, который представляет тип String. Через параметры конструктора мы можем передать извне данные и использовать их для инициализации объекта. При этом первичный конструктор в отличие от функций не определяет никаких действий, он только может принимать данные извне через параметры.

Если первичный конструктор не имеет никаких аннотаций или модификаторов доступа, как в данном случае, то ключевое слово constructor можно опустить.

Значения параметров первичного конструктора можно использовать внутри класса, например, передать их значения переменным класса.

Стоит отметить, что класс может иметь только один первичный конструктор. И также в классе могут быть одновременно и первичный, и вторичные конструкторы. Однако если для класса определен первичный конструктор, то вторичный конструктор должен вызывать первичный с помощью ключевого слова this:

```
class Person(_name: String){
    val name: String = _name
    var age: Int = 0

    constructor(_name: String, _age: Int) : this(_name){
        age = _age
    }
}
```

Первичный конструктор также может использоваться для определения свойств:

```
fun main() {

    val bob = Person("Bob", 23)

    println("Name: ${bob.name}  Age: ${bob.age}")
}

class Person(val name: String, var age: Int)    // { } – здесь не нужны
```

Свойства определяются как и параметры, при этом их определение начинается с ключевого слова val (если их не планируется изменять) и var (если свойства должны быть изменяемыми). И в этом случае нам уже необязательно явным образом определять эти свойства в теле класса, так как их уже определяет конструктор. И при вызове конструктора этим свойствам автоматически передаются значения: Person("Bob", 23)

Обратите внимание, что если в классе определен только первичный конструктор, то фигурные скобки, которые оформляют тело класса, использовать необязательно.

Кроме конструкторов для инициализации объектов мы можем использовать блоки инициализаторов. Они представляют блок кода в фигурных скобках, перед которым идет слово init:

```
fun main() {

    val tom = Person("Tom", -100)
    val bob = Person("Bob", 45)

    println("Name: ${tom.name}  Age: ${tom.age}")
    println("Name: ${bob.name}  Age: ${bob.age}")
}

class Person (_name: String, _age: Int){
    val name: String
    var age: Int = 1

    init{
        name = _name
        if(_age >0 && _age < 110) age = _age
    }
}
```

Здесь в классе Person определен следующий блок инициализатора:

```
init{
        name = _name
        if(_age >0 && _age < 110) age = _age
    }
```

В данном случае блок инициализатора применяется для установки переменных. Так, мы можем определить здесь проверку входных значений, как в случае с параметром _age. Можно определить какую-то другую логику, которая должна выполняться при инициализации объекта.

Стоит отметить, что в классе может быть определено одновременно несколько блоков инициализатора.

Наследование позволяет создавать классы, которые расширяют функциональность или изменяют поведение уже существующих классов. В отношении наследования выделяются два ключевых компонента. Прежде всего это базовый класс (класс-родитель, родительский класс, суперкласс), который определяет базовую функциональность. И производный класс (класс-наследник, подкласс), который наследует функциональность базового класса и может расширять или модифицировать ее.

Чтобы функциональность класса можно было унаследовать, необходимо определить для этого класса аннотацию open. По умолчанию без этой аннотации класс не может быть унаследован.

```
open class базовый_класс
class производный_класс: первичный_конструктор_базового_класса
```

Для установки наследования после названия производного класса идет двоеточие. После двоеточия в общем случае вызывается первичный конструктор класса, от которого идет наследование. Рассмотрим пример:

```
open class Person{
    var name: String = "Undefined"
    fun printName() = println(name)
}
class Employee: Person()
```

Например, в данном случае класс `Person` представляет человека, который имеет свойство `name` (имя человека) и метод `printName()` для вывода информации о человеке. Класс `Employee` представляет условного работника. Поскольку работник является человеком, то класс работника будет разделять общий функционал с классом человека. Поэтому вместо того, чтобы заново определять в классе `Employee` свойство `name`, лучше уснаследовать весь функционал класса Person. То есть в данном случае класс `Person` является базовым или суперклассом, а класс `Employee` – производным классом или классом-наследником.

Но стоит учитывать, что при наследовании производный класс должен вызывать первичный конструктор (а если такого нет, то конструктор по умолчанию) базового класса.

Здесь класс `Person` явным образом не определяет первичных конструкторов, поэтому в классе `Employee` надо вызывать конструктор по умолчанию для класса `Person.`

Вызвать конструктор базового класса в производном классе можно двумя способами. Первый способ –после двоеточия сразу указать вызов конструктора базового класса

Kotlin позволяет переопределять в производном классе функции и свойства, которые определенны в базовом классе. Чтобы функции и свойства базового класа можно было переопределить, к ним применяется аннотация `open`. При переопределении в производном классе к этим функциям применяется аннотация override.

Чтобы указать, что свойство можно переопределить в производном классе, перед его определением указывается ключевое слово `open`:

```
open class Person(val name: String){
    open var age: Int = 1
}
```

В данном случае свойство `age` доступно для переопределения.

Если свойство определяется через первичный конструктор, то также перед его определением ставится аннотация open:

```
open class Person(val name: String, open var age: Int = 1)
```

В производном классе для переопределения свойства перед ним указывается аннотация override.

Также можно переопределять геттеры и сеттеры свойств:

```
open class Person(val name: String){

    open val fullInfo: String
        get() = "Person $name – $age"

    open var age: Int = 1
        set(value){
            if(value in 1..109) field = value
        }
}
open class Employee(name: String): Person(name){

    override val fullInfo: String
        get() = "Employee $name – $age"

    override var age: Int = 18
        set(value){
            if(value in 18..109) field = value
        }
}

fun main() {

    val tom = Person("Tom")
    tom.age = 14
    println(tom.fullInfo)       // Person Tom – 14

    val bob = Employee("Bob")
    bob.age = 14
    println(bob.fullInfo)       // Employee Bob – 18
}
```

Здесь класс Employee переопределяет геттер свойства fullInfo и сеттер свойства age.

Чтобы функции базового класа можно было переопределить, к ним применяется аннотация `open`. При переопределении в производном классе к этим функциям применяется аннотация `override`:

```
open class Person(val name: String){
    open fun display() = println("Name: $name")
}
class Employee(name: String, val company: String): Person(name){

    override fun display() = println("Name: $name Company: $company")
}

fun main() {

    val tom = Person("Tom")
    tom.display()       // Name: Tom

    val bob = Employee("Bob", "JetBrains")
    bob.display()       // Name: Bob  Company: JetBrains
}
```

Функция `display` определена в классе `Person` с аннотацией `open`, поэтому в производных классах его можно переопределить. В классе `Employee` эта функция переопределена с применением аннотации `override`.

В языке Kotlin поддержка асинхронности и параллельных вычислений воплощена в виде корутин (coroutine). По сути корутина представляет блок кода, который может выполняться параллельно с остальным кодом. А базовая функциональность, связанная с корутинами, сосредоточена в библиотеке kotlinx.coroutines.

Корутины позволяют возвращать одиночные значения. Для этого мы можем, к примеру, создавать корутину с помощью построителя async. Но Kotlin также позволяет создавать асинхронные потоки (Asynchronous Flow), которые возвращают набор объектов.

В принципе для получения набора объектов мы могли бы в корутине возвращать коллекцию элементов, например, список List, наподобие следующего:

```
import kotlinx.coroutines.*

suspend fun main() = coroutineScope<Unit>{
    launch {
        getUsers().forEach { user -> println(user) }
    }
}

suspend fun getUsers(): List<String> {
    delay(1000L)  // имитация продолжительной работы
    return listOf("Tom", "Bob", "Sam")
}
```

Однако проблема таких коллекций в том, что они одномоментно возвращают все объекты. Например, если в списке ожидается 1000 объектов, то соответственно пока функция `getUsers()` не возвратит список из 1000 объектов (например, получая их из базы данных или из внешнего интернет-ресурса), мы не сможем манипулировать объектами из этого списка.

Эту проблему в Kotlin как раз и позволяют решить асинхронные потоки. Изменим пример выше с применением асинхронных потоков:

```
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.*

suspend fun main(){
    getUsers().collect { user -> println(user) }
}

fun getUsers(): Flow<String> = flow {
    val database = listOf("Tom", "Bob", "Sam")  // условная база данных
    var i = 1;
    for (item in database){
        delay(400L) // имитация продолжительной работы
        println("Emit $i item")
        emit(item) // емитируем значение
        i++
    }
}
```

Для создания асинхронного потока данных применяется интерфейс `Flow`. То есть, по сути, асинхронный поток −это объект `Flow`. Он типизируется типом тех

данных, которые должны передаваться в потоке. В данном случае передаем строки, поэтому `Flow` типизируется типом `String`.

При этом при определении функции-потока (в данном случае функции `getUsers`) необязательно использовать модификатор `suspend`.

Для создания объекта `Flow` применяется специальная функция `flow()`.

Стоит отметить, что асинхронный поток не запускается, пока не будет применена терминальная операция над получаемыми данными, например, функция `collect()`:

```
import kotlinx.coroutines.flow.*

suspend fun main(){

    val numberFlow = getNumbers()       // поток создан, но не запущен
    println("numberFlow has created")
    println("launch collect function")
    numberFlow.collect { number -> println(number) }   // запуск потока
}

fun getNumbers() = flow{
    println("numberFlow has started")
    for(item in 1..5){
        emit(item * item)
    }
}
```

БИБЛИОТЕКА JETPACK COMPOSE

Jetpack Compose представляет современный тулкит от компании Google для создания приложений под ОС Android на языке программирования Kotlin. Jetpack Compose упрощает написание и обновление визуального интерфейса приложения, предоставляя декларативный подход.

Jetpack совместим с существующим набором библиотек Android, которые можно использовать в стандартных проектах на Java и Kotlin для написания приложений под Android. Отличительной же чертой Jetpack Compose является то, что он предлагает кардинально другой подход к созданию приложений под Android.

Jetpack уменьшает объем кода, что упрощает управление кода, его поддержку и дальнейшее развитие.

Jetpack Compose предлагает декларативный API, который является более интуитивным.

Jetpack Compose ориентирован на данные (data-driven-подход). В частности, Compose применяет концепцию состояния (state). Данные сохраняются как состояние, что гарантирует, что при любые изменения в данных автоматически отразяться на изменении пользовательского интерфейса. Соответственно не потребуется определять какой-то дополнительный код, который бы отслеживал наличие изменений. Любой компонент интерфейса, который обращается к состоянию, подписывается на все изменения этого состояния. И когда состояние изменяется, подписанный компонент пересоздается, чтобы отразить изменения в состоянии. Данный процесс еще называется рекомпозиция.

При этом Jetpack Compose совместим с уже имеющимся кодом. Например, можно вызывать код Jetpack Compose из традиционного приложения на Android. Большинство стандартных библиотек под Android также работают с Jetpack Compose.

Ключевой концепцией тулкита Jetpack Compose является composable-функция (функция, которая имеет аннотацию @Composable). Такие функции представляют некоторые части визуального интерфейса, из которых строится приложение. Это упрощает построение и обновление сложных интерфейсов, тестирование и поддержку самих компонентов.

При создании проекта, который использует Jetpack Compose, в Android Studio, в проект по умолчанию добавляется файл MainActivity.kt, который содержит одноименный класс MainActivity. Этот класс определяет интерфейс, который мы видим на устройстве/эмуляторе при запуске проекта. Так, по умолчанию графический интерфейс будет выглядеть будет следующим образом:

Рисунок 3.1 – Графический интерфейс Jetpack Compose

Рассмотрим основные моменты кода, который добавляется в файл MainActivity.kt и который определяет подобный интерфейс:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.helloapp.ui.theme.HelloappTheme
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            HelloappTheme {
                Scaffold(modifier    =    Modifier.fillMaxSize())    {
innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }

    @Composable
    fun Greeting(name: String, modifier: Modifier = Modifier) {
        Text(
            text = "Hello $name!",
            modifier = modifier
        )
    }

    @Preview(showBackground = true)
    @Composable
    fun GreetingPreview() {
        HelloappTheme {
            Greeting("Android")
        }
    }
```

Рассмотрим основные моменты этого кода вкратце. Вначале идет определение пакета, к которому принадлежит класс **MainActivity**. В нашем случае это пакет `com.example.helloapp`.

Далее идут подключаемые пакеты, функционал который использует класс MainActivity.

Обратите внимание на последний подключаемый пакет – он представляет функционал из каталога **ui.theme**, который также добавляется в проект по умолчанию и который содержит ряд файлов на языке Kotlin: **Color.kt, Theme.kt и Type.kt.**

Далее идет определение класса MainActivity.

Приложение на Android определяется как набор объектов, которые называются **activity**. Объект **activity** фактически представляет отдельное окно графического приложения. И класс **MainActivity** как раз представляет окно, которое отображается при запуске приложения

MainActivity наследуется от встроенного класса **ComponentActivity. ComponentActivity** обеспечивает построение интерфейса из визуальных компонентов и для этого предоставляет минимальный функционал. В частности, ComponentActivity предоставляет метод `onCreate()`, который вызывается при запуске MainActivity и создает ее интерфейс.

В метод передается объект **Bundle**, который хранит состояние MainActivity – некоторые значения, которые хранят связанные с MainActivity данные. А в самом методе `onCreate()` вначале вызывается реализация этого метода из базового класса ComponentActivity.

Затем идет вызов метода `enableEdgeToEdge()`, который позволяет сделать верхние и нижние (правые и левые) панели устройства прозрачными.

Для собственно создания интерфейса в `onCreate()` вызывается другой метод базового класса ComponentActivity − метод `setContent()`. Этот метод, собственно, и определяет, какой интерфейс мы увидим на экране устройства. Этот метод выполняет функцию, в которой устанавливается компонент **@Composable**.

Функция **@Composable** представляет центральную концепцию фреймворка Jetpack Compose и, грубо говоря, представляет некоторый визуальный компонент. Причем этот компонент должен быть оформлен в виде функции с аннотацией **@Composable**. Так, в проекте по умолчанию пользовательский интерфейс определяется функцией `HelloAppTheme`, которая определена в проекте в файле **Theme.kt**.

Эта функция кроме того, что задает визуальный интерфейс, также обеспечивает соответствие приложения текущей теме (светлой или темной) устройства. Для этого она в качестве первого параметра принимает в качестве первого параметра булевое значение, которое указывает, выбрана ли темная тема. В зависимости от значения этого параметра устанавливает соответствующие цвета, которые определены в файле **Color.kt**. Последний параметр представляет еще один компонент **@Composable** −то есть опять же некоторый визуальный компонент. Далее этот компонент передает в в **MaterialTheme**.

MaterialTheme задает визуальный интерфейс в стиле Material Design. Для этого он использует также настройки шрифта в виде компонента Typography из файла Type.kt. Можно сказать, что фактически за компонентом `HelloAppTheme` прячется объект MaterialTheme, которой устанавливает дизайн в стиле Material Design.

Scaffold фактически представляет промежуточный компонент, который задает дополнительное оформление в стиле Material Design и применяется для построения структуры страницы. В этот компонент передается один параметр − `modifier` – модификатор, который устанавливает различные аспекты внешнего вида и поведения компонента. По умолчанию этот параметр принимает результат функции `Modifier.fillMaxSize()`, которая указывает растянуть компонент по всему доступному пространству (в итоге компонент будет занимать все пространство экрана устройства).

Последний параметр функции-компонента Scaffold – параметр `content` устанавливает содержимое компонента. Причем этот параметр представляет функцию типа @Composable ((PaddingValues) -> Unit).

В коде приложения по умолчанию параметр content реализован в виде концевой лямбды, в которую передается значение `innerPadding`. А после стрелки указывается компонент, который и представляет содержимое Scaffold:

```
{ innerPadding ->
    Greeting(
        name = "Android",
        modifier = Modifier.padding(innerPadding)
    )
}
```

Это компонент Greeting, который определен в том же файле.

Этот компонент использует другой встроенный компонент –`Text`, который представляет некоторый текст. Именно этот текст в итоге мы увидим на экране устройства.

И в данном случае мы видим, что для вывода простой надписи "Hello Android" используется целый набор компонентов **@Composable**, которые вложены в друг друга по принципу матрешки: Text -> Greeting -> Scaffold -> MaterialTheme -> HelloAppTheme.

Однако выше не была упомянута еще одна часть кода MainActivity.kt – функция `GreetingPreview`.

Весь интерфейс в компоненте `GreetingPreview` можно увидеть в Android Studio при просмотре в режиме **Design** или **Split**:

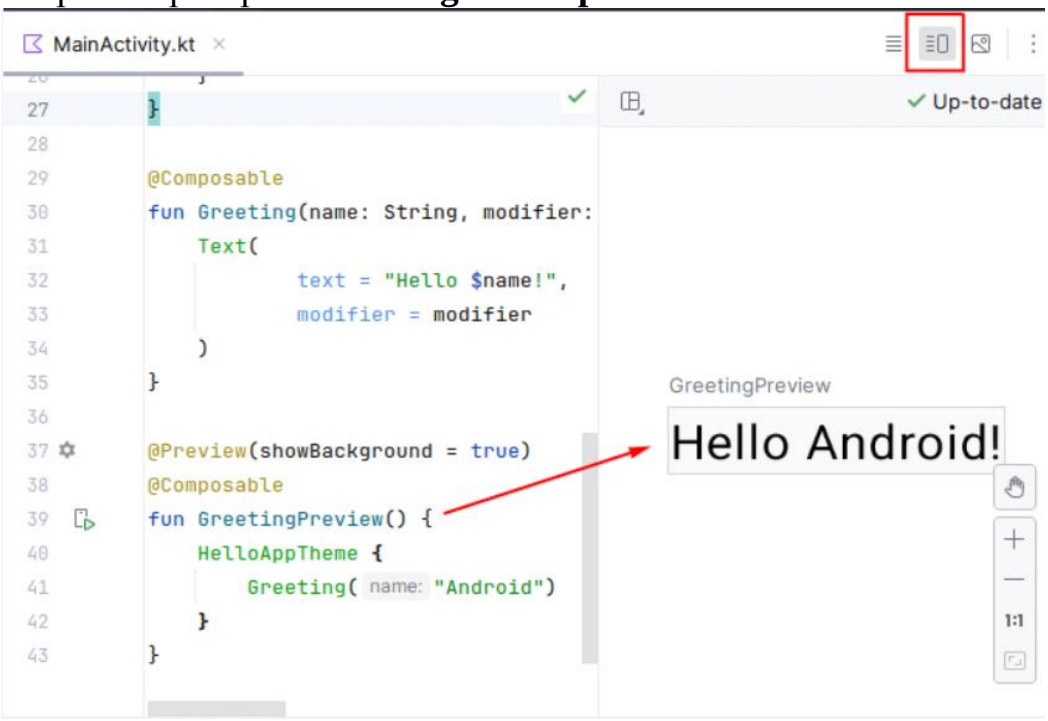


Рисунок 3.2– Интерфейс в компоненте `GreetingPreview`

С помощью параметров аннотации **@Preview** можно настраивать предварительный просмотр. Например, если надо отобразить остальную часть экрана, то можно установить опцию `showSystemUi = true`:

```
@Preview(showSystemUi = true)
```

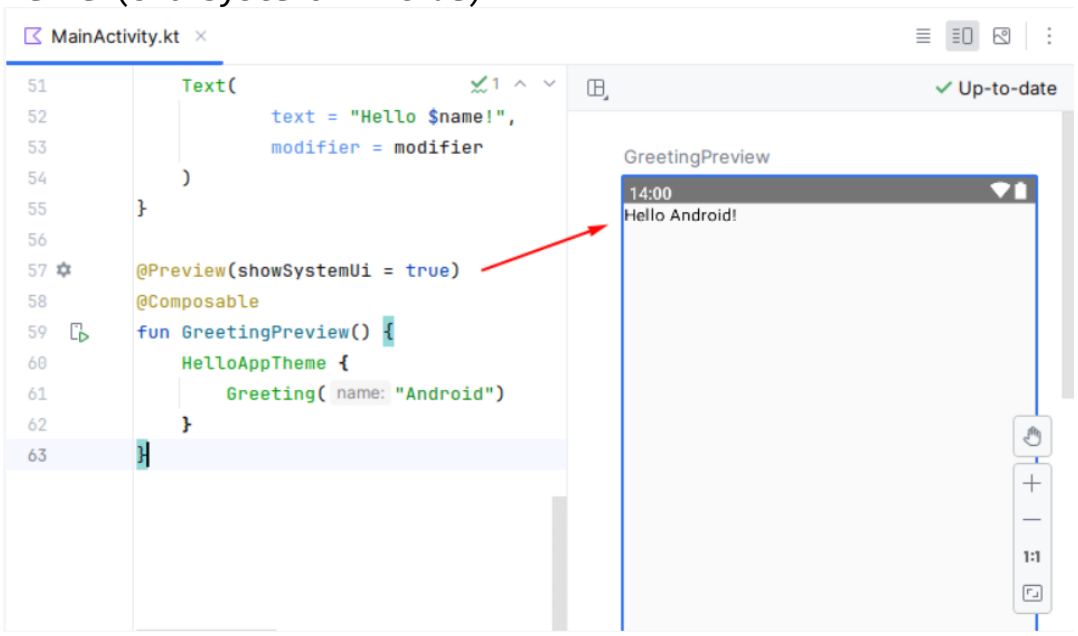


Рисунок 3.3 – Предварительный просмотр

Стоит отметить, что окно предварительного просмотра автоматически подхватывает все изменения в интерфейсе приложения

Изменение интерфейса

По умолчанию корневым компонентом является `HelloAppTheme`, который определен в проекте в файле `ui.theme/Theme.kt`. Однако нам в принципе необязательно использовать всю эту комплексную композицию компонентов. Так, в примере выше, если убрать настройку темы приложения, то, по сути, все, что оно делает – это выводит некоторый текст на экран устройства. И в реальности мы могли бы в этом случае ограничиться одним компонентом **Text**. Так, изменим код **MainActivity.kt** следующим образом:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material3.Text
import androidx.compose.ui.unit.sp

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(text = "Hello METANIT.COM!", fontSize = 28.sp)
        }
    }
}
```

Компонент, который представляет, по сути, то, что мы увидим на экране устройства, передается в метод **setContent()** класса MainActivity. И в данном случае в этот метод сразу передается компонент **Text.**

У этого компонента устанавливаются в данном случае два свойства. Свойство **text** представляет выводимый текст. Кроме того, здесь устанавливается свойство **fontSize**, которое задает размер шрифта. В качестве значения оно принимает числовое значение в единицах **sp.**

МОДИФИКАТОРЫ И ВИЗУАЛЬНЫЙ ИНТЕРФЕЙС

В Jetpack Compose для настройки внешнего вида и стилизации большинства стандартных компонентов используются модификаторы. Это функции, каждая из которых отвечает за конкретный аспект оформления − например, задаёт размеры, фон или отступы.

Почти все встроенные компоненты предоставляют параметр modifier, через который можно применить такие настройки. К примеру, компонент **Text** (как и остальные) реализован в виде **composable**-функции с аннотацией **@Compose** и содержит ряд параметров, включая **modifier.**

```
@Composable
fun Text(
    value: String,
    modifier: Modifier = Modifier,

    //..... остальные параметры

): @Composable Unit
```

В определении composable-функции видно, что у неё есть параметр modifier с типом (точнее, интерфейсом) **Modifie**r. Его назначение − хранить конфигурацию, которая затем применяется к компоненту. Сам интерфейс **Modifier** включает множество встроенных функций-модификаторов, позволяющих изменять различные аспекты компонентов.

Далее мы рассмотрим эти функции по группам. Большинство из них достаточно универсальны и подходят для многих компонентов. А пока на примере компонента **Text** посмотрим, как именно применяются модификаторы.

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.background
import androidx.compose.material3.Text
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.sp
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(
                "Hello METANIT.COM",
                fontSize = 28.sp,
                modifier = Modifier.background(Color.LightGray)
            )
        }
    }
}
```

Здесь для компонента Text задается три параметра. Первый параметр представляет отображаемый текст. Второй параметр – `fontSize` задает размер шрифта – 28 единиц. Третий параметр задает модификаторы, которые применяются к компоненту `modifier = Modifier.background(Color.LightGray).`

В данном случае применяется только один модификатор – функция `background()`, которая устанавливает фоновый цвет. В нее передается определение цвета – здесь желтый цвет, предоставленный встроенным значением **Color.LightGray**.

Для установки фонового цвета применяется модификатор background(). Есть две версии этой функции. Первая версия:

```
fun Modifier.background(color: Color, shape: Shape = RectangleShape): Modifier
```

color: значение цвета в виде объекта Color;

shape: закрашиваемая форма – объект Shape. По умолчанию это прямоугольная форма.

Вторя версия:

```
fun Modifier.background(
    brush: Brush,
    shape: Shape = RectangleShape,
    alpha: Float = 1.0f
): Modifier
```

brush: применяемая для покрытия цветом кисть в виде объекта Brush;

shape: закрашиваемая форма – объект Shape. По умолчанию это прямоугольная форма;

alpha: значение прозрачности. Оно должно находится в диапазоне от 0 (полностью прозрачно) до 1 (отсутствие прозрачности).

В данном случае разберем только установку цвета.

В Jetpack Compose цвет представлен классом Color из пакета androidx.compose.ui.graphics..

Есть несколько способов создания цвета. Рассмотрим некоторые:

Color(value): в конструктор передается шестнадцатеричное значение цвета. Например:

```
val myColor: Color = Color(0xFF0000FF)
```

Color(red, green, blue, alpha): в конструктор передается числовые значения для каждой отдельной компоненты цвета.

```
val myColor: Color = Color(red = 0xFF, green = 0xFF, blue = 0xFF, alpha = 0xFF)
```

Также есть ряд встроенных значений:

```
Color.Black;
Color.Blue;
Color.Cyan;
Color.DarkGray;
Color.Gray;
Color.Green;
Color.LightGray;
Color.Magenta;
Color.Red;
Color.Transparent;
Color.White;
Color.Yellow.
```

Для установки размеров компонентов в Jetpack Compose определен целый ряд модификаторов:

`Modifier.height()`: устанавливает высоту;

`Modifier.width()`: устанавливает ширину;

`Modifier.fillMaxHeight()`: растягивает компонент по всей длине контейнера;

`Modifier.heightIn()`: устанавливает минимальную и максимальную высоту;

`Modifier.widthIn()`: устанавливает минимальную и максимальную ширину

`Modifier.size()`: устанавливает размер;

`Modifier.sizeIn()`: устанавливает минимальный и максимальный размер.

Для установки применяются единицы `dp` (`device-independent pixels/density-independent pixels` или независимые от устройства/плотности пиксели).

Например, установим для компонента Text высоту и ширину:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.width
import androidx.compose.material3.Text
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(
                "Hello METANIT.COM",
                fontSize=28.sp,
                modifier                                                    =
Modifier.background(color=Color.LightGray).width(300.dp).height(200.dp)
            )
        }
    }
}
```

Для установки отступов внутри компонента применяется модификатор `padding()`, которые имеет несколько вариантов:

```
// устанавливает отступы от каждой стороны по отдельности
fun Modifier.padding(start: Dp = 0.dp, top: Dp = 0.dp, end: Dp = 0.dp,
bottom: Dp = 0.dp): Modifier
// устанавливает отступы по вертикали и по горизонтали
fun Modifier.padding(horizontal: Dp = 0.dp, vertical: Dp = 0.dp):
Modifier
// устанавливает одно значение для отступов от всех четырех сторон
fun Modifier.padding(all: Dp): Modifier
// устанавливает отступы в виде объекта PaddingValues
fun Modifier.padding(paddingValues: PaddingValues): Modifier
```

В качестве значения отступов применяются единицы **dp.**

Тип **PaddingValue**s применяет значения для установки отступов:

```
public fun PaddingValues(
    start: Dp,
    top: Dp,
    end: Dp,
    bottom: Dp
): PaddingValues
```

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Text
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent{
            val innerPadding = PaddingValues(top = 20.dp, start =
15.dp)
            Text(
                "Hello METANIT.COM",
                fontSize=28.sp,
                modifier = Modifier.padding(innerPadding)
            )
        }
    }
}
```

Для сдвига содержимого компонента по горизонтали и вертикали применяется модификатор **offset**. Он имеет следующие версии:

```
fun Modifier.offset(x: Dp = 0.dp, y: Dp = 0.dp): Modifier
fun Modifier.offset(offset: Density.() -> IntOffset): Modifier
```

Модификатор **border** позволяет определить границу вокруг компонента. Этот модификатор имеет следующие определения:

```
Modifier.border(border: BorderStroke, shape: Shape = RectangleShape)
Modifier.border(width: Dp, brush: Brush, shape: Shape = RectangleShape)
Modifier.border(width: Dp, color: Color, shape: Shape = RectangleShape)
```

Первый вариант в качестве первого параметра принимает объект `BorderStroke`, в который через конструктор передается ширина линии в единицах **dp** и ее цвет в виде объекта **Brush**: `BorderStroke(width: Dp, brush: Brush)`.

Последний параметр представляет объект **Shape**, который определяет форму контура вокруг компонента. По умолчанию это `RectangleShape`, который представляет прямоугольную область.

Второй и третий варианты, по сути, повторяют первый вариант, только в третьем случае для установки цвета применяется объект **Color**.

Модификатор **clip** применяется для определения фрагмента компонента. Он имеет следующее определение:

```
fun Modifier.clip(shape: Shape): Modifier
```

В качестве параметра он принимает объект интерфейса **Shape**, который определяет вырезаемую форму.

Распространенным сценарием использования этого модификатора является создание фрагмента компонента на основе непрямоугольной формы, например, в виде овала или круга. Например, образуем из компонента **Text** округлую форму:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.padding
```

```
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material3.Text
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.sp

import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
  class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text("Hello METANIT.COM!",
                fontSize = 28.sp,
                modifier = Modifier
                    .padding(10.dp)
                    .clip(shape= RoundedCornerShape(30.dp))
                    .background(Color.LightGray)
                    .padding(15.dp)
            )
        }
    }
}
```

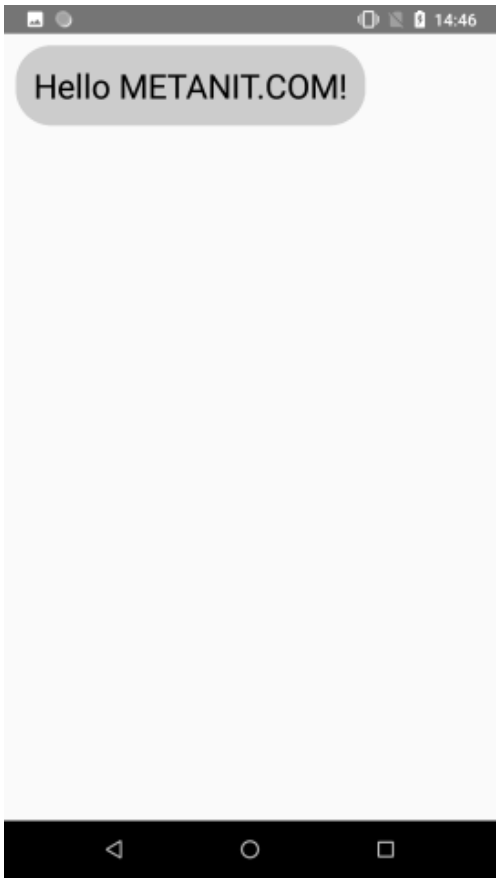


Рисунок 4.1 – Образование округлой формы

Модификатор **clip** упрощает создание более сложных по композиции визуализаций компонентов. Например:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Text
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.sp

import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.graphics.RectangleShape
import androidx.compose.ui.unit.dp


class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text("Hello METANIT.COM!",
                fontSize = 28.sp,
                modifier = Modifier
                    .padding(10.dp)     // отступ от границ контейнера
                    .background(Color.DarkGray)
                    .padding(25.dp)    // отступ от границ фрагмента
                    .clip(shape= RectangleShape)
                    .border(width = 2.dp, color = Color.White)
                    .background(Color.LightGray)
                    .padding(20.dp)     // отступ между границей во
фрагменте и текстом
                )
        }
    }
```

Модификатор **shadow** позволяет создать тень вокруг элемента. Этот модификатор имеет следующую форму:

```
fun Modifier.shadow(
    elevation: Dp,
    shape: Shape = RectangleShape,
    clip: Boolean = elevation > 0.dp,
    ambientColor: Color = DefaultShadowColor,
    spotColor: Color = DefaultShadowColor
): Modifier
```

Функция модификатора принимает следующие параметры:

`elevation:` высота тени в **dp;**

`shape`: определяет форму тени с помощью объекта **Shape**;

`clip`: если равно t**rue**, то создается фрагмент с использованием формы из параметра **shap**e;

`ambientColor`: цвет затенения на компоненте;

`spotColor`: цвет тени вокруг компонента.

Для создания тени достаточно задать ее высоту.

Параметр shape позволяет задать форму тени. Он принимает объект интерфейса **Shape**. Например, определим округлую тень и для этого используем встроенный класс **CircleShape/**

Объект **Modifier** позволяет добавить компонентам возможность обработки нажатия. Для этого он предоставляет функцию `clickable():`

```
fun Modifier.clickable(
    enabled: Boolean = true,
    onClickLabel: String? = null,
    role: Role? = null,
    onClick: () -> Unit
): Modifier
```

Параметры функции:

`enabled`: значение типа Boolean, которое указывает, будет ли доступен компонент для нажатия. По умолчанию имеет значение true, то есть компонент будет доступен для нажатия. При значении false обработка нажатий оключена.

`onClickLabel`: предоставляет значение типа **String** и задает метку, с помощью которой можно быстро обратиться к компоненту. По умолчанию равен null;

role: объект типа **Role**, который устанавливает тип визуального элемента. По умолчанию равен null;

`onClick`: функция типа () -> Unit, которая собственно и обрабатывает нажатие.

КОНТЕЙНЕРЫ КОМПОНОВКИ

В Jetpack Compose за расположение и компоновку компонентов в приложении отвечает специальный компонент − **Layou**t. Он имеет следующую сигнатуру:

```
@Composable inline fun Layout(
    content: @Composable () -> Unit,
    modifier: Modifier = Modifier,
    measurePolicy: MeasurePolicy
)
```

Он имеет следующие параметры:

`content`: Composable-функция, которая хранит все вложенные компоненты;

`modifier:` объект Modifier, которые устанавливает функции-модификаторы, применяемые для стилизации компонента;

`measurePolicy`: объект, который отвечает за вычисление размеров компонентов и их определение их расположения.

Создание своего алгоритма вычисления размеров компонентов внутри компонента-контейнера, должная компоновка вложенных компонентов представляют трудоемкую работу, поэтому фреймворк Compose предоставляет ряд встроенных компонентов-контейнеров, которые автоматически выполняют эту работу. Основными из них являются **Box, Row и Column**.

Компонент **Box** является наиболее простым контейнером, позволяя позиционировать вложенное содержимое. Он представляет некоторую область экрана. Функция данного компонента принимает четыре параметра:

```
@Composable
inline fun Box(
    modifier: Modifier = Modifier,
    contentAlignment: Alignment = Alignment.TopStart,
    propagateMinConstraints: Boolean = false,
    content: @Composable BoxScope.() -> Unit
): @Composable Unit
```

modifier: объект **Modifie**r, который позволяет настроить внешний вид и поведение компонента с помощью модификаторов;

contentAlignment: объект **Alignmen**t, который устанавливает расположение компонента. По умолчанию имеет значение `Alignment.TopStart` (расположение вначале контейнера в верхнем углу);

propagateMinConstraints: значение типа **Boolean**, который указывает, надо ли применять к содержимому ограничения по минимальным размерам. По умолчанию равно false (ограничения не применяются);

content: объект интерфейса **BoxScope,** который представляет вложенное содержимое.

Контейнер **Box** может использоваться как самодостаточный компонент без какого-либо вложенного содержимого.

Для установки визуальных свойств объекта **Box** применяется свойство **modifier**, которое представляет объект **Modifier**, рассмотренный в прошлой главе. Как и для других компонентов, для компонента **Box** мы можем вызвать функцию **size()**, которая принимает ширину и длину контейнера − в данном случае ширина 300 единиц и высота 250 единиц.

```
Modifier.size(300.dp, 250.dp)
```

Эта функция опять же возвращает объект **Modifier**, у которого далее вызывается другая функция −**background**(), которая устанавливает фоновый цвет контейнера − в данном случае синий цвет или значение `Color.Blue`.

```
Modifier.size(300.dp, 250.dp).background(Color.Blue)
```

В итоге выведется на экран синяя прямоугольная область размером 300 x 250.

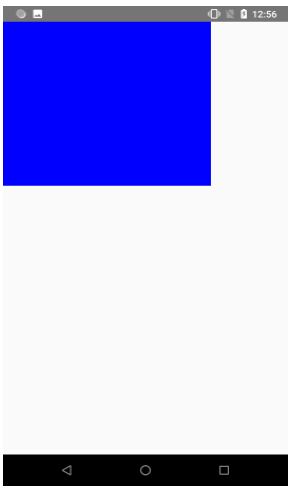


Рисунок 5.1 − Результат вывода background(Color.Blue)

По умолчанию Box занимает те размеры на экране, которые необходимы, чтобы вместить содержимое. Но выше в Box никакого вложенного содержимого не было, поэтому чтобы установить размеры, применялся объект `Modifier` и его функция `size()`.

Если необходимо растянуть элемент по всей ширине и длине экрана, то, как и в общем случае, применяется функция − **fillMaxSize()**:

```
Modifier.fillMaxSize().background(Color.Blue)
```

Для позиционирования внутри Box данный компонент определяет параметр **contentAlignment**, которому передаются свойства объекта **Alignment:**

`Alignment.BottomCenter`: внизу по центру;

`Alignment.BottomEnd`: внизу в конце;

`Alignment.BottomStart`: внизу в начале;

`Alignment.Center`: по центру по вертикали и горизонтали;

`Alignment.CenterEnd`: по центру про вертикали и в конце по горизонтали;

`Alignment.CenterStart`: по центру про вертикали и в начале по горизонтали;

`Alignment.TopCenter`: вверху по центру;

`Alignment.TopEnd`: вверху в конце;

`Alignment.TopStart`: вверху в начале.

Значение `Alignment.Center` указывает, что содержимое будет позиционироваться по центру как по горизонтали, так и по вертикали. А чтобы Box был растянут по всей площади экрана, применяется модификатор `Modifier.fillMaxSize()`:

Рисунок 5.2 – Использование свойства позиционирования `Alignment.Center`

Контейнер **Column** позволяет выстроить вложенные компоненты в столбик. Функция **Column** принимает четыре параметра:

```
@Composable
inline fun Column(
    modifier: Modifier = Modifier,
    verticalArrangement: Arrangement.Vertical = Arrangement.Top,
    horizontalAlignment: Alignment.Horizontal = Alignment.Start,
    content: @Composable ColumnScope.() -> Unit
): @Composable Unit
```

- **modifier**: объект **Modifier**, который позволяет настроить внешний вид и поведение компонента;
- **verticalArrangement**: объект **Arrangement.Vertical**, который устанавливает выравнивание компонента по вертикали. По умолчанию имеет значение Arrangement.Top (расположение в верху);
- **horizontalAlignment**: объект **Alignment.Horizontal**, который устанавливает выравнивание компонента по горизонтали. По умолчанию имеет значение `Alignment.Start` (расположение в начале – слева для языков с левосторонним письмом и справа для языков с правосторонним письмом);
- **content**: объект интерфейса **BoxScope**, который представляет вложенное содержимое.

Если ни у одного вложенного компонента НЕ указан вес, **Column** занимает то пространство, которое необходимо, чтобы вместить вложенные компоненты. Если подобный размер контейера нежелателен, то можно задать конкретный размер с помощью модификатора **Modifier.height**:

```
Column(modifier = Modifier.height(550.dp))
```

С помощью модификатора **Modifier.fillMaxHeight** можно растянуть контейер по всей длине экрана.

```
Column(modifier = Modifier.fillMaxSize())
```

Если как минимум для одного вложенного компонента указан вес, то в модификаторе **Modifier.fillMaxHeight** нет смысла. Однако в этот случае по-прежнему можно использовать модификаторы **Modifier.height** и **Modifier.size** для ограничения длины контейера.

Если высота контейера C**olumn** больше суммы высот его вложенных компонентов, то для позиционирования этих компонентов может применяться параметр **verticalArrangemen**t, который может принимать следующие значения:

- **Arrangement.Center**: расположение по центру;
- **Arrangement.Bottom**: расположение внизу;
- **Arrangement.Top**: расположение вверху;
- **Arrangement.SpaceAround**: компоненты равномерно распределяются по всей высоте с равномерными отступами между элементами, при этом отступы между первым и последним элементами и границами контейнера равен половине отступов между элементами;
- **Arrangement.SpaceBetween**: компоненты равномерно распределяются по всей высоте с равномерными отступами между элементами, при этом первый и последний элементы прижимаются к границам контейнера;
- **Arrangement.SpaceEvenly**: компоненты равномерно распределяются по всей высоте с равномерными отступами между элементами, при этом отступы между первым и последним элементами и границами контейнера равны отступам между элементами.

Например:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
```

```
 class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column(modifier         =         Modifier.fillMaxSize(),
verticalArrangement = Arrangement.SpaceBetween )
            {
                Box(modifier                                                =
Modifier.background(Color.Red).fillMaxWidth().height(100.dp))
                Box(modifier                                                =
Modifier.background(Color.DarkGray).fillMaxWidth().height(100.dp))
                Box(modifier                                                =
Modifier.background(Color.Blue).fillMaxWidth().height(100.dp))
            }
        }
    }
}
```

Рисунок 5.3 – Выравнивание вложенных компонентов

Параметр `horizontalAlignment` позволяет выровнять содержимое вложенных компонентов. Этому параметру можно передать одно из следующих значений:

`Alignment.Start`: выравнивание в начале (по умолчанию);

`Alignment.End`: выравнивание в конце;

`Alignment.CenterHorizontally`: выравнивание по центру;

Начало и конец в зависимости от направления написания для текущего языка может означать левый или правый край. Например:

```
package com.example.helloapp
```

```
import android.os.Bundle
import 74ndroid.activity.ComponentActivity
import 74ndroid.activity.compose.setContent
import 74ndroid.compose.foundation.layout.Column
import 74ndroid.compose.foundation.layout.fillMaxSize
import 74ndroid.compose.material3.Text
import 74ndroid.compose.ui.Alignment
import androidx.compose.ui.Modifier
import 74ndroid.compose.ui.unit.sp


class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column(horizontalAlignment = Alignment.End, modifier =
Modifier.fillMaxSize())
            {
                Text(“Kotlin”, fontSize = 28.sp)
                Text(“Java”, fontSize = 28.sp)
                Text(“JavaScript”, fontSize = 28.sp)
                Text(“Python”, fontSize = 28.sp)
            }
        }
    }
}
```

Здесь идет выравнивание по правому краю:

Рисунок 5.4 − Выравнивание по правому краю

Поскольку вложенные компоненты в Row определяются внутри ColumnScope, то **ColumnScope** предоставляет вложенным компонентам еще ряд модификаторов:

- **align()**: определяет выравнивание компонента по вертикали и может принимать значения:
  - `Alignement.Start:` выравнивание по началу контейнера (в зависимости от ориентации это правая или левая сторона);
  - `Alignement.CenterHorizontally`: выравнивание по центру;
  - `Alignement.End`: выравнивание по концу контейнера (в зависимости от ориентации это правая или левая сторона);
- **alignBy()**: выравнивает компонент относительно определенной вертикальной линии;
- **weight()**: задает вес компонента.

Вес (weight) внутри **Column** позволяет назначить вложенным компонентам высоту в соответствии с некоторым коэффициентом. Для указания веса применяется модификатор **ColumnScope.weight**. Стоит учитывать, что если контейнер **Column** обеспечивает вертикальную прокрутку или располагается в контейере, который предполагает вертикальную прокрутку, то веса компонентов игнорируются, поскольку общее пространство по вертикали условно бесконечно.

Если у компонента не указан вес, то контейер сначала запрашивает его предпочтительную высоту. Затем оставшее простанство распределяется между компонентами, для которых указан вес, в соответствии с их весом.

```
Package com.example.helloapp

import android.os.Bundle
import 76ndroid.activity.ComponentActivity
import 76ndroid.activity.compose.setContent
import 76ndroid.compose.foundation.background
import 76ndroid.compose.foundation.layout.Box
import 76ndroid.compose.foundation.layout.Column
import 76ndroid.compose.foundation.layout.fillMaxWidth
import 76ndroid.compose.foundation.layout.height
import 76ndroid.compose.ui.Modifier
import 76ndroid.compose.ui.graphics.Color
import 76ndroid.compose.ui.unit.dp


class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column {
                Box(modifier =
Modifier.background(Color.Red).fillMaxWidth().weight(1f))
                Box(modifier =
Modifier.background(Color.DarkGray).fillMaxWidth().height(150.dp))
                Box(modifier =
Modifier.background(Color.Blue).fillMaxWidth().weight(2f))
            }
        }
    }
}
```

Здесь для второго элемента Box установлена высота в 150 единиц, а для остальных установлены веса – 1f и 2f. Таким образом, получится, что от всей длины контейера Column (а он будет растягиваться на весь экран) второй элемент получит высоту в 150 единиц. Оставшееся пространство затем будет распределено между первым и третим элементами в соответствии с их весами.

Контейнер **Row** располагает вложенные компоненты в строку. Функция **Row** принимает четыре параметра:

```
@Composable
inline fun Row(
    modifier: Modifier = Modifier,
    horizontalArrangement: Arrangement.Horizontal = Arrangement.Start,
    verticalAlignment: Alignment.Vertical = Alignment.Top,
    content: @Composable RowScope.() -> Unit
): @Composable Unit
```

- **modifier**: объект **Modifier**, который позволяет настроить внешний вид и поведение компонента;
- **horizontalArrangement**: объект **Arrangement.Horizontal**, который устанавливает выравнивание компонента по горизонтали. По умолчанию имеет

значение `Arrangement.Start` (расположение в вначале: слева для левосторонних языков и справа для правосторонних языков);

- **verticalAlignment**: объект **Alignment.Vertical**, который устанавливает выравнивание компонента по вертикали. По умолчанию имеет значение `Alignment.Top` (расположение вверху);
- **content**: объект интерфейса **RowScope**, который представляет вложенное содержимое.

Разместим в **Row** несколько элементов Box:

```
package com.example.helloapp

import android.os.Bundle
import 77ndroid.activity.ComponentActivity
import 77ndroid.activity.compose.setContent
import 77ndroid.compose.foundation.background
import 77ndroid.compose.foundation.layout.Box
import 77ndroid.compose.foundation.layout.Row
import 77ndroid.compose.foundation.layout.height
import 77ndroid.compose.foundation.layout.width
import 77ndroid.compose.ui.Modifier
import 77ndroid.compose.ui.graphics.Color
import 77ndroid.compose.ui.unit.dp


class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Row {
                Box(modifier =
Modifier.background(Color.Red).height(150.dp).width(100.dp))
                Box(modifier =
Modifier.background(Color.DarkGray).height(150.dp).width(100.dp))
                Box(modifier =
Modifier.background(Color.Blue).height(150.dp).width(100.dp))
            }
        }
    }
}
```

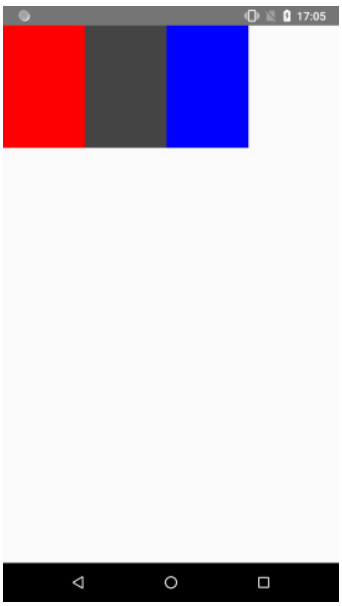


Рисунок 5.5 – Размещение элементов в **Row**

Если ни у одного вложенного компонента НЕ указан вес, **Row** занимает то пространство, которое необходимо, чтобы вместить вложенные компоненты. Если подобный размер контейера не желателен, то можно задать конкретный размер с помощью модификатора **Modifier.width**:

```
Row(modifier = Modifier.width(350.dp))
```

С помощью модификатора **Modifier.fillMaxWidth** можно растянуть контейер по всей длине экрана.

```
Row(modifier = Modifier.fillMaxWidth())
```

Если как минимум для одного вложенного компонента указан вес, то в модификаторе **Modifier.fillMaxWidth** нет смысла. Однако в этот случае по прежнему можно использовать модификаторы **Modifier.width** и **Modifier.size** для ограничения ширины контейера.

Если ширина контейера **Row** больше суммы ширин его вложенных компонентов, то для позиционирования этих компонентов по горизонтали может применяться параметр **horizontalArrangement**, который может принимать следующие значения:

- **Arrangement.Center**: расположение по центру;
- **Arrangement.End**: расположение в конце (справа – для левосторонних языков, слева – для правосторонних языков);
- **Arrangement.Start**: расположение в начале (слева – для левосторонних языков, справа – для правосторонних языков);
- **Arrangement.SpaceAround**: компоненты равномерно распределяются по всей ширине с равномерными отступами между элементами, при этом отступы между первым и последним элементами и границами контейнера равен половине отступов между элементами;
- **Arrangement.SpaceBetween**: компоненты равномерно распределяются по всей ширине с равномерными отступами между элементами, при этом первый и последний элементы прижимаются к границам контейнера;
- **Arrangement.SpaceEvenly**: компоненты равномерно распределяются по всей ширине с равномерными отступами между элементами, при этом отступы

между первым и последним элементами и границами контейнера равны отступам между элементами.

Параметр **verticalAlignment** в Row позволяет установить выравнивание по вертикали для вложенных компонентов и принимает объект интерфейса `Alignment.Vertical`. В частности, мы можем использовать одно из следующих значений:

- Top: выравнивание по верху;
- CenterVertically: выравнивание по центру;
- Bottom: выравнивание по нижнему краю контейнера.

Например, применим выравнивание по центру строки:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.Text
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.sp


class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Row(modifier = Modifier.fillMaxSize(),
                horizontalArrangement = Arrangement.SpaceEvenly,
                verticalAlignment = Alignment.CenterVertically)
            {
                Text("Kotlin", fontSize = 28.sp)
                Text("JavaScript", fontSize = 28.sp)
                Text("Python", fontSize = 28.sp)
            }
        }
    }
}
```

Рисунок 5.6 − Выравнивание по центру строки

Поскольку вложенные компоненты в Row определяются внутри RowScope, то **RowScope** предоставляет вложенным компонентам еще ряд модификаторов:

- **align()**: определяет выравнивание компонента по вертикали и может принимать значения:
  - `Alignement.Top`: выравнивание по верху контейнера;
  - `Alignement.CenterVertically`: выравнивание по центру;
  - `Alignement.Bottom`: выравнивание по нижней стороне контейнера;
- **alignBy()**: выравнивает компонент относительно определенной горизонтальной линии;
- **alignByBaseline()**: выравнивает базовую линию компонента относительно базовой линии другого сестринского компонента, применяется для выравнивания текста компонентов по одной линии;
- **paddingFrom()**: добавляет отступ при выравнивании;
- **weight()**: задает вес компонента.

СОСТОЯНИЕ КОМПОНЕНТОВ

Jetpack Compose применяет декларативный подход для создания интерфейса, поэтому единственный способ обновить визуальный компонент представляет повторный вызов функции этого компонента, в которую передаются новые значения. И чтобы упростить обновление компонентов и вообще визуального интерфейса Jetpack Compose предоставляет концепцию **состояние** или **state.**

Состояние представляет некоторое значение, которое хранится в приложении и которое в процессе его работы может изменяться.

Когда значение состояния изменяется, происходит рекомпозиция (**recomposition**) или обновление пользовательского интерфейса. Перекомпоновка всего пользовательского интерфейса каждый раз при изменении значения состояния была бы крайне неэффективным подходом к рендерингу и обновлению пользовательского интерфейса. Compose позволяет избежать этих накладных расходов, используя метод, называемый интеллектуальной рекомпозицией, который включает в себя перекомпоновку только тех компонентов, на которые непосредственно влияет изменение состояния. Как только Compose обнаруживает изменение состояния, он перекомпоновывает любые компоненты, на которые повлияло изменение состояния. Перекомпоновка просто означает, что функция компонента вызывается снова, и ей передается новое значение состояния.

Измененяемое состояние в Jetpack Compose представлено объектом интерфейса **MutableState<T>**, который имеет следующее определение:

```
interface MutableState<T> : State<T> {
    override var value: T
}
```

Объект этого интерфейса хранит значение, которое будет изменяться, в виде переменной **value**. Причем данное значение может представлять любой тип.

Объект MutableState<T> интегрирован со средой выполнения Compose и позволяет отслеживать изменения хранимого в нем значения. Любые изменения этого значения приведут к обоновлению (или рекомпозиции) любого компонента, который использует данное значение.

Для создания объекта MutableState<T> Jetpack Compose предоставляет функцию **mutableStateOf()**:

```
val mutableState = mutableStateOf(значение)
```

В функцию `mutableStateOf()` в качестве параметра передается отслеживаемое значение. Возвращаемый из функции объект MutableState позволяет связать отслеживаемое значение с компонентом.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val message = mutableStateOf("Hello METANIT.COM")
            Text(text = message.value, fontSize = 25.sp)
        }
    }
}
```

Здесь в функцию `mutableStateOf()` передается строка, то есть объект **String**, поэтому возвращаемый функцией объект будет представлять тип `MutableState<String>`.

С помощью свойства **value** можно получить отслеживаемое значение: **message.value**. В данном случае это значение передается в компонент Text для отображения.

Однако просто определение состояния − это еще не все. В вполне возможно, что Android Studio даже не сможет скомпилировать вышеприведенный код. Потому что нам нужен еще ожин элемент − функция **remember**. Эта функция применяется для сохранения некоторого объекта в памяти. Она может хранить как изменяемые (mutable), так и неизменяемые (immutable) объекты. Причем данные объекты сохраняются во время начального построения интерфейса (то что называется **initial composition**) и продолжают храняться во время обновлений интерфейса (рекомпозиции).

Есть несколько способов применения функции `remember` в сочетании с функцией `mutableStateOf`. Самый распространенный:

```
val mutableState = remember { mutableStateOf(значение) }
```

Обращаясь к свойству **value** также можно изменить отслеживаемое значение:

```
val mutableState = mutableStateOf("Hello METANIT.COM")
mutableState.value = "Hello World"
```

Теперь посмотрим, как динамически изменять это состояние. Компонент Text позволяет применить модификатор clickable, который обрабатывает нажатия на компонент. Применим этот модификатор для динамического изменения текста в компоненте Text по нажатию на него:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val message = remember{mutableStateOf("Hello
METANIT.COM")}
            Text(
                text = message.value,
                fontSize = 25.sp,
                modifier = Modifier.clickable( onClick = {
message.value = "Hello Work!" })
                )
        }
    }
}
```

Здесь в модификаторе `clickable` параметру **onClick** передается обработчик нажатия, который изменяет отслеживаемое значение. То есть мы ожидаем, что по нажатию на текст он изменится с "Hello METANIT.COM" на "Hello Work!".

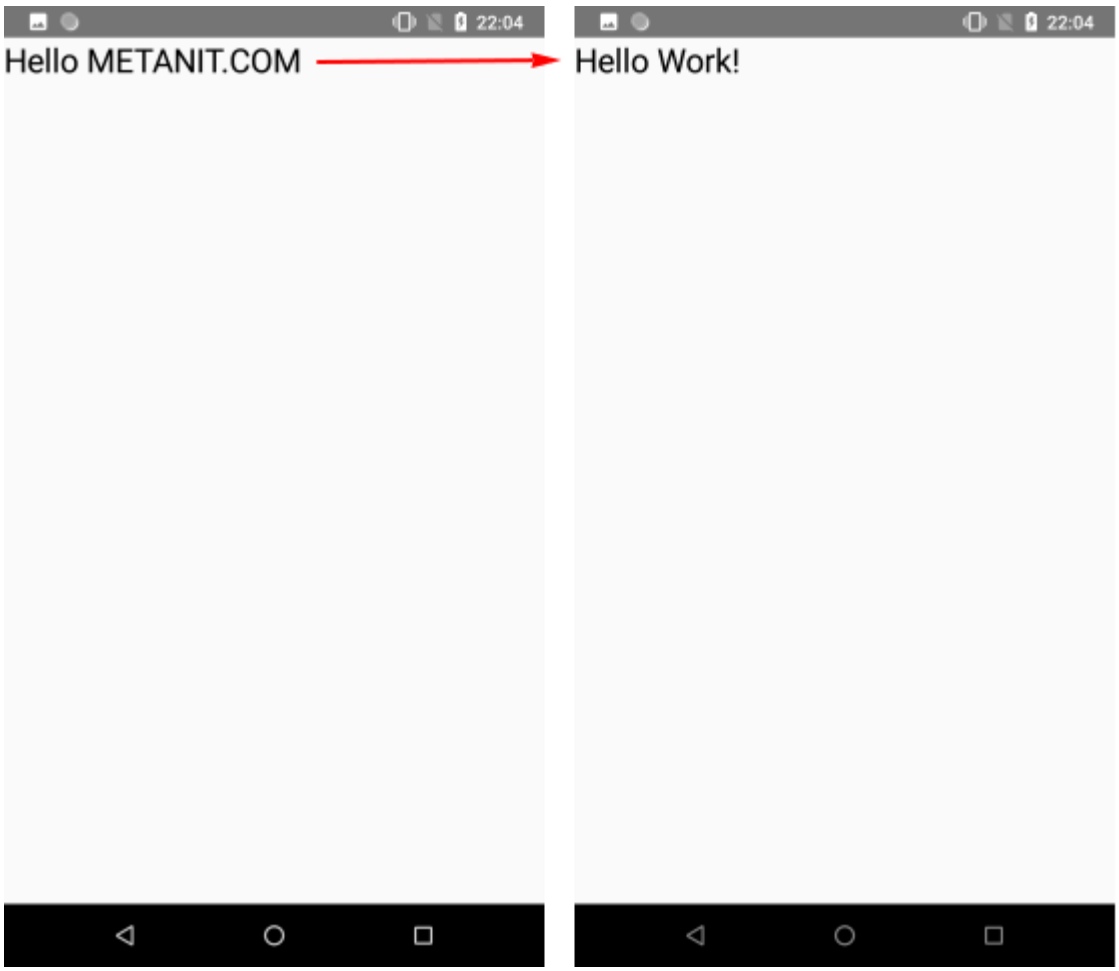


Рисунок 6.1 − Передача обработчика нажатия

Стоит отметить, что есть еще два способа использования функции **remember:**

```
var value by remember { mutableStateOf(значение) }
```

В этом случае применятся делегат **by**, для работы с которым необходимо импортировать следующие функции:

import androidx.compose.runtime.getValue

import androidx.compose.runtime.setValue

Даже если мы изменим состояние, при изменении ориентации мобильного устройства (например, при переходе от портретного режима к альбомному) наше состояние пропадет:

То же самое касается и других конфигурационных изменений, например, изменения системных настроек. Подобные изменения вызывают не просто рекомпозицию отдельных компонентов, а уничтожение и пересоздание всего класса MainActivity. В итоге вновь созданный объект MainActivity не понмнит никаких изменений состояния.

Для сохранения состояния между поворотами экрана и другими подобными системными изменениями конфигурации применяется обертка над функцией `remember` − функцию **rememberSaveable**, которая сохраняет данные в объект **Bundle**. Правда, не все объекты можно сохранить, а только данные

примитивных типов, либо классов, которые реализуют интерфейс Parcelable. Например, изменим предыдущий пример применив функцию rememberSaveable:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            var value by rememberSaveable{mutableStateOf("Hello METANIT.COM")}

            Text(
                text = value,
                fontSize = 28.sp,
                modifier = Modifier.clickable( onClick = { value = "Hello Work!"})
            )
        }
    }
}
```

ВИЗУАЛЬНЫЕ КОМПОНЕНТЫ

Неотъемлемой частью визуального интерфейса является текст. Для отображения текста Jetpack Compose предоставляет ряд встроенных компонентов. Прежде всего это компонент **Text**, который имеет следующее определение:

```
@Composable
fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified,
    fontStyle: FontStyle? = null,
    fontWeight: FontWeight? = null,
    fontFamily: FontFamily? = null,
    letterSpacing: TextUnit = TextUnit.Unspecified,
    textDecoration: TextDecoration? = null,
    textAlign: TextAlign? = null,
    lineHeight: TextUnit = TextUnit.Unspecified,
    overflow: TextOverflow = TextOverflow.Clip,
    softWrap: Boolean = true,
    maxLines: Int = Int.MAX_VALUE,
    minLines: Int = 1,
    onTextLayout: ((TextLayoutResult) -> Unit)? = null,
    style: TextStyle = LocalTextStyle.current
): Unit
```

Параметры компонента:

- `text`: объект String, который представляет выводимый текст;
- `modifier`: объект **Modifier**, который представляет применяемые к компоненту модификаторы;

• color: объект **Color**, который представляет цвет текста. По умолчанию имеет значение Color.Unspecified;

• fontSize: объект **TextUnit**, который представляет размер шрифта. По умолчанию равен TextUnit.Unspecified;

• fontStyle: объект **FontStyle**, который представляет стиль шрифта. По умолчанию равен null;

• fontWeight: объект **FontWeight**, который представляет толщину шрифта. По умолчанию равен null;

• fontFamily: объект **FontFamily**, который представляет тип шрифта. По умолчанию равен null;

• letterSpacing: объект **TextUnit**, который представляет отступы между символами. По умолчанию равен TextUnit.Unspecified;

• textDecoration: объект **TextDecoration**, который представляет тип декораций (например, подчеркивание), применяемых к тексту. По умолчанию равен null;

• textAlign: объект **TextAlign**, который представляет выравнивание текста. По умолчанию равен null;

• lineHeight: объект **TextUnit**, который представляет высоту строки текста. По умолчанию равен TextUnit.Unspecified;

• overflow: объект **TextOverflow**, который определяет поведение текста при его выходе за границы контейнера. По умолчанию равен TextOverflow.Clip;

• softWrap: объект **Boolean**, который определяет, должен ли текст переносится при завершении строки. При значении false текст не переносится, как будто строка имеет бесконечную длину. По умолчанию равен true;

• maxLines: объект **Int**, который представляет максимальное количество строк. Если текст превысил установленное количество строк, то он усекается в соответствии с параметрами overflow и softWrap. По умолчанию равен Int.MAX_VALUE;

• minLines: минимальное количество строк;

• onTextLayout: объект **(TextLayoutResult) -> Unit**, который представляет функцию, выполняемую при определении компоновки текста;

• style: объект **TextStyle**, который представляет стиль текста. Значение по умолчанию – LocalTextStyle.current.

Размер шрифта определяется параметром fontSize. В качестве параметру может передаваться значение типов Int, Double и Float, после которых указывается тип единиц. Это могут быть масштабируемые пиксели (единицы **sp**, например, 22.sp), либо это может быть относительный размер шрифта в единицах **em** (например, 18.em). Значение TextUnit.Unspecified указывает, что высота шрифта наследуется от настроек родительного компонента.

За определение цвета шрифта отвечает параметр color, который представляет объект **Color**.

Стиль шрифта определяется параметром **fontStyle**, который представляет класс **FontStyle**. Для определения этот класс предоставляет два встроенных значения:

- `FontStyle.Italic` (наклоннный шрифт);
- `FontStyle.Normal` (стандартный шрифт).

```
import androidx.compose.ui.text.font.FontStyle
//.........
Text("Hello Jetpack Compose!", fontSize=22.sp, fontStyle = FontStyle.Italic)
Text("Hello Jetpack Compose!", fontSize=22.sp, fontStyle = FontStyle.Normal)
```

Толщина шрифта задается параметром **fontWeight**, который представляет класс **FontWeight**.

Есть два способа установки толщины шрифта. Прежде всего можно использовать конструктор этого класса, в который передается числовое значение от 1 до 1000. Чем больше значение, тем толще будет шрифт.

Второй способ заключается в применении встроенных значений:

- `FontWeight.Black` (Эквивалентно значению W900);
- `FontWeight.Bold` (Эквивалентно значению W700);
- `FontWeight.ExtraBold` (Эквивалентно значению W800);
- `FontWeight.ExtraLight` (Эквивалентно значению W200);
- `FontWeight.Light` (Эквивалентно значению W300);
- `FontWeight.Medium` (Эквивалентно значению W500);
- `FontWeight.Normal` (Эквивалентно W400 − значение по умолчанию);
- `FontWeight.`SemiBold (Эквивалентно значению W600);
- `FontWeight.Thin` (Эквивалентно значению W100.

Тип или семейство шрифта определяется параметром **fontFamily**, который представляет объект **FontFamily**

Для определения шрифта `FontFamily` предоставляет ряд встроенных констант:

- `FontFamily.Cursive` (курсивный, рукописный шрифт);
- `FontFamily.Monospace`;
- `FontFamily.Serif`;
- `FontFamily.SansSerif`;
- `FontFamily.Default` (шрифт по умолчанию на текущей платформе);
- `FontFamily.SansSerif`.

Параметр `letterSpacing` задает расстояние между символами и представляет класс `TextUnit`. В данном случае мы можем установить расстояние, так как и размер шрифта, с помощью единиц **sp** или **em**.

```
Text("Hello Jetpack Compose!",  fontSize=22.sp, letterSpacing= 1.3.sp)
Text("Hello Jetpack Compose!",  fontSize=22.sp, letterSpacing= 0.3.em)
```

Параметр **textDecoration** позволять задать декорации для текста. Данный параметр принимает объект класса **TextDecoration**, который предоставляет несколько встроенных значений:

- `TextDecoration.LineThrough` (зачеркивает текст);
- `TextDecoration.Underline` (подчеркивает текст);
- `TextDecoration.None` (отсутствие декораций).

```
import androidx.compose.ui.text.style.TextDecoration
//...........
Column {
    Text("Hello Jetpack Compose!",  fontSize=28.sp, textDecoration =
TextDecoration.LineThrough)
    Text("Hello Jetpack Compose!",  fontSize=28.sp, textDecoration =
TextDecoration.Underline)
    Text("Hello Jetpack Compose!",  fontSize=28.sp, textDecoration =
TextDecoration.None)
}
```

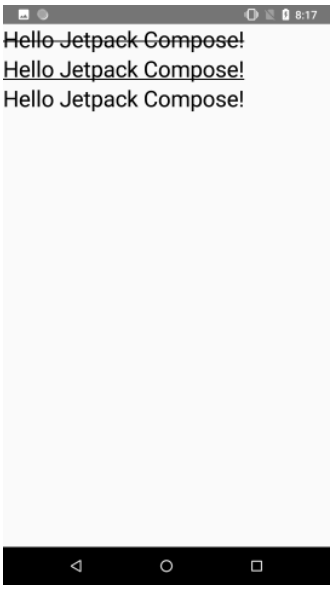


Рисунок 7.1 − Декорация текста

Параметр **textAlign** управляет выравниванием текста и представляет объект класса **TextAlign**. В качестве значения этому параметру можно передать значение одного из свойств класса TextAlign:

- `TextAlign.Center`: выравнивание текста по центру контейнера;
- `TextAlign.Justify`: текст равномерно растягивается по всей ширине контейнера;
- `TextAlign.End`: выравнивание текста по конечному краю контейнера (в зависимости от ориентации текста это может быть левый или правый край);
- `TextAlign.Start`: выравнивание текста по началу контейнера (в зависимости от ориентации текста это может быть левый или правый край);
- `TextAlign.Left`: выравнивание текста по левому краю контейнера;

• TextAlign.Right: выравнивание текста по правому краю контейнера.

Для создания кнопок в Jetpack Compose применяется компонент **Button**, который имеет следующее определение:

```
@Composable
fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    shape: Shape = ButtonDefaults.shape,
    colors: ButtonColors = ButtonDefaults.buttonColors(),
    elevation: ButtonElevation? = ButtonDefaults.buttonElevation(),
    border: BorderStroke? = null,
    contentPadding: PaddingValues = ButtonDefaults.ContentPadding,
    interactionSource: MutableInteractionSource? = null,
    content: @Composable RowScope.() -> Unit
): Unit
```

Параметры функции компонента:

• onClick: представляет функцию-обработчик нажатия кнопки;

• modifier: представляет объект **Modifier**, который определяет модификаторы кнопки;

• enabled: значение типа Boolean устанавливает, доступна ли кнопка для нажатия. По умолчанию равно true (то есть кнопка доступна для нажатия);

• interactionSource: представляет объект типа **MutableInteractionSource**, который устанавливает поток взаимодействий для кнопки. Значение по умолчанию – remember { MutableInteractionSource() };

• elevation: объект типа **ButtonElevation**, который определяет анимацию для кнопки. По умолчанию равно ButtonDefaults.elevation();

• shape: объект типа **Shape**, который устанавливает форму кнопки. По умолчанию равно MaterialTheme.shapes.small;

• border: объект типа **BorderStroke**, который устанавливает границу кнопки. По умолчанию равно null;

• colors: объект типа **ButtonColors**, который устанавливает цвета кнопки. По умолчанию равно ButtonDefaults.buttonColors();

• contentPadding: объект типа **PaddingValues**, который устанавливает отступы между границами кнопки и ее содержимым. По умолчанию равно ButtonDefaults.ContentPadding;

• content: содержимое кнопки в виде строки Row.

Кнопка представляет сложный компонент, который может содержать другие компоненты. Таким образом мы можем создавать комплексные кнопки с различным содержимым.

Для обработки нажатия параметру **onClick** передается функция, которая будет выполняться при нажатии.

Параметр elevation определяет анимацию для кнопки в разных состояниях и представляет объект интерфейса `ButtonElevation`. По умолчанию этот параметр в качестве значения имеет компонент `ButtonDefaults.elevation`:

```
@Composable
fun elevation(
    defaultElevation: Dp = 2.dp,
    pressedElevation: Dp = 8.dp,
    disabledElevation: Dp = 0.dp
): @Composable ButtonElevation
```

Этот компонент определяет ряд параметров, между значениями которых будет идти анимация при переключении состояния кнопки:

- `defaultElevation`: определяет анимацию, когда кнопка доступна для нажатия и когда для нее не определено других объектов Interaction, которые определяют состояние для кнопки.
- `pressedElevation`: определяет анимацию для кнопки в нажатом состоянии.
- `defaultElevation`: определяет анимацию для кнопки в отключенном состоянии.

Цвета кнопки задаются с помощью параметра colors, который предоставляет объект **ButtonColors** и по умолчанию равен компоненту `ButtonColors.buttonColors`:

```
@Composable
public final fun buttonColors(
    containerColor: Color = FilledButtonTokens.ContainerColor.toColor(),
    contentColor: Color = FilledButtonTokens.LabelTextColor.toColor(),
    disabledContainerColor: Color = FilledButtonTokens.DisabledContainerColor.toColor()
        .copy(alpha = FilledButtonTokens.DisabledContainerOpacity),
    disabledContentColor: Color = FilledButtonTokens.DisabledLabelTextColor.toColor()
        .copy(alpha = FilledButtonTokens.DisabledLabelTextOpacity)
): ButtonColors
```

- `containerColor`: определяет фоновый цвет, когда кнопка доступна для нажатия.
- `contentColor`: определяет цвет содержимого, когда кнопка доступна для нажатия.
- `disabledBackgroundColor`: определяет фоновый цвет, когда кнопка не доступна для нажатия.
- `disabledContentColor`: определяет цвет содержимого, когда кнопка не доступна для нажатия.

Для установки цветов кнопки мы можем создать свой класс или компонент интерфейса **ButtonColors**, либо воспользоваться встроенным компонентом `ButtonDefaults.buttonColors`.

За установки границы кнопки (а именно ее толщины и цвета) отвечает параметр border, который представляет класс **BorderStroke** со следующими конструкторами:

```
BorderStroke(width: Dp, color: Color)
BorderStroke(width: Dp, brush: Brush)
```

Первый параметр конструкторов устанавливает толщину границы, а второй – ее цвет с помощью объекта Brush или Color.

Для ввода текста в приложении предназначен компонент **TextField**. Он имеет несколько версий, возьмем одну из них:

```
@Composable
fun TextField(
    value: String,
    onValueChange: (String) -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    readOnly: Boolean = false,
    textStyle: TextStyle = LocalTextStyle.current,
    label: (@Composable () -> Unit)? = null,
    placeholder: (@Composable () -> Unit)? = null,
    leadingIcon: (@Composable () -> Unit)? = null,
    trailingIcon: (@Composable () -> Unit)? = null,
    prefix: (@Composable () -> Unit)? = null,
    suffix: (@Composable () -> Unit)? = null,
    supportingText: (@Composable () -> Unit)? = null,
    isError: Boolean = false,
    visualTransformation:            VisualTransformation            =
VisualTransformation.None,
    keyboardOptions: KeyboardOptions = KeyboardOptions.Default,
    keyboardActions: KeyboardActions = KeyboardActions.Default,
    singleLine: Boolean = false,
    maxLines: Int = if (singleLine) 1 else Int.MAX_VALUE,
    minLines: Int = 1,
    interactionSource: MutableInteractionSource? = null,
    shape: Shape = TextFieldDefaults.shape,
    colors: TextFieldColors = TextFieldDefaults.colors()
): Unit
```

Основные параметры компонента:

- `value`: представляет введенное в текстовое поле значение в виде строки, то есть объекта **String;**

• onValueChange: функция обработки изменения введенного значения. Представляет функцию типа **(String) -> Unit;**

• modifier: объект типа **Modifier**, который задает модификаторы компонента;

• enabled: устанавливает, будет ли поле доступно для ввода. Представляет значение типа **Boolean**. По умолчанию равно true, то есть поле доступно для ввода;

• readOnly: устанавливает, будет ли поле доступно только для чтения. Представляет значение типа **Boolean**. По умолчанию равно false, то есть поле доступно не только для чтения, но для изменения значения;

• textStyle: объект типа **TextStyle**, который устанавливает стиль текста. Значение по умолчанию – LocalTextStyle.current;

• label: устанавливает дополнительную метку, которая отображается внутри поля. Для установки метки применяется функция типа **() -> Unit**. Значение по умолчанию – null;

• placeholder: плейсхолдер – временный текст, который отображается внутри поля. Для установки этого текста применяется функция типа **() -> Unit.** Значение по умолчанию – null;

• leadingIcon: устанавливает иконку, которая отображается перед текстом. Для установки применяется функция типа **() -> Unit**. Значение по умолчанию – null;

• trailingIcon: устанавливает иконку, которая отображается после текста. Для установки применяется функция типа **() -> Unit**. Значение по умолчанию – null;

• isError: указывает, является ли текущее введенное в поле значение некорректным. Представляет значение типа **Boolean.** По умолчанию равно false, то есть введенное значение корректно. Если равно true, то для поля устанавливаютс соответствующие индикаторы – метка, иконка, выделение цветом, которые поддчеркивают, что введенное значение некорректно;

• visualTransformation: объект типа **VisualTransformation**, который задает визуальные трансформации для вводимого текста. Значение по умолчанию –VisualTransformation.None;

• keyboardOptions: объект **KeyboardOptions**, который задает параметры клавиатуры (например, ее тип). Значение по умолчанию – KeyboardOptions.Default;

• keyboardActions: **KeyboardActions**, который задает набор функций, которые вызываются в ответ на некоторые действия пользователя. Значение по умолчанию – KeyboardActions();

• singleLine: устанавливает, будет ли текст однострочным. По умолчанию равно false, то есть поле будет многосточным;

• maxLines: задает максимальное количество строк в поле. По умолчанию равно Int.MAX_VALUE;

• `minLines`: задает минимальное количество строк в поле. По умолчанию равно 1;

• `interactionSource`: объект **MutableInteractionSource**, который задает поток взаимодействий для поля ввода. Значение по умолчанию – `remember { MutableInteractionSource() }`;

• `shape`: представляет объект **Shape**, который задает форму для поля ввода. Значение по умолчанию – `MaterialTheme.shapes.small.copy(bottomEnd = ZeroCornerSize, bottomStart = ZeroCornerSize);`

• `colors`: объект **TextFieldColors**, который задает цвета для поля ввода. Значение по умолчанию – `TextFieldDefaults.textFieldColors()`.

Определение простейшего поля ввода:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            TextField(value = "Hello Work", onValueChange = {})
        }
    }
}
```

При создании поля необходимо задать как минимум два параметра – текущее значение (параметр value) и функцию обработки ввода текста (параметр onValueChange).

В зависимости от задач приложения может потребоваться вводить разную информацию – когда буквы, когда числа и т.д. Для упрощения ввода Jetpack Compose предоставляет тип **KeyboardType**, который позволяет настроить тип клавиатуры с помощью своих свойств:

• `KeyboardType.Ascii`: предоставляет ввод символов ASCII;

• `KeyboardType.Email`: для ввода электронного адреса;

• `KeyboardType.Number`: для ввода цифр;

• `KeyboardType.NumberPassword`: для ввода пароля из цифр;

• `KeyboardType.Password`: для ввода пароля;

• `KeyboardType.Phone`: для ввода номера телефона;

• `KeyboardType.Text`: предоставляет стандартную клавиатуру;

• `KeyboardType.Uri`: предоставляет клавиатуру для ввода URI.

С помощью параметра `keyboardOptions`, который представляет класс **KeyboardOptions**, можно задать тип клавиатуры.

Рисунок 7.2 – Применение клавиатуры для ввода номера телефона

Параметр `leadingIcon` задает иконку перед текстом, а параметр `trailingIcon` –иконку после текста. В качестве значения оба параметра принимают функцию типа () -> Unit. Определим иконки для поля ввода:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val message = remember{mutableStateOf("")}

            TextField(
                message.value,
                {message.value = it},
                textStyle = TextStyle(fontSize =  28.sp),
                leadingIcon     =     {     Icon(Icons.Filled.Check,
contentDescription = "Проверено") },
                trailingIcon     =     {     Icon(Icons.Filled.Info,
contentDescription = "Дополнительная информация") }
            )
        }
    }
}
```

Для определения иконок применяется встроенный тип `androidx.compose.material.Icon`, в функцию которого передается значок иконки. В данном случае применяются встроенные иконки `Icons.Filled.Check` и `Icons.Filled.Info`. Второй параметр – contentDescription позволяет указать к иконке описание.

Параметр `placeholder` устанавливает плейсхолдер или заменитель текста, отображаемый в поле ввода, в виде другого компонента:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val message = remember{mutableStateOf("")}
            TextField(
                message.value,
                {message.value = it},
                textStyle = TextStyle(fontSize =  28.sp),
                placeholder = { Text("Hello Work!") }
            )
        }
    }
}
```

Компонент **OutlinedTextField** во многом похож на `TextField` за тем исключением, что он добавляет границу вокуг поля ввода. Простейшее определение компонента OutlinedTextField:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val message = remember{mutableStateOf("Hello")}
            OutlinedTextField(
                message.value,
                {message.value = it},
                textStyle = TextStyle(fontSize =  30.sp)
            )
        }
    }
}
```

В функции компонента OutlinedTextField мы видим, что в отличие от TextField, меняется стандартное значение для последнего параметра – `colors`, который устанавливает цвета. Теперь он по умолчанию равен компоненту `TextFieldDefaults.outlinedTextFieldColors()`. Фактически здесь мы можем установить те же самые цвета за тем исключением, что мы также можем настроить текст границы с помощью таких параметов как `focusedBorderColor, unfocusedBorderColor, disabledBorderColor и errorBorderColor`, который позволяют установить цвет границы для различных состояний поля ввода.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val message = remember{mutableStateOf("Hello")}
            OutlinedTextField(
                message.value,
                {message.value = it},
                textStyle = TextStyle(fontSize =  30.sp),
                colors = OutlinedTextFieldDefaults.colors(
                    focusedBorderColor= Color(0xff16a085), // цвет при
получении фокуса
                    unfocusedBorderColor = Color(0xffcccccc)  // цвет
при отсутствии фокуса
                )
            )
        }
    }
}
```

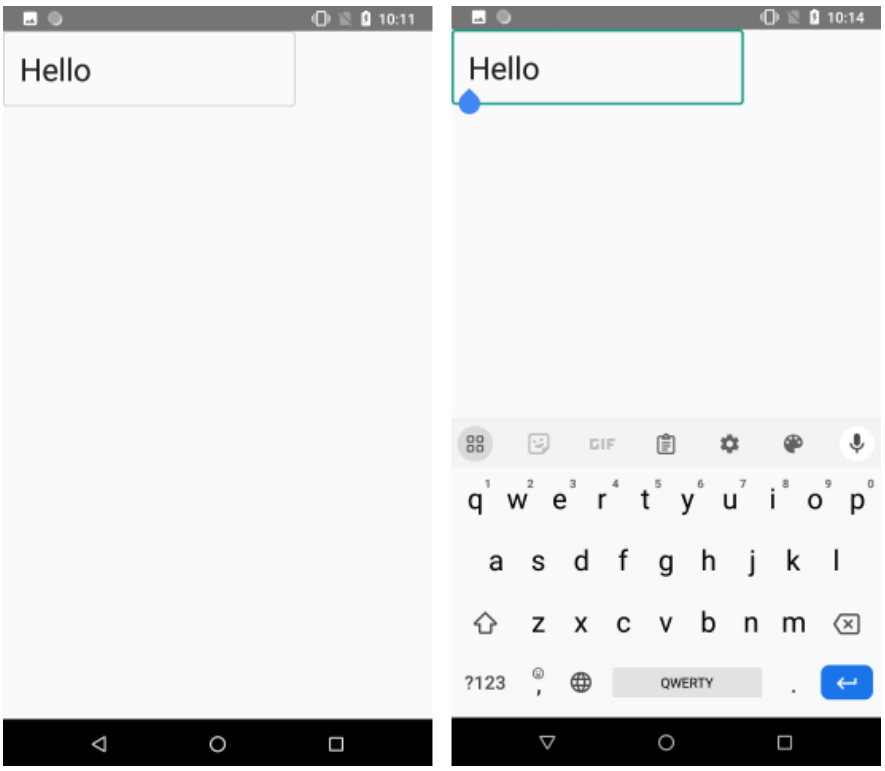


Рисунок 7.3 – Установка цвета границы поля ввода

Модификатор **Modifier.toggleable** устанавливает для компонента два состояния и позволяет переключаться между этими состояниями. Подобные компоненты в этом плане похожи на флажки (checkbox), в которых можно поставить отметку, а можно ее снять, тем самым изменив состояние флажка.

Функция модификатора принимает следующие параметры:

```
fun Modifier.toggleable(
    value: Boolean,
    enabled: Boolean = true,
    role: Role? = null,
    onValueChange: (Boolean) -> Unit
): Modifier
```

- value хранит состояние компонента в виде объекта Boolean, поэтому состояние может принимать только два значения: true и false;
- enabled указывает, будет ли компонент доступен для выбора. Если он имеет значение true (значение по умолчанию), то компонент будет доступен;
- role представляет объект **Role**, который представляет тип элемента интерфейса;
- onValueChange представляет функцию типа (Boolean) -> Unit, которая вызывается при нажатии на компонент.

Создадим переключаемый компонент на основе компонента Text:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            var checked = remember { mutableStateOf(false) }
            Text(
                modifier = Modifier.toggleable(value = checked.value,
onValueChange = { checked.value = it }),
                text = checked.value.toString(),
                fontSize = 30.sp
            )
        }
    }
}
```

Для отслеживания состояния создается переменная checked, которая по умолчанию хранит значение `false`. Это значение передается в функцию модификатора toggleable параметру value. А с помощью другого параметра – `onValueChange` задаем функцию, которая получает новое состояние через параметр it и передает его переменной checked. Таким образом, при нажатии состояние компонента будет переключаться.

Компонент **Checkbox** представляет флажок, который может быть в отмеченном и неотмеченном состоянии. Его функция принимает следующие параметры:

```
@Composable
fun Checkbox(
    checked: Boolean,
    onCheckedChange: ((Boolean) -> Unit)?,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    colors: CheckboxColors = CheckboxDefaults.colors(),
    interactionSource: MutableInteractionSource? = null
): Unit
```

- `checked`: указывает, будет ли отмечен флажок. Представляет значение `Boolean`. Если равен `true`, то флажок отмечен;
- `onCheckedChange`: представляет функции типа `(Boolean)` -> Unit, которая выполняется при изменении состояния флажка (установки или снятия отметки). В качестве параметра в функцию передается новое состояние флажка;
- `modifier`: объект **Modifier**, который устанавливает для флажка модификаторы;
- `enabled`: указывает, будет ли доступен флажок. Представляет значение Boolean и по умолчанию равен true (то есть флажок будет доступен);
- `interactionSource`: объект **MutableInteractionSource**, который задает поток взаимодействий для флажка. По умолчанию равен `remember { MutableInteractionSource() };`
- `colors`: объект **CheckboxColors**, который задает цвета для флажка. По умолчанию равен `CheckboxDefaults.colors()`.

Простейший флажок:

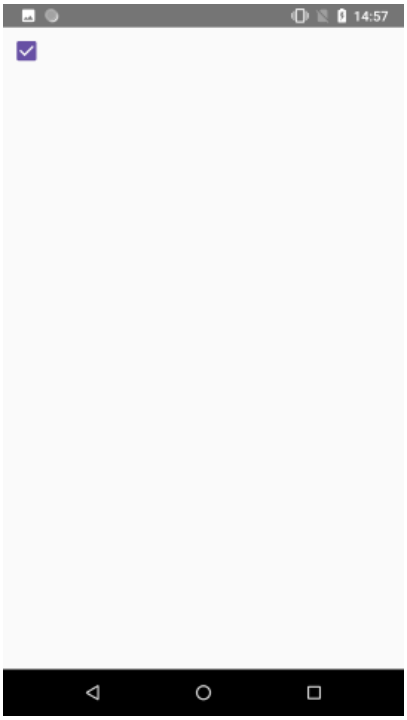


Рисунок 7.4 – Флаг

Здесь надо отметить два момента. Прежде всего для компонента Checkbox необходимо задать как минимум два параметра: `checked` и `onCheckedChange`. Для этого в примере выше определена переменная `checkedState`, которая представляет состояние флажка (отмечен или нет). Эта переменная представляет объект типа **MutableState<Boolean>**, который создается функцией **mutableStateOf()**. В эту функцию передается собственно хранимое значение – в данном случае значение `true`, которое затем можно получить с помощью свойства `value` объекта `MutableState<T>`. А функция **remember** позволяет сохранить это значение.

В коде с помощью свойства `value` мы привязываем значение переменной к свойству `checked`:

```
checked = checkedState.value
```

А в функции обработки изменения состояния флажка с помощью параметра it передаем в переменную новое состояние флажка (если отмечен – `true`, если отметка отсутвует – `false`):

```
onCheckedChange = { checkedState.value = it }
```

Второй момент, который надо отметить, флажок не предоставляет встроенных возможностей по установке текстовой метки. Однако, как правило, при флажке идет некоторый текст, который некоторым образом объясняет назначение флажка. Но в Jetpack Compose подобную текстовую метку нам надо устанавливать дополнительно.

Компонент **TriStateCheckbox** расширяет Checkbox, добавляя возможность установить флажок в третье – неопределенное состояние.

```
@Composable
fun TriStateCheckbox(
    state: ToggleableState,
    onClick: (() -> Unit)?,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    colors: CheckboxColors = CheckboxDefaults.colors(),
    interactionSource: MutableInteractionSource? = null
): Unit
```

Он принимает почти те же параметры, что и Checkbox, за исключением двух параметров. Так, параметр `state` представляет тип **ToggleableState** и устанавливает состояние флажка с помощью следующих значений:

- `ToggleableState.Indeterminate`: неопределенное состояние;
- `ToggleableState.Off`: отметка снята;
- `ToggleableState.On`: флажок отмечен.

Модификатор **Modifier.selectable** позволяет сделать компонент **выбираемым** (или выделяемым). Данный модификатор имеет следующие параметры:

```
fun Modifier.selectable(
    selected: Boolean,
    enabled: Boolean = true,
    role: Role? = null,
    onClick: () -> Unit
): Modifier
```

- `selected` указывает, будет ли компонент выбран. Если он имеет значение true, то компонент выбран;
- `enabled` указывает, будет ли компонент доступен для выбора. Если он имеет значение true (значение по умолчанию), то компонент будет доступен;
- `role` представляет тип элемента графического интерфейса;
- `onClick` представляет функцию типа () -> Unit, которая вызывается при нажатии на компонент.

Параметр selected модификатора, который указывает, будет ли выбран компонент, получает значение из переменной `selected`, которая представляет объект `MutableState<Boolean>`, то есть значение `selected.value` будет равно `true` или `false`.

А параметр `onClick`, который задает функцию, выполняемую при нажатии на компонент, будет переключать значение `selected.value`. То есть если это значение было равно `true`, оно становится равным `false` и наоборот.

Компонент **RadioButton** представляет переключатель или радиокнопку и служит для создания группы радиокнопок, из которых одномоментно можно выбрать только один переключатель. Этот компонент имеет следующие параметры:

```
@Composable
fun RadioButton(
    selected: Boolean,
    onClick: (() -> Unit)?,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    colors: RadioButtonColors = RadioButtonDefaults.colors(),
    interactionSource: MutableInteractionSource? = null
): Unit
```

- `selected`: указывает, будет ли отмечена радиокнопка. Представляет значение `Boolean`. Если равен `true`, то радиокнопка отмечена.
- `onClick`: представляет функцию типа () -> Unit, которая выполняется при нажатии на радиокнопку.
- `modifier`: объект **Modifier**, который устанавливает для радиокнопки модификаторы
- `enabled`: указывает, будет ли доступна радиокнопка. Представляет значение Boolean и по умолчанию равен true (то есть радиокнопка будет доступна).
- `interactionSource`: объект **MutableInteractionSource**, который задает поток взаимодействий для радиокнопки. По умолчанию равен `remember { MutableInteractionSource() }`.
- `colors`: объект **RadioButtonColors**, который задает цвета для радиокнопки. По умолчанию равен `RadioButtonDefaults.colors()`.

РЕСУРСЫ В JETPACK COMPOSE

Кроме файлов кода на языке Kotlin проект может включать дополнительные файлы, которые используются в приложении, например, файлы изображений, определения иконок, файлы xml и так далее. Подобные файлы называются **ресурсами**. Все ресурсы находятся в проекте в каталоге **res.**

Одним из наиболее применяемых типов ресурсов являются ресурсы строк. Они используются при выведении названия приложения, различного текста, например, текста кнопок и т.д. Например, нам нужно вывести в различных местах приложения один и тот же текст. Вместо того, чтобы по несколько раз определять его в коде Kotlin, мы можем определить его один раз в виде строкового ресурса и использовать в любом месте приложения.

По умолчанию строковые ресурсы хранятся в проекте в каталоге **res/values.** При создании проекта в этот каталог по умолчанию добавляется файл строковых ресурсов **strings.xml**.

Если мы откроем файл, то мы найдем в нем строки наподобие следующих:

```
<resources>
    <string name="app_name">helloapp</string>
</resources>
```

Каждый отдельный строковый ресурс определяется с помощью элемента **string**, а его атрибут **name** содержит название ресурса. Так, в самом простом виде этот файл определяет один ресурс "app_name", который устанавливает название приложения и который в моем случае имеет значение "helloapp" (по умолчанию значение этого ресурса соответствует названию проекта). Но естественно мы можем определить любые строковые ресурсы.

Затем в приложении в файлах кода мы можем ссылаться на эти ресурсы через их идентификатор, который имеет следующий вид:

```
R.string.название_ресурса
```

Например, для обращения к определенному по умолчанию строковому ресурсу app_name применяется идентификатор.

```
R.string.app_name
```

Чтобы получить строковый ресурс в коде Kotlin, применяется встроенная функция **androidx.compose.ui.res.stringResource()**, в которую передается идентификатор ресурса и которая возвращает строку в виде объекта String. Например, получим в коде ресурс app_name:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(
                text = stringResource(R.string.app_name),
                fontSize = 28.sp
            )
        }
    }
}
```

Подобным образом при необходимости мы можем определять и свои ресурсы.

Функция **stringResource()** позволяет применять к ресурсам строк форматирование. Преимуществом форматирования является то, что мы можем определить общий щаблон и затем подставлять в него необходимые значения. Например, изменим файл **strings.xml**:

```
<resources>
    <string name="app_name">helloapp</string>
    <string name="user_data">Имя: %1$s. Возраст: %2$d. Компания: %3$s</string>
</resources>
```

Здесь ресурс `user_data` представляет строку с форматированием. Так, она содержит такие символы как %1$s, %2$d и %3$ds. Что они означают? %1$s указывает, что это первый аргумент, а символ "s" говорит, что этот аргумент представляет строку. %2$d представляет второй аргумент, а символ "d" в конце указывает, что это будет целое число. Аналогично %3$s указывает, что это третий аргумент, который представляет строку.

Получим ресурс в коде Kotlin:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column{
                Text(
                    text = stringResource(R.string.user_data, "Tom", 37, "JetBrains"),
                    fontSize = 28.sp
                )
                Text(
                    text = stringResource(R.string.user_data, "Bob", 41, "Google"),
                    fontSize = 28.sp
                )
            }
        }
    }
}
```

Вызов функции stringResource(R.string.user_data, "Tom", 37, "JetBrains") получает ресурс "user_data" и в качестве последующих параметров передает в строку вставляемые значения. Так, вместо первого аргумента − строки в user_data вставляется второй аргумент функции − строка "Tom", вместо второго аргумента-числа в user_data вставляется число 37, а вместо третьего аргумента в user_data передается строка "JetBrains".

Еще один тип ресурсов представляют массивы строк в виде элемента **string-array**. Его можно использовать, когда надо определить какой-нибудь набор строк.

Для получения массива применяется метод **getStringArray()**, в который передается имя ресурса:

```
val items = resources.getStringArray(R.array.languages)
```

Возвращаемый объект представляет массив типа `Array<String>`, который, например, можно перебрать и вывести в виде компонентов Text в приложении.

Ресурсы **dimension** задают размеры. Определение размеров должно находиться в папке `res/values` в файле с любым произвольным именем. Общий синтаксис определения ресурса следующий:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="имя_ресурса">используемый_размер</dimen>
</resources>
```

Как и другие ресурсы, ресурс dimension определяется в корневом элементе `<resources>`. Тег `<dimen>` обозначает ресурс и в качестве значения принимает некоторое значение размера в единицах **dp**.

Чтобы получить ресурс dimen в коде Kotlin, применяется встроенная функция **androidx.compose.ui.res.dimensionResource()**, в которую передается идентификатор ресурса и которая возвращает объект **Dp**.

Ресурсы Color хранят определения цветов. Они должны храниться в проекте в каталоге **res/values** и также, как и ресурсы строк, заключены в тег <resources>. Так, по умолчанию при создании самого простого проекта в папку **res/values** добавляется файл **colors.xml.**

По умолчанию он имеет следующее определение:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
    <color name="purple_700">#FF3700B3</color>
    <color name="teal_200">#FF03DAC5</color>
    <color name="teal_700">#FF018786</color>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
</resources>
```

Цвет определяется с помощью элемента <color>. Его атрибут name устанавливает название цвета, которое будет использоваться в приложении, а шестнадцатеричное число – значение цвета.

Для задания цветовых ресурсов можно использовать следующие форматы:

- #RGB (#F00 −12-битное значение);
- #ARGB (#8F00 −12-битное значение с добавлением альфа-канала);
- #RRGGBB (#FF00FF −24-битное значение);
- #AARRGGBB (#80FF00FF − 24-битное значение с добавлением альфа-канала).

Чтобы получить ресурс color в коде Kotlin, применяется встроенная функция **androidx.compose.ui.res. colorResource()**, в которую передается идентификатор ресурса и которая возвращает объект **Color**.

РАБОТА С ИЗОБРАЖЕНИЯМИ

Для выведения изображений в приложении в Jetpack Compose предназначен компонент **Image**. Он имеет ряд версий. Первая версия компонента:

```
@Composable
fun Image(
    bitmap: ImageBitmap,
    contentDescription: String?,
    modifier: Modifier = Modifier,
    alignment: Alignment = Alignment.Center,
    contentScale: ContentScale = ContentScale.Fit,
    alpha: Float = DefaultAlpha,
    colorFilter: ColorFilter? = null,
    filterQuality: FilterQuality = DefaultFilterQuality
): @Composable Unit
```

Параметры функции компонента:

- `bitmap`: объект типа `ImageBitmap`, которое, собственно, представляет изображение для отрисовки;
- `contentDescription`: представляет строковое описание для изображения, которое применяется сервисами `accessibility` в служебных целях;
- `modifier`: представляет объект **Modifier**, который определяет модификаторы компонента;
- `alignment`: представляет объект типа **Alignment**, которое задает выравнивание изображения. Значение по умолчанию – `Alignment.Center`;
- `contentScale`: объект типа **ContentScale**, который принцип масштабирования изображения. По умолчанию равно `ContentScale.Fit`.

Может принимать следующие значения:

- `ContentScale.Crop`: масштабирует изображение с сохранением аспектного отношения (отношение высоты и ширины) таким образом, что ширина и высота оказываются равными или больше сторон контейнера;
- `ContentScale.FillBounds`: неравномерно масштабирует изображение для полного заполнения пространства контейнера;
- `ContentScale.FillHeight`: масштабирует изображение с сохранением аспектного отношения таким образом, что высота изображения равна высоте контейнера;

- `ContentScale.FillWidth`: масштабирует изображение с сохранением аспектного отношения таким образом, что ширина изображения равна ширине контейнера;
- `ContentScale.Fit`: масштабирует изображение с сохранением аспектного отношения (отношение высоты и ширины) таким образом, что ширина и высота оказываются равными или меньше сторон контейнера;
- `ContentScale.Inside`: масштабирует изображение с сохранением аспектного отношения (отношение высоты и ширины) таким образом, чтобы вместить изображение внутри контейнера, если ширина и(или) высота изображения больше ширины и(или) высоты контейнера;
- `ContentScale.None`: масштабирование отсутствует.

- `alpha`: задает прозрачность изображения в виде значения типа `Float`;
- `colorFilter`: устанавливает применяемые к изображению цветовые фильтры в виде объекта **ColorFilter;**
- `filterQuality`: задает алгоритм выборки пикселей из изображения. Представляет объект типа **FilterQuality** и по умолчанию имеет значение `FilterQuality.Low`.

Изображение, которое мы хотим отобразить в нашем приложении, может располагаться в различных местах – это может быть сетевой проект, но также изображение может храниться в самом приложении. Для хранения изображений в проекте предназначена папка **res/drawable.**

По умолчанию в этой папке уже имеется определения векторной графики в виде файлов `ic_launcher_background.xml` и `ic_launcher_foreground.xml`, которые применяются для создания иконок приложения.

Теперь добавим в эту папку какой-нибудь файл изображения. Для этого скопируем файл изображения с расширением png или jpg и с помощью стандартной комбинации клавиш Ctrl+V добавим его в папку **res/drawable**. Также можно нажать на папку правой кнопкой мыши и в появившемся меню выбрать пункт **Paste.**

При добавлении графических файлов в эту папку для каждого из них Android создает ресурс **Drawable**. После этого мы можем обратиться к ресурсу следующим образом в коде Kotlin:

```
R.drawable.имя_файла
```

Для отображения растровых изображений, а именно файлов png и jpg, в компоненте Image применяется интерфейс **ImageBitmap**. Этот интерфейс предоставляет статический метод `imageResource(идентификатор_ресурса)` для получения объекта ImageBitmap из ресурса drawable.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Image(
                bitmap = ImageBitmap.imageResource(R.drawable.forest),
                contentDescription = "Зимний лес"
            )
        }
    }
}
```

Также для выведения изображения можно использовать класс Painter, а точнее его класс-наследник **BitmapPainter**, который отрисовывает изображение. Данное изображение передается в виде объекта ImageBitmap в конструктор `BitmapPainter` в качестве параметра:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Image(
                painter = BitmapPainter(ImageBitmap.imageResource(R.drawable.forest)),
                contentDescription = "Зимний лес"
            )
        }
    }
}
```

Для отображения векторной графики в компоненте Image у этого компонента применяется параметр `imageVector`, который представляет объект класса **ImageVector**.

Например, встроенные в Jetpack Compose иконки как раз представляют векторную графику, которую мы можем вывести в компоненте Image:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Image(
                imageVector = Icons.Filled.Email,
                contentDescription = "Значок электронной почты",
                modifier = Modifier.size(200.dp, 150.dp)
            )
        }
    }
}
```

Также мы можем использовать **ImageVector** для загрузки векторной графики, которая хранится в ресурсах приложения. Для хранения векторной графики, как и вообще изображений, в проекте предназначена папка **res/drawable.**

АРХИТЕКТУРНЫЕ ШАБЛОНЫ И ПАТТЕРНЫ

При работе с современной архитектурой Android-приложений, компоненты ViewModel из библиотеки Jetpack стали золотым стандартом для управления данными UI. Они решают классическую проблему потери состояния при изменении конфигурации (например, при повороте экрана). Однако, разработчики часто сталкиваются с выбором между двумя классами: ViewModel и его подклассом AndroidViewModel. Посмотрим, чем они различаются.

ViewModel − это класс, предназначенный для хранения и управления данными, связанными с пользовательским интерфейсом, с учётом жизненного цикла. Его основная задача − пережить изменения конфигурации. Когда объекты Activity или Fragment пересоздаются, экземпляр ViewModel, связанный с ними, остаётся в памяти, сохраняя все свои данные. Новый экземпляр Activity просто подключается к уже существующей ViewModel.

Но ViewModel ничего не знает о фреймворке Android Framework и не имеет доступа к Context или любым другим специфичным для Android классам.

Простейший пример ViewModel:

```
import androidx.lifecycle.ViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow

class GameViewModel : ViewModel() {
    private val _score = MutableStateFlow(0)
    val score: StateFlow<Int> = _score

    fun onCorrectAnswer() {
        _score.value += 10
    }
}
```

Этот GameViewModel содержит только чистую логику и данные, без каких-либо Android-зависимостей.

AndroidViewModel – это специальный подкласс ViewModel, который знает о существовании Application. Основная задача AndroidViewModel та же, что и у ViewModel – сохранять данные при изменении конфигурации, но с одним дополнением: он предоставляет доступ к Context приложения. Но ключевое отличие заключается в том, что конструктор типа AndroidViewModel принимает объект Application, который можно использовать для получения applicationContext.

Пример AndroidViewModel:

```
// Зависимость та же, что и для ViewModel
import android.app.Application
import androidx.lifecycle.AndroidViewModel
 class      UserProfileViewModel(application:      Application)      :
AndroidViewModel(application) {

    // содержимое класса
}
```

Нередко приложению приходится обращаться к ресурсам строк, изображений и т.д., которые определены в проекте. Например, пусть в файле **string.xml** определена некоторая строка с ключом "welcome_user". Благодаря передаче объекта Application в конструктор мы могли бы получить этот ресурс:

```
import android.app.Application
import androidx.lifecycle.AndroidViewModel
import com.yourpackage.R // Импорт R-класса для доступа к ресурсам

class      UserProfileViewModel(application:      Application)      :
AndroidViewModel(application) {

    fun getWelcomeMessage(userName: String): String {
        // Получаем доступ к ресурсам по требованию
        val                    welcomeFormat                    =
getApplication<Application>().getString(R.string.welcome_user)
        return String.format(welcomeFormat, userName)
    }
}
```

Внутри класса AndroidViewModel доступен метод getApplication<Application>(), который возвращает значение Context и через который мы можем получить доступ к ресурсам проекта. Например, для получения строки применяется метод `getString()`, в который передается идентификатор ресурса строки.

То есть, иными словами, AndroidViewModel удобно применять, когда необходим доступ к приложению и его ресурсам, что по умолчанию не доступно из стандартного класса ViewModel.

Но здесь надо помнить, что даже при использовании AndroidViewModel не следует хранить Context в поле ViewModel. Всегда получайте его по требованию через вызов `getApplication()`, чтобы избежать потенциальных утечек и предупреждений от статического анализатора.

Таблица 10.1− Сравнение ViewModel и AndroidViewModel

Характеристика	**ViewModel**	**AndroidViewModel**
Базовый класс	androidx.lifecycle.ViewModel	androidx.lifecycle.AndroidViewModel (наследуется от ViewModel)
Зависимость от Android	Нет. Полностью независим.	Да. Зависит от класса android.app.Application.
Конструктор	Пустой конструктор по умолчанию.	Требует Application в качестве параметра.
Доступ к Context	Нет прямого доступа.	Да. Через метод getApplication().
Тестируемость	Очень легко. Можно использовать стандартные юнит-тесты.	Сложнее. Требует имитации Context в юнит-тестах.

РАБОТА С БАЗАМИ ДАННЫХ

База данных является наиболее распространенным местом, где мобильное приложение хранит свои данные. Наиболее популярной с системой управления базами данных в мобильном приложении является SQLite. И Android SDK предоставляет необходимый функционал для работы с SQLite. Конечно, прежде чем углубляться в особенности SQLite в контексте разработки под Android, необходимо иметь базовое понимание запросов SQL.

Базы данных SQLite размещаются в файлах (обычно с расширением *.db), которые зачастую расположены по следующему пути:

```
/data/data/<имя_пакета>/databases/<имя_базы_данных>.db
```

Например, если пакет приложения называется "com.example.helloapp", и оно использует базу данных "mydatabase.db", то путь к файлу базы данных будет следующим:

```
/data/data/com.example.helloapp/databases/mydatabase.db
```

Для работы с SQLite Jetpack Compose имееет специальную библиотеку **Room**, которая предоставляет высокоуровневый интерфейс поверх системы базы данных SQLite и которая значительно упрощает взаимодействие с базами данными. Вся схема взаимодействия с базой данных выглядит следующим образом:

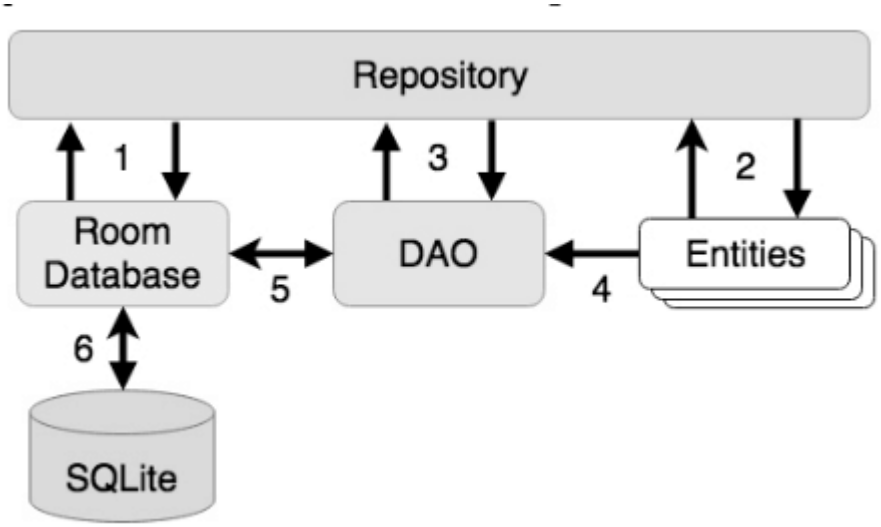


Рисунок 11.1− Схема взаимодействия с базой данных

Прежде всего для получения данных из базы данных обычно определяется отдельный модуль репозитория. Однако код приложения, в частности, репозиторий не обращается напрямую к базе данных. Вместо этого все операции с базой данных выполняются с использованием комбинации базы данных Room, DAO и сущностей. DAO (Data Access Object − объект доступа к данным) содержит операторы SQL, необходимые репозиторию для добавления, извлечения и удаления данных в базе данных SQLite. Эти инструкции SQL сопоставляются с методами, которые затем вызываются из репозитория для выполнения соответствующего запроса.

Объект базы данных Room предоставляет интерфейс к базовой базе данных SQLite. Он также предоставляет репозиторию доступ к объекту DAO. Приложение должно иметь только один экземпляр базы данных Room, который могут использовать для доступа к нескольким таблицам базы данных.

Для определения данных, которые хранятся в отдельной таблице, применяются специальные классы − **сущности** (entity). Сущности определяют схему таблицы в базе данных, имя таблицы, ее столбцы и сопутствующую информацию. Когда репозиторий получает данные базы данных через DAO, то эти данные как раз представляют объекты классов сущностей. Аналогично, когда репозиторию необходимо добавить в базу данных новые данные, он создает объект сущности и затем вызывает методы добавления, определенные в DAO.

В итоге все взаимодействие с базой данных выглядит следующим образом:

1. Репозиторий взаимодействует с базой данных Room для получения объекта базы данных, который, в свою очередь, используется для получения ссылок на объект DAO;

2. Репозиторий создает объекты сущностей и настраивает их с данными перед передачей их в DAO;

3. Репозиторий вызывает методы DAO, передавая в них объекты сущностей, которые надо добавить и т.д.;

4. Когда объект DAO получает результат запроса для передачи в репозиторий, он упаковывает эти результаты в объекты сущностей;

5. Объект DAO взаимодействует с базой данных Room для выполнения операций с базой данных и обработки результатов;

6. База данных Room обрабатывает все низкоуровневые взаимодействия с базой данных SQLite, отправляя запросы и получая результаты.

Перед началом работы с Room, необходимо настроить конфигурацию проекта и добавить все необходимые зависимости. Первым шагом является добавление плагина ksp и дополнительных библиотек в конфигурацию сборки Gradle. Для этого перейдем в проекте к файлу **libs.versions.toml** и изменим его следующим образом:

```
[versions]
...................
roomRuntime = "2.6.1"
runtimeLivedata = "1.6.5"
ksp = "1.9.0-1.0.13"

[libraries]
.............................
androidx-lifecycle-viewmodel-compose = { module = "androidx.lifecycle:lifecycle-viewmodel-compose", version.ref = "lifecycleRuntimeKtx" }
androidx-room-ktx = { module = "androidx.room:room-ktx", version.ref = "roomRuntime" }
androidx-room-room-compiler = { module = "androidx.room:room-compiler", version.ref = "roomRuntime" }
androidx-room-runtime = { module = "androidx.room:room-runtime", version.ref = "roomRuntime" }
androidx-runtime-livedata = { module = "androidx.compose.runtime:runtime-livedata", version.ref = "runtimeLivedata" }


[plugins]
.....................................
devtoolsKsp = { id = "com.google.devtools.ksp", version.ref = "ksp"}
```

Далее для добавления плагина ksp изменим файл **build.gradle.kts** уровня проекта ("build.gradle.kts (Project: HelloApp)") следующим образом:

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.
plugins {
    alias(libs.plugins.androidApplication) apply false
    alias(libs.plugins.jetbrainsKotlinAndroid) apply false
    alias(libs.plugins.devtoolsKsp)
}
```

И далее отредактируем файл **build.gradle.kts** уровня модуля ("build.gradle.kts (Module :app)") следующим образом:

```
plugins {
    alias(libs.plugins.androidApplication)
    alias(libs.plugins.jetbrainsKotlinAndroid)
    alias(libs.plugins.devtoolsKsp)
}

.....................

dependencies {

    implementation(libs.androidx.room.runtime)
    implementation(libs.androidx.room.ktx)
    implementation (libs.androidx.runtime.livedata)
    implementation(libs.androidx.lifecycle.viewmodel.compose)
    annotationProcessor(libs.androidx.room.room.compiler)
    ksp(libs.androidx.room.room.compiler)

    ......................................
}
```

Затем синхронизируем проект, нажав на кнопку **Sync Now.**

Центральным звеном в организации взаимодействия с базой данных через Room является **сущность**. С каждой таблицей базы данных связан определенный класс сущности, который определяет схему таблицы. Синтаксически сущность представляет стандартный класса Kotlin с некоторыми специальными аннотациями Room. Например:

```
@Entity(tableName = "users")
class User {
    @PrimaryKey(autoGenerate = true)
    @NonNull
    @ColumnInfo(name = "userId")
    var id: Int = 0
    @ColumnInfo(name = "userName")
    var name: String? = null
    var age: Int? = null
```

```
    constructor() {}

    constructor(id: Int, name: String, age: Int) {
        this.id = id
        this.name = name
        this.age = age
    }
    constructor(name: String, age: Int) {
        this.name = name
        this.age = age
    }
}
```

Определение сущности начинается с аннотации **@Entity**, которая настраивает ряд метаданных для сущности. В частности, параметр `tableName` указывает, с какой таблицей в базе данных будет сопоставляться эта сущность:

```
@Entity(tableName = "users")
class User {
```

И здесь мы видим, что класс сущности User будет сопоставляться в базе данных с таблицей "users".

Свойства класса сопоставляются с определенными столбцами. По умолчанию сопоставление между свойствами классов и столбцами в таблице выполняется по имени. Например, свойство age не имеет никаких аннотаций:

```
var age: Int? = null
```

Поэтому данные этого свойства будут храниться в таблице базы данных в одноименном столбце "age". Однако с помощью аннотации **@ColumnInfo** это можно переопределить с помощью параметра name:

```
@ColumnInfo(name = "userName")
var name: String? = null
```

Каждой таблице базы данных необходим столбец, который будет выступать в качестве первичного ключа. Для установки свойства как первичного ключа применяется аннотация **@PrimaryKey**. В примере выше в качестве такого столбца выступает **userId**. И с этим столбцом будет сопоставляться свойство id:

```
@PrimaryKey(autoGenerate = true)
@NonNull
@ColumnInfo(name = "userId")
var id: Int = 0
```

Кроме того, с помощью параметра autoGenerate = true аннотации @PrimaryKey значение id настроено на автоматическую генерацию. Это значит, что система автоматически сгенерирует идентификатор, присваиваемый новым добавляемым объектам, чтобы избежать дублирования ключей.

Кроме того, к свойству id применяется аннотация **@NonNull**, которая указывает, что свойство (и соответственно столбец в таблице) не может хранить значение null/NULL.

Если мы не хотим, чтобы данные какого-то столбца хранились в базе данных, то мы можем к соответствующему свойству применить аннотацию **@Ignore**:

```
@Ignore
var age: Int? = null
```

Объект доступа к данным или DAO (Data Access Object) предоставляет способ доступа к данным, хранящимся в базе данных SQLite. DAO объявляется как стандартный интерфейс Kotlin с помощью аннотации **@Dao**:

```
@Dao
interface UserDao {

}
```

Интерфейс DAO может содержать функции, которые с помощью дополнительных аннотаций сопоставляются с конкретными инструкциями SQL. Подобные функции затем может вызывать репозиторий. Например, определим метод `getUsers()` для получения всех объектов User из таблицы:

```
@Dao
interface UserDao {
    @Query("SELECT * FROM users")
    fun getUsers(): LiveData<List<User>>
}
```

В данном случае вызов метода `getUsers()` приведет к выполнению SQL-выражения "SELECT * FROM users". Этот метод возвращает объект List, который содержит объекты сущности User для каждой строки таблицы. DAO также использует **LiveData**, чтобы репозиторий мог отслеживать изменения в базе данных.

Методы интерфейсов также могут принимать параметры, на которые затем можно ссылаться в выражениях SQL в качестве аргументов. Например:

```
@Query("SELECT * FROM users WHERE name = :userName")
fun getUser(userName: String): List<User>
```

Данный метод получает через параметр userName имя пользователя и ищет по этому имени всех пользователей. А для вставки значения параметра в SQL-выражение применяется двоеточие **:**.

Для сопоставления метода интерфейса с SQL-выражением добавления данных может применяться специальная аннотация **@Insert**:

```
@Insert
fun addUser(user: User)
```

В подобном случае библиотека Room может определить, что сущность User, переданная методу `addUser()`, должна быть добавлена в базу данных. Причем эта аннотация позволяет добавлять сразу несколько объектов в рамках одной транзакции:

```
@Insert
fun insertUsers(User... userList)
```

База данных Room представляет слой поверх фактической базы данных SQLite, который отвечает за предоставление доступа к экземплярам DAO,

связанным с базой данных. Каждое приложение Android должно иметь только один экземпляр базы данных Room.

Синтаксически база данных Room представляет класс, который наследуется от **RoomDatabase**. Например:

```
@Database(entities = [(User::class)], version = 1)
abstract class UserRoomDatabase: RoomDatabase() {

    abstract fun UserDao(): UserDao

    // реализуем синглтон
    companion object {
        private var INSTANCE: UserRoomDatabase? = null
        fun getInstance(context: Context): UserRoomDatabase {

            synchronized(this) {
                var instance = INSTANCE
                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        UserRoomDatabase::class.java,
                        "User_database"

                    ).fallbackToDestructiveMigration().build()
                    INSTANCE = instance
                }
                return instance
            }
        }
    }
}
```

К классу базы данных Room применяется аннотация **@Database**, которая объявляет сущности, с которыми должна работать база данных. В остальном класс содержит фактически реализацию паттерна синглтон, который гарантирует, что одномоментно может быть только один объект этого класса.

РАБОТА С СЕТЬЮ

Создание современных Android-приложений практически невозможно представить без взаимодействия с сетью. Получение данных с сервера, отправка информации и аутентификация − все это требует надежного и удобного инструмента для выполнения сетевых запросов. В экосистеме Kotlin одним из таких инструментов является **Ktor Client**. **Ktor** − это асинхронный фреймворк от JetBrains для создания как серверов, так и клиентов на языке Kotlin. **Ktor Client** представляет собой мощную и гибкую библиотеку для выполнения HTTP-запросов. Его ключевые преимущества:

- **Полностью на Kotlin**: идеально интегрируется с корутинами, что делает асинхронный код чистым и читаемым.
- **Мультиплатформенность**: Ktor Client можно использовать не только на Android, но и на iOS, JVM, и в нативных приложениях.
- **Расширяемость**: имеет богатую систему плагинов для таких задач, как аутентификация, сериализация JSON, логирование и многое другое.

Для работы с Ktor в Android прежде всего необходимо добавить зависимости Ktor Client в проект. Для этого откроем файл **build.gradle.kts** модуля app и добавим в него в узел "dependencies" следующие зависимости:

```
dependencies {

    // Ktor Core
    implementation("io.ktor:ktor-client-core:3.2.1")

    // Движок для Android
    implementation("io.ktor:ktor-client-android:3.2.1")

    // ... другие зависимости
}
```

И также в файл **AndroidManifest.xml** добавим разрешение на доступ в интернет:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Допустим, нам надо при загрузке приложения загрузить некоторые данные. В качестве тестового адреса используем адрес "https://google.com", то есть будет загружать код главной страницы google. Для этого определим в файле **MainActivity.kt** следующий код:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            var text by remember { mutableStateOf("Loading") }

            val client = HttpClient()  // создаем HttpClient для обработки запроса

            LaunchedEffect(true) {    // при старет приложения выполняется запрос
                try {
                    val response = client.get("https://google.com/") // выполняем запрос к google.com
                    text =  response.bodyAsText()       // считываем текст ответа
                } catch (e: Exception) {
                    e.localizedMessage ?: "error"
                }
                finally {
                    client.close()  // закрываем клиент
                }
            }
            Text(text = text, Modifier.padding(10.dp), fontSize = 16.sp)
        }
    }
}
```

Итак, здесь визуальный интерфейс представлен одним виджетом –Text, который вначале выводит сообщение о загрузке – строку "Loading", а затем после завершения загрузки – html-код, полученный от google.com.

Вначале создаем объект **HttpClient**:

```
val client = HttpClient()
```

Функции HttpClient, которые выполняют запросы (в данном случае функция `get()`) обычно являются **suspend**-функциями и поэтому должны выполняться из других **suspend**-функций или корутин. В данном же случае используем втроенный компонент **LaunchedEffect**, который при запуске приложения немедленно запускает корутину.

```
LaunchedEffect(true) {  // при старет приложения выполняется запрос
.........................
}
```

Поскольку при выполнении запроса могут произойти различные ошибки, то помещаем выполнение запроса в блок try..catch. В блоке try выполняем запрос с

помощью метода get(), получаем ответ в текстовой форме и передаем его в переменную text:

```
val response = client.get("https://google.com/") // выполняем запрос к google.com
text =  response.bodyAsText()       // считываем текст ответа
```

После завершения запроса (успешного или неудачного) в блоке finally закрываем HttpClient:

```
finally {
    client.close()  // закрываем клиент
}
```

Стоит отметить, что **HttpClient** – очень тяжелый объект. Его создание и последующее освобождение требует много ресурсов. Так, в примере выше при нажатии на кнопку создается объект HttpClient. Если нужно снова выполнить запрос, мы снова создаем объект HttpClient и так далее.

В случае с первым примером, где HttpClient создается и применяется в LaunchedEffect, то даже при том, что HttpClient создается один раз, при рекомпозициях HttpClient может повторно создаваться.

И если HttpClient нужен всему приложению или некоторому наболу Activity, то HttpClient можно сделать **синглтоном** с жизненным циклом всего приложения. Это самый эффективный способ, так как клиент создается один раз при запуске приложения и используется всеми экранами:

```
object AppHttpClient {
    val instance = HttpClient(Android)
}
```

Кроме того, оптимальнее привязать HttpClient к **ViewModel**: создавать клиент при инициализации ViewModel и закрывать его в методе `onCleared()`. Это гарантирует, что клиент переживет повороты экрана и будет корректно закрыт, когда экран больше не нужен. Для использования **ViewModel** добавим в **build.gradle.kts** соответствующую зависимость:

```
implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.9.1")
```

II. ПРАКТИЧЕСКИЙ РАЗДЕЛ

ПЛАНЫ И ЗАДАНИЯ ЛАБОРАТОРНЫХ РАБОТ

Лабораторная работа 1. Основы языка Kotlin

Цель работы: изучить основы языка Kotlin.

Теоретическая часть

Условные конструкции

Инструкция if в Kotlin выполняет блок кода, если указанное условие истинно. Если условие ложно, может быть выполнена другая ветка.

Синтаксис:

```
if (условие) {
    инструкция1
} else {
    инструкция2
}
```

Условие – выражение, которое должно быть истинным (true) или ложным (false). В Kotlin условие обязано иметь тип Boolean (нельзя использовать числа или объекты).

Инструкция1 – выполняется, если условие истинно.

Инструкция2 – выполняется, если условие ложно.

Для группировки нескольких инструкций используются фигурные скобки {...}.

Вложенные условия и else if

Несколько условий можно записать с помощью else if:

```
if (условие1) {
    инструкция1
} else if (условие2) {
    инструкция2
} else {
    инструкция3
}
```

В Kotlin также существует when – мощная альтернатива switch, которая может использоваться как выражение:

```
when (x) {
    1 -> println("Один")
    2 -> println("Два")
    else -> println("Другое")
}
```

Примеры

```
val cipherChar = 'a'
val fromChar = 'a'
val toChar = 'b'
var result = ""
if (cipherChar == fromChar) {
    result += toChar
} else {
    result += cipherChar
}
```

```
val x = 10
if (x > 5) {
    // ничего не делаем
} else if (x > 50) {
    println("Больше 50")
} else {
    println("Меньше или равно 5")
}
```

Циклы

Цикл for

Цикл for в Kotlin используется для перебора диапазонов, коллекций, массивов и всего, что предоставляет итератор. Синтаксис:

```
for (элемент in коллекция) {
    // тело цикла
}
```

Пример перебора диапазона:

```
for (i in 0..2) {
    println(i) // выведет 0, 1, 2
}
```

Для исключения верхней границы используется until:

```
for (i in 0 until 3) {
    println(i) // 0, 1, 2
}
```

Шаг задаётся с помощью step:

```
for (i in 0..6 step 2) {
    println(i) // 0, 2, 4, 6
}
```

Цикл do..while

Цикл do..while выполняет тело цикла, а затем проверяет условие. Тело выполняется хотя бы один раз.

```
do {
    // тело цикла
} while (условие)
```

Пример:

```
var i = 0
do {
    i += 1
    println(i)
} while (i < 5)
```

Цикл while

Цикл while проверяет условие до выполнения тела. Если условие ложно с самого начала, тело не выполнится ни разу.

```
while (условие) {
    // тело цикла
}
```

Пример:

```
var n = 0
var x = 0
while (n < 3) {
    n++
    x += n
}
```

Функции

Функции в Kotlin −ключевая концепция. Они являются объектами первого класса (можно передавать как аргументы, возвращать из других функций, сохранять в переменные). Kotlin поддерживает функциональный стиль: функции высшего порядка, лямбда-выражения, анонимные функции.

Объявление функций

Функция объявляется с помощью ключевого слова fun. Указываются имя, список параметров (с типами), возвращаемый тип (если он есть) и тело.

```
fun square(number: Int): Int {
    return number * number
}
```

Тип возвращаемого значения часто выводится автоматически:

```
fun square(number: Int) = number * number
```

Параметры передаются по значению. Изменение параметра внутри функции не влияет на аргумент вне её.

Функции как выражения (лямбды)

Kotlin позволяет создавать анонимные функции и сохранять их в переменные:

```
val square = { number: Int -> number * number }
val x = square(4) // x = 16
```

Или с явным указанием типа:

```
val square: (Int) -> Int = { it * it }
```

Вызов функций

Функция вызывается по имени с передачей аргументов:

```
square(5) // возвращает 25
```

Аргументом может быть любой объект: числа, строки, другие функции и т.д.

Область видимости (Scope)

Переменные, объявленные внутри функции, не видны за её пределами. Функции могут обращаться к переменным, объявленным во внешней области (замыкания).

```
val num1 = 20
val num2 = 3
val name = "Chamahk"

fun multiply(): Int = num1 * num2

fun getScore(): String {
    val num1 = 2
    val num2 = 3
    fun add(): String = "$name scored ${num1 + num2}"
    return add()
}

println(multiply())   // 60
println(getScore())   // Chamahk scored 5
```

Строки

В Kotlin строки представлены типом String. Они неизменяемы. Для работы со строками доступно множество методов.

Символы для экранирования:

Символ	**Значение**
\n	новая строка
\t	табуляция
\'	одинарная кавычка
\"	двойная кавычка
\\	обратный слеш

Длина строки

```
val str = "Привет"
println(str.length) // 6
```

Поиск подстроки

```
val text = "Hello, world!"
println(text.indexOf("world")) // 7
println(text.contains("Hello")) // true
```

Получение подстроки

```
val text = "Kotlin programming"
println(text.substring(0, 6)) // "Kotlin"
println(text.substring(7))    // "programming"
```

Сравнение строк

Для сравнения содержимого используется оператор == (в отличие от Java, где == сравнивает ссылки).

Для сравнения с учётом регистра или без – методы equals с параметрами.

Массивы и коллекции. Операции с ними

В Kotlin массивы представлены классом Array. Однако чаще используются списки (List, MutableList), которые более гибкие и похожи на массивы JavaScript.

Создание массива

```
val fruits = arrayOf("Яблоко", "Банан")
println(fruits.size) // 2
```

Создание изменяемого списка (динамический массив)

```
val fruits = mutableListOf("Яблоко", "Банан")
println(fruits.size) // 2
```

Доступ к элементу по индексу

```
val last = fruits[fruits.size - 1] // "Банан"
```

Итерирование по массиву/списку

```
fruits.forEachIndexed { index, item ->
    println("$item $index")
}
// Яблоко 0
// Банан 1
```

Основные методы работы с коллекциями

Категория	Методы (примеры)
Добавление/удаление	add, remove, removeAt, removeFirst, removeLast
Поиск	indexOf, contains, find, filter
Преобразование	map, filter, sorted, toList, toSet
Проверка типа	list is List<*> (или is Array<*>)

Практическая часть

Вариант 1

Тимми и Сара думают, что они влюблены. Если у одного цветка четное количество лепестков, а у другого нечётное – это означает, что они влюблены.

Напишите функцию isLove(flower1: Int, flower2: Int): Boolean, которая принимает количество лепестков каждого цветка и возвращает true, если они влюблены, и false в противном случае.

```
fun isLove(flower1: Int, flower2: Int): Boolean =
    (flower1 % 2) != (flower2 % 2)

// Тесты
fun main() {
    println(isLove(1, 4)) // true
    println(isLove(2, 2)) // false
    println(isLove(0, 1)) // true
    println(isLove(0, 0)) // false
}
```

Вариант 2

Дан список овец, где true означает «присутствует», false – «отсутствует». Напишите функцию, которая подсчитывает количество присутствующих овец.

```
fun countSheeps(sheep: List<Boolean>): Int = sheep.count { it }

// Тесты
fun main() {
    val sheepList = listOf(
        true, true, true, false, true, true, true, true, true, false,
        true, false, true, false, false, true, true, true, true, true,
        false, false, true, true
    )
    println(countSheeps(sheepList)) // 17
}
```

Вариант 3

Дан список целых чисел. Напишите функцию для поиска наименьшего целого числа в списке.

```kotlin
fun findSmallestInt(numbers: List<Int>): Int = numbers.minOrNull() ?: throw IllegalArgumentException("Empty list")

// Тесты
fun main() {
    println(findSmallestInt(listOf(78, 56, 232, 12, 8)))  // 8
    println(findSmallestInt(listOf(78, 56, 232, 12, 18))) // 12
}
```

Вариант 4

Дан список целых чисел. Реализуйте функцию, которая проверяет, являются ли какие-либо из чисел кодами символов для гласных в нижнем регистре (a, e, i, o, u). Если это так, замените число на строку этой гласной. Верните результирующий список (элементы могут быть Int или String).

```kotlin
fun isVow(array: List<Int>): List<Any> {
    val vowels = mapOf(
        'a'.code to "a",
        'e'.code to "e",
        'i'.code to "i",
        'o'.code to "o",
        'u'.code to "u"
    )
    return array.map { vowels[it] ?: it }
}

// Тесты
fun main() {
    println(isVow(listOf(118,117,120,121,117,98,122,97,120,106,104,116,113,114,113,120,106)))
    // [118, "u", 120, 121, "u", 98, 122, "a", 120, 106, 104, 116, 113, 114, 113, 120, 106]

    println(isVow(listOf(101,121,110,113,113,103,121,121,101,107,103)))
    // ["e", 121, 110, 113, 113, 103, 121, 121, "e", 107, 103]
}
```

Вариант 5

Напишите функцию для разделения строки и преобразования её в массив (список) слов.

```kotlin
fun stringToArray(str: String): List<String> = str.split(" ")

// Тесты
fun main() {
    println(stringToArray("Robin Singh"))        // [Robin, Singh]
    println(stringToArray("I love arrays they are my favorite"))
    // [I, love, arrays, they, are, my, favorite]
}
```

Вариант 6

Реализуйте функцию, которая принимает год и возвращает столетие. Первое столетие – с 1 по 100 год включительно, второе – с 101 по 200 и т.д.

```
fun century(year: Int): Int = (year - 1) / 100 + 1

// Тесты
fun main() {
    println(century(1705)) // 18
    println(century(1900)) // 19
    println(century(1601)) // 17
    println(century(2000)) // 20
    println(century(89))   // 1
}
```

Вариант 7

Реализуйте функцию, которая принимает список и возвращает его в обратном порядке.

```
fun <T> reverseList(list: List<T>): List<T> = list.reversed()

// Тесты
fun main() {
    println(reverseList(listOf(1, 2, 3, 4)))   // [4, 3, 2, 1]
    println(reverseList(listOf(3, 1, 5, 4)))   // [4, 5, 1, 3]
}
```

Вариант 8

Напишите функцию repeatStr, которая повторяет заданную строку ровно n раз.

```
fun repeatStr(n: Int, str: String): String = str.repeat(n)

// Тесты
fun main() {
    println(repeatStr(3, "*"))    // "***"
    println(repeatStr(5, "#"))    // "####"
    println(repeatStr(2, "ha "))  // "ha ha "
}
```

Вариант 9

Напишите функцию removeExclamationMarks, которая удаляет все восклицательные знаки из строки.

```
fun removeExclamationMarks(str: String): String = str.replace("!", "")

// Тесты
fun main() {
    println(removeExclamationMarks("Hello World!")) // "Hello World"
}
```

Вариант 10

Реализуйте функцию doubleChar(str: String): String, которая возвращает новую строку, где каждый символ (с учётом регистра) повторяется дважды.

```
fun doubleChar(str: String): String = str.flatMap { listOf(it, it) }.joinToString("")

// Тесты
fun main() {
    println(doubleChar("abcd"))      // "aabbccdd"
    println(doubleChar("Adidas"))    // "AAddtiidaass"
    println(doubleChar("1337"))      // "11333377"
    println(doubleChar("illuminati")) // "iillllluumminnaatti"
}
```

Вариант 11

Реализуйте функцию digitize(n: Int): List<Int>, которая принимает неотрицательное число и возвращает список его цифр в обратном порядке.

```
fun digitize(n: Int): List<Int> = n.toString().map { it.digitToInt() }.reversed()

// Тесты
fun main() {
    println(digitize(35231)) // [1, 3, 2, 5, 3]
}
```

Вариант 12

Реализуйте функцию positiveSum(arr: List<Int>): Int, которая возвращает сумму всех положительных чисел.

```
fun positiveSum(arr: List<Int>): Int = arr.filter { it > 0 }.sum()

// Тесты
fun main() {
    println(positiveSum(listOf(1, 2, 3, 4, 5)))         // 15
    println(positiveSum(listOf(1, -2, 3, 4, 5)))        // 13
    println(positiveSum(emptyList()))                   // 0
    println(positiveSum(listOf(-1, -2, -3, -4, -5)))    // 0
    println(positiveSum(listOf(-1, 2, 3, 4, -5)))       // 9
}
```

Вариант 13

Вы были в походе со своими друзьями далеко от дома. Чтобы добраться обратно, вам нужно топливо. Напишите функцию zeroFuel(distance: Int, mpg: Int, fuelLeft: Int): Boolean, которая возвращает true, если вы можете доехать, и false в противном случае.

```
fun zeroFuel(distance: Int, mpg: Int, fuelLeft: Int): Boolean = distance <= mpg * fuelLeft

// Тесты
fun main() {
    println(zeroFuel(50, 25, 2))  // true
    println(zeroFuel(100, 50, 1)) // false
}
```

Вариант 14

Реализуйте функцию take(arr: List<T>, n: Int): List<T>, которая возвращает первые n элементов списка.

```
fun <T> take(arr: List<T>, n: Int): List<T> = arr.take(n)

// Тесты
fun main() {
    println(take(listOf(0, 1, 2, 3, 5, 8, 13), 3)) // [0, 1, 2]
}
```

Вариант 15

Реализуйте функцию, которая удаляет дубликаты из списка чисел и возвращает новый список уникальных значений, сохраняя порядок.

```
fun distinct(numbers: List<Int>): List<Int> = numbers.distinct()

// Тесты
fun main() {
    println(distinct(listOf(1)))          // [1]
    println(distinct(listOf(1, 2)))       // [1, 2]
    println(distinct(listOf(1, 1, 2)))    // [1, 2]
}
```

Вариант 16

Реализуйте функцию, которая преобразует входную строку в верхний регистр.

```
fun makeUpperCase(str: String): String = str.uppercase()

// Тесты
fun main() {
    println(makeUpperCase("hello")) // "HELLO"
}
```

Вариант 17

Реализуйте функцию multipleOfIndex(arr: List<Int>): List<Int>, которая возвращает новый список, состоящий из элементов, кратных их собственному индексу во входном списке (длина > 1).

```
fun multipleOfIndex(arr: List<Int>): List<Int> =
    arr.filterIndexed { index, value -> index != 0 && value % index == 0 }

// Тесты
fun main() {
    println(multipleOfIndex(listOf(22, -6, 32, 82, 9, 25)))        // [-6, 32, 25]
    println(multipleOfIndex(listOf(68, -1, 1, -7, 10, 10)))        // [-1, 10]
    println(multipleOfIndex(listOf(11, -11)))                      // [-11]
    println(multipleOfIndex(listOf(1, -49, -1, 67, 8, -60, 39, 35))) // [-49, 8, -60, 35]
}
```

Лабораторная работа 2. Библиотека Jetpack Compose

Цель работы: изучить основы библиотеки Jetpack Compose для разработки пользовательских интерфейсов на Android.

Теоретическая часть

Jetpack Compose − это современный инструментарий для создания UI на Android, использующий язык Kotlin и декларативную парадигму. Вместо описания разметки в XML, интерфейс строится с помощью Composable-функций−обычных функций, аннотированных @Composable.

Composable-функции

Функция, помеченная аннотацией @Composable, может вызывать другие такие же функции и описывать элементы интерфейса.

```
@Composable
fun Greeting(name: String) {
    Text(text = "Привет, $name!")
}
```

Управление состоянием (State)

Состояние UI описывается с помощью mutableStateOf и remember. При изменении значения Compose автоматически перерисовывает зависимые части.

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Button(onClick = { count++ }) {
        Text("Нажато $count раз")
    }
}
```

Remember − сохраняет значение между рекомпозициями.

mutableStateOf − создаёт наблюдаемое состояние.

by (делегирование) позволяет использовать переменную как обычный тип.

Для сохранения состояния при повороте экрана используется rememberSaveable или ViewModel.

Пример простого экрана

```
@Composable
fun SimpleScreen() {
    var name by remember { mutableStateOf("") }
    var greeting by remember { mutableStateOf("") }

    Column(
        modifier = Modifier.padding(16.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        TextField(
            value = name,
            onValueChange = { name = it },
            label = { Text("Ваше имя") }
        )
        Button(onClick = { greeting = "Привет, $name!" }) {
            Text("Показать приветствие")
        }
        Text(text = greeting)
    }
```

Практическая часть

Выберите один из вариантов в соответствии с вашим уровнем подготовки. Решение должно быть оформлено в виде одного или нескольких. kt файлов (например, MainActivity.kt).

Все результаты должны быть видны на экране эмулятора или устройства.

Вариант 1

Создать экран с полем ввода имени, кнопкой «Отправить» и текстовым полем для вывода приветствия. При нажатии на кнопку введённое имя должно отображаться в формате «Привет, [имя]!». Также добавить кнопку «Очистить», которая сбрасывает поле ввода и убирает приветствие.

Вариант 2

Реализовать экран «Лайк-счётчик». На экране отображается число (количество лайков), кнопка «Лайк» и кнопка «Сбросить». При каждом нажатии «Лайк» число увеличивается на 1. При нажатии «Сбросить» число обнуляется. Дополнительно: когда количество лайков достигает 10, под счётчиком появляется текст «Отличный результат!» (при сбросе или уменьшении текст должен исчезать, если лайков меньше 10).

Вариант 3

Создать экран с отображением таймера (в секундах, начальное значение 10), кнопкой «Старт», «Пауза» и «Сброс».

Вариант 4

Реализовать экран, который позволяет переключать цветовую тему приложения между светлой и тёмной.

Вариант 5

Создать экран с двумя полями ввода чисел (целых) и кнопкой «Сумма». При нажатии на кнопку под ней отображается результат сложения.

Вариант 6

Реализовать приложение с двумя экранами (без использования библиотеки Navigation, просто переключение через состояние).

Вариант 7

Создать экран с тремя кнопками с названиями цветов: «Красный», «Зелёный», «Синий». При нажатии на кнопку фоновый цвет всего экрана меняется на соответствующий.

Вариант 8

Реализовать простой список покупок. Ввод «Молоко» → добавить → в списке: «Молоко [X]». Нажать [X] → элемент удалён.

Вариант 9

Экран содержит кнопку «Показать/Скрыть» и блок с произвольным текстом (например, «Это секретное сообщение»).

Вариант 10

Создать счётчик, который увеличивается или уменьшается на заданное значение (шаг). Шаг = 3, счётчик = 4 → нажатие «+» → 7; «-» → 4 → 1 → 0 (дальше не уменьшается).

Вариант 11

Реализовать анимированный прогресс-бар, который заполняется от 0% до 100% за 5 секунд после нажатия кнопки «Старт».

Вариант 12

Создать собственный компонент переключателя (Toggle), который имитирует Switch, но без использования стандартного Switch.

Вариант 13

Реализовать интерфейс с двумя вкладками: «Первый экран» и «Второй экран».

Вариант 14

Создать список из 5-7 предопределённых элементов (например, названия фруктов). Каждый элемент имеет Checkbox и название. Внизу экрана – кнопка «Удалить выбранные».

Вариант 15

Экран регистрации: поля Email, Пароль, Подтверждение пароля, кнопка «Зарегистрироваться».

Вариант 16

Реализовать диапазонный слайдер для выбора ценового диапазона (мин и макс). Слайдер: 200–800. В поле min ввели 500 → слайдер станет 500-800. Если ввели 900 → макс автоматически станет 900, а мин не больше макс.

Лабораторная работа 3. Модификаторы и визуальный интерфейс

Цель работы: научиться применять модификаторы для управления внешним видом и поведением UI-компонентов, а также обрабатывать пользовательские взаимодействия (клики, жесты).

Теоретическая часть

Модификатор (Modifier) – это объект, который позволяет изменять внешний вид, расположение и поведение composable-элемента. Модификаторы применяются цепочкой:

```
Modifier
.size(100.dp)                // размер
.background(Color.Blue)      // цвет фона
.padding(16.dp)              // внутренний отступ
.offset(x = 10.dp, y = 20.dp) // смещение
.clickable { /* действие */ } // обработка нажатия
.verticalScroll(rememberScrollState()) // прокрутка
```

Основные группы модификаторов:

Категория	Примеры методов
Размеры	size(), width(), height(), fillMaxSize(), fillMaxWidth(), aspectRatio()
Цвет	background(), border()
Отступы и смещение	padding(), offset()
Прокрутка	verticalScroll(), horizontalScroll(), scrollable()
Взаимодействие	clickable(), combinedClickable(), pointerInput()
Расположение	align(), weight() (в Row/Column)
Видимость/эффекты	alpha(), blur(), clip()

Практическая часть

Вариант 1

Создать карточку, реализованную с помощью компонента Box или Card. Карточка должна иметь фиксированный размер 250 на 200 плотонезависимых пикселей. Фон должен быть градиентным, например, линейным градиентом от красного к синему. Углы карточки следует скруглить на 16 dp, а также добавить тень с поднятием (elevation) 8 dp. Внутри карточки нужно установить отступы по 16 dp со всех сторон. По центру карточки размещается текст «Нажми меня». При нажатии на карточку текст должен меняться на «Нажато!», а через одну секунду возвращаться обратно. Для задержки использовать LaunchedEffect или корутину с

delay. Необходимо применить модификаторы background с кистью Brush.linearGradient, clickable, padding и size.

Вариант 2

Реализовать кнопку (можно использовать Box или Button). В исходном состоянии она имеет размер 200 на 80 dp. При нажатии на кнопку её размер должен анимированно уменьшиться до 180 на 70 dp. Когда пользователь отпускает кнопку, размер должен вернуться к исходному. Фон кнопки – зелёный, текст – белый. Для обработки нажатия и отпускания использовать combinedClickable или pointerInput с detectTapGestures. Анимацию размера реализовать с помощью animateDpAsState. Также примените модификатор clip с RoundedCornerShape(12.dp) для скругления углов.

Вариант 3

Создать горизонтальный список, состоящий из десяти цветных прямоугольников. Ширина каждого прямоугольника – 120 dp, высота – 80 dp. Внутри прямоугольника должен быть текст с его порядковым номером от 1 до 10. Весь список должен прокручиваться горизонтально – использовать Row с модификатором horizontalScroll. Фон каждого прямоугольника может быть случайным или циклически меняющимся (красный, синий, зелёный, жёлтый). При нажатии на прямоугольник он должен выделяться рамкой: border толщиной 3 dp золотого цвета. Если нажать на выделенный прямоугольник повторно, рамка должна сниматься. Состояние выделения хранится в remember для каждого элемента.

Вариант 4

На экране, свёрстанном с помощью Box, разместить плавающую кнопку в правом нижнем углу. Кнопка должна иметь размер 56 на 56 dp, круглую форму (достигается с помощью clip), фон акцентного цвета и иконку (или текст «+»). Отступ от краёв экрана задайте в 16 dp с помощью модификатора padding. При нажатии на кнопку она должна сместиться вверх на 20 dp (используйте модификатор offset с анимацией через animateDpAsState), затем через 300 миллисекунд вернуться обратно. Во время смещения цвет фона кнопки меняется на оранжевый. Для расположения кнопки используйте Modifier.align(Alignment.BottomEnd).

Вариант 5

Создать вертикальный список (LazyColumn) из двадцати элементов. Каждый элемент представляет собой строку текста «Элемент N». Чётные элементы должны иметь светло-серый фон, нечётные – белый. Между элементами задать нижний отступ в 4 dp. При нажатии на элемент его фон временно (на 200 мс) меняется на жёлтый, после чего возвращается к исходному цвету. Список должен занимать всю высоту экрана (fillMaxSize) и обеспечивать прокрутку. Применить модификаторы background, clickable, padding и fillMaxSize.

Вариант 6

Реализовать горизонтальную полосу прогресса высотой 20 dp и шириной 300 dp. Заливка полосы должна быть градиентной – горизонтальный градиент от зелёного к жёлтому. На экране также должны быть две кнопки: «Запустить» и

«Сброс». При нажатии на «Запустить» прогресс анимированно (за 3 секунды) заполняется от 0% до 100%. При нажатии на «Сброс» прогресс мгновенно возвращается к 0% (без анимации). Текущий процент заполнения должен отображаться текстом. Для анимации прогресса использовать Modifier.fillMaxWidth(fraction) с animateFloatAsState, а для градиента – Modifier.background(brush = Brush.horizontalGradient(...)).

Вариант 7

На экране отобразить круг диаметром 100 dp. При обычном нажатии круг должен увеличиться до 120 dp, при повторном нажатии – вернуться к 100 dp. При долгом нажатии (удержание более 500 мс) цвет круга должен измениться на случайный, а размер остаться прежним. Для обработки разных типов нажатий использовать combinedClickable с параметрами onClick и onLongClick. Применить модификаторы size, clip с CircleShape и background.

Вариант 8

Создать квадрат синего цвета размером 100 на 100 dp, который можно перетаскивать по экрану в любом направлении. Перетаскивание должно быть ограничено видимой областью (то есть квадрат не должен выходить за границы экрана). Когда пользователь отпускает квадрат, он должен плавно вернуться в исходное положение с координатами (0, 0). Под квадратом отобразите текст с текущими координатами смещения. Для реализации жеста использовать pointerInput с detectDragGestures. Изменение координат выполнить через Modifier.offset { IntOffset(x, y) }.

Вариант 9

Спроектировать карточку (Card) с заголовком. При нажатии на заголовок под ним должно появляться дополнительное содержимое (например, какой-либо текст), которое изначально скрыто. Повторное нажатие на заголовок скрывает это содержимое. Появление и исчезновение должны сопровождаться анимацией изменения высоты – для этого использовать модификатор animateContentSize. Карточка должна иметь отступы, тень и скруглённые углы. Состояние видимости дополнительного содержимого хранить в remember.

Вариант 10

Экран должен состоять из трёх частей. Верхняя панель высотой 60 dp, серого цвета, содержит текст «Закреплённая панель» и не прокручивается. Ниже располагается прокручиваемая область – Column с модификатором verticalScroll, в которой находятся 20 элементов (каждый элемент – строка текста и иконка). В самом низу экрана всегда видна кнопка «Действие» (она не участвует в прокрутке). При нажатии на эту кнопку необходимо вывести в консоль (или показать тост) количество пикселей, на которое была прокручена область. Значение прокрутки можно получить из ScrollState, переданного в verticalScroll.

Вариант 11

Кнопка с рамкой и изменением цвета при наведении. Создать кнопку, имеющую рамку (border) шириной 2 dp синего цвета. Внутренние отступы кнопки должны составлять 12 dp. При наведении курсора на кнопку (использовать модификатор hoverable) рамка должна становиться красной и увеличиваться до 4

dp. При нажатии на кнопку фон кнопки должен меняться на светло-зелёный. Применить модификаторы border, hoverable и clickable.

Вариант 12

Необходимо создать сетку размером 3 на 3 квадрата. Каждый квадрат имеет размер 80 на 80 dp, между квадратами заданы отступы по 8 dp. При нажатии на любой квадрат происходит два действия: во-первых, его размер анимировано увеличивается до 90 на 90 dp, а затем возвращается обратно; во-вторых, цвет квадрата меняется на случайный. Остальные квадраты при этом не изменяются. Состояние размера и цвета для каждого квадрата храните индивидуально с помощью remember. Для расположения квадратов можно использовать комбинацию Column и Row либо LazyVerticalGrid. Анимацию размера реализовать через animateDpAsState.

Вариант 13

Вращающееся изображение при нажатии. На экране отображается картинка или иконка размером 150 на 150 dp. При каждом нажатии на неё она должна поворачиваться на 90 градусов (по часовой стрелке) с анимацией. Таким образом, после четырёх нажатий изображение совершает полный оборот в 360 градусов. Для поворота использовать модификатор graphicsLayer с параметром rotationZ, а анимацию – animateFloatAsState. Состояние текущего угла поворота хранить в mutableStateOf.

Вариант 14

Реализовать горизонтальную полосу высотой 20 dp и шириной 300 dp. Левая часть полосы залита красным цветом, правая – серым. На полосе находится бегунок в виде круга диаметром 30 dp. Бегунок можно перетаскивать горизонтально в пределах полосы. Положение бегунка определяет долю красного цвета: чем правее бегунок, тем больше красной части. Процент красного (от 0 до 100) должен отображаться текстом. Для перетаскивания использовать detectDragGestures, для перемещения бегунка – Modifier.offset. Полосу можно нарисовать с помощью Canvas или через два Box с весами.

Вариант 15

Создать вертикальный список (LazyColumn) из десяти элементов, каждый из которых содержит текст «Задача N». Реализовать жест свайпа влево для удаления элемента. При выполнении свайпа элемент должен плавно смещаться влево (анимация смещения) и затем исчезать из списка. Для обнаружения свайпа использовать Modifier.swipeable или pointerInput с detectHorizontalDragGestures. Добавить на экран кнопку «Сбросить список», которая восстанавливает все удалённые элементы в исходном порядке. Список должен храниться в mutableStateListOf.

Лабораторная работа 4. Контейнеры компоновки

Цель работы: научиться использовать основные контейнеры компоновки Box, Column и Row для построения пользовательских интерфейсов различной сложности, комбинировать их между собой, управлять выравниванием, распределением пространства и вложенностью элементов.

Теоретическая часть

В Jetpack Compose существует три основных контейнера (layout-компонента), которые служат строительными блоками для любого интерфейса.

Column – вертикальный контейнер, располагает свои дочерние элементы один под другим. Параметры verticalArrangement управляют распределением места по вертикали (например, Top, Center, Bottom, SpaceBetween), a horizontalAlignment – выравниванием по горизонтали (Start, CenterHorizontally, End).

Row – горизонтальный контейнер, располагает дочерние элементы в ряд. Аналогично имеет horizontalArrangement и verticalAlignment.

Box – контейнер, который накладывает дочерние элементы друг на друга (похож на FrameLayout в Android View). Позволяет позиционировать элементы внутри себя с помощью Modifier.align(). Также может растягивать содержимое.

Комбинация этих контейнеров позволяет строить интерфейсы любой сложности – от простых карточек до сложных экранов с несколькими зонами.

Основные модификаторы

Modifier.weight(weight) − в Row или Column задаёт, какую долю свободного пространства займёт элемент.

Modifier.align(alignment) − внутри Box задаёт положение элемента. В Column/Row управляет выравниванием отдельного элемента.

Modifier.fillMaxSize(), fillMaxWidth(), fillMaxHeight() − растягивает элемент на всё доступное пространство контейнера.

Практическая часть

Вариант 1. Профильная карточка пользователя

Создайть карточку профиля пользователя, используя Column для вертикального расположения всех элементов. В верхней части карточки разместить круглый аватар (размер 80×80 dp) с помощью Box – внутри Box должен быть сам аватар, а поверх него в правом нижнем углу – маленький зелёный кружок (статус «онлайн»). Для позиционирования статуса используйте Modifier.align(Alignment.BottomEnd). Ниже аватара в Column разместить имя пользователя (шрифт 20 sp), город (шрифт 14 sp) и короткое описание (шрифт 12

sp). Все текстовые элементы должны быть выровнены по центру горизонтали. Добавьте отступы внутри карточки 16 dp. Саму карточку обернуть в Box, чтобы добавить тень и скругление углов 16 dp.

Вариант 2. Горизонтальное меню с индикатором выбора

Реализовать горизонтальное меню из трёх пунктов: «Главная», «Профиль», «Настройки». Использовать Row, которая занимает всю ширину экрана. Каждый пункт меню представляет собой Column с иконкой (можно использовать текстовые символы) и текстом под ней. Распределить пункты равномерно по ширине с помощью Modifier.weight(1f). При нажатии на пункт меню его цвет меняется на акцентный (например, синий), а под текстом появляется маленькая полоска-индикатор (высота 2 dp, ширина 20 dp, цвет акцентный). Использовать Box для наложения индикатора под текст. Активный пункт должен быть только один. Добавить отступы между пунктами 8 dp.

Вариант 3. Многострочный список новостей с заголовком и кнопкой

Создайть вертикальный список новостей (статический, из 3-4 элементов) с помощью Column. Каждый элемент новости – это Row, содержащая слева миниатюру (прямоугольник 40×40 dp серого цвета), а справа – Column с заголовком новости (шрифт 16 sp) и датой (шрифт 12 sp, серый цвет). Между элементами добавить разделитель (Divider) высотой 1 dp. Над списком разместите Row с заголовком «Последние новости» (выравнивание по левому краю) и кнопкой «Всё» (выравнивание по правому краю). Внизу экрана добавьте кнопку «Загрузить ещё», центрированную по горизонтали с помощью Box, занимающую всю ширину.

Вариант 4. Нижняя панель навигации

Создайть нижнюю панель навигации с помощью Row, расположенную внизу экрана. Панель должна содержать три элемента: «Дом», «Поиск», «Аккаунт». Каждый элемент – это Column с иконкой (или символом) и текстом. Панель должна иметь высоту 70 dp, фоновый цвет, тень и выравнивание элементов по центру. При нажатии на элемент его иконка и текст меняют цвет на акцентный, остальные становятся серыми. Активный элемент только один. Всю панель завернуть в Box, который прижимает её к низу экрана (Modifier.align(Alignment.BottomCenter)). Основной контент над панелью может быть любым (например, просто текст).

Вариант 5. Кастомный переключатель

Реализовать переключатель (аналог Switch) с помощью Box. Создать прямоугольник размером 60×30 dp с серым фоном и скруглёнными углами 30 dp. Внутри этого Box поместить круг диаметром 26 dp белого цвета. Круг должен смещаться влево или вправо в зависимости от состояния (включён/выключен). При нажатии на переключатель круг анимированно перемещается на противоположную сторону, а фон плавно меняется с серого на зелёный (или наоборот). Для анимации используйте animateDpAsState. Разместите на экране два таких переключателя в Column – «Уведомления» и «Звук». Под каждым переключателем подпись.

Вариант 6. Карточка с кнопкой закрытия в углу

Создайте карточку (диалоговое окно) с помощью Box, который занимает размер по содержимому. Внутри Box разместите Column с заголовком «Внимание!», текстом «Вы уверены?» и двумя кнопками («Да», «Нет») в Row.

Column должна быть центрирована внутри Box по горизонтали и вертикали (используйте Modifier.align(Alignment.Center)). В правом верхнем углу Box поместить маленькую кнопку «✕» с помощью Modifier.align(Alignment.TopEnd). При нажатии на крестик карточка исчезает (меняйте состояние видимости). Карточка должна иметь фон, скруглённые углы 16 dp и тень. Разместить карточку поверх полупрозрачного фона, закрывающего весь экран.

Вариант 7. Строка поиска с иконками слева и справа

Реализовать строку поиска с помощью Row. Внутри Row слева разместите иконку лупы, затем TextField (поле ввода), который должен занимать всё оставшееся пространство – использовать Modifier.weight(1f). Справа добавить иконку микрофона (или фильтра). Row должна иметь фон, скруглённые углы 24 dp, внутренние отступы 8 dp и тень. Под строкой поиска в Column отобразить список недавних запросов (3-4 строки). Каждый недавний запрос – это Row с текстом запроса (занимает всё место, weight(1f)) и иконкой удаления «✕» справа. При нажатии на иконку удаления запрос удаляется из списка.

Вариант 8. Шахматная доска

Создать шахматную доску размером 4×4 клетки (для простоты) или 8×8 (для усложнения). Использовать внешний Column для строк. Каждая строка – это Row для клеток. Каждая клетка – это Box размером 40×40 dp. Цвет клеток должен чередоваться (например, чёрный и белый). При нажатии на любую клетку она выделяется золотой рамкой (border толщиной 2 dp). Если нажать на выделенную клетку повторно, рамка снимается. Для управления состоянием выделения использовать список (матрицу) Boolean в remember.

Вариант 9. Вертикальный слайдер громкости

Создать вертикальный слайдер громкости с помощью Column, которая занимает высоту 200 dp. Внутри Column разместите: сверху иконку динамика, по центру – вертикальный Slider (ползунок), снизу – текст с текущим процентом громкости. Слайдер должен занимать всё доступное пространство между иконкой и текстом – использовать Modifier.weight(1f) для контейнера слайдера или для самого слайдера. Слайдер можно сделать вертикальным, обернув его в Box с Modifier.rotate(-90f), либо используя готовый вертикальный слайдер. Всю Column разместите в центре экрана с помощью Box.

Вариант 10. Панель инструментов с группами кнопок

Создать панель инструментов с помощью Column. Верхняя часть панели – Row с заголовком «Редактор» по центру (занимает weight(1f)) и иконкой меню (три точки) справа. Нижняя часть – Row с тремя группами кнопок форматирования: группа «Стиль» (жирный, курсив), группа «Выравнивание» (влево, центр, вправо) и группа «Цвет» (красный, синий). Каждая группа – это Row из двух-трёх маленьких кнопок (можно использовать TextButton). Между группами добавить вертикальные разделители (Divider). При нажатии на кнопки они меняют свой цвет. Панель должна иметь фоновый цвет и тень.

Вариант 11. Список товаров с горизонтальной прокруткой

Создать экран с вертикальным списком категорий (Column). В каждой категории – горизонтальный список товаров с прокруткой (Row с

Modifier.horizontalScroll). Категория содержит заголовок и под ним Row из 4-5 карточек товаров. Каждая карточка товара – это Column шириной 120 dp с прямоугольником-изображением (80×80 dp), названием товара и ценой. Между карточками добавить отступ. При нажатии на карточку выводите её название в консоль. Категории должны отделяться друг от друга отступом.

Вариант 12. Чат-пузыри

Реализовать экран чата с сообщениями. Использовать Column с прокруткой (verticalScroll) для списка сообщений. Каждое сообщение – это Row. В зависимости от того, отправлено сообщение пользователем или получено, расположение меняется:

Свои сообщения: аватар (круг) справа, текст справа, Row выравнивается по правому краю.

Чужие сообщения: аватар слева, текст слева, Row выравнивается по левому краю.

Текст сообщения находится в Box с фоном, скруглёнными углами и отступами. Цвет фона для своих сообщений – зелёный, для чужих – серый. Внизу экрана разместить Row с TextField и кнопкой отправки, прижатую к низу с помощью Box.

Вариант 13. Индикатор загрузки

Создать экран-заглушку загрузки. В центре экрана (использовать Box с Modifier.fillMaxSize()) разместить Column, выровненную по центру. В Column поместить круговой индикатор прогресса (CircularProgressIndicator), затем текст «Загрузка...» и ниже – линейный прогресс-бар, который анимированно заполняется от 0% до 100% за 5 секунд. Добавить кнопку «Отмена» внизу экрана, прижатую к нижнему краю с помощью Box и Alignment.BottomCenter. При нажатии на «Отмена» анимация останавливается и прогресс сбрасывается.

Вариант 14. Аккордеон (раскрывающиеся панели)

Реализовать аккордеон – вертикальный список из трёх разделов, каждый из которых раскрывается при нажатии. Использовать внешнюю Column. Каждый раздел – это Column с Modifier.animateContentSize(). Верхняя часть раздела – Row с названием раздела и иконкой стрелки (вниз, когда закрыто; вверх, когда открыто). При нажатии на эту Row под ней появляется вложенная Column с подпунктами. Одновременно может быть открыт только один раздел – при открытии другого предыдущий автоматически закрывается. Состояние открытого раздела храните в remember.

Вариант 15. Кнопка с бейджем

Создать кнопку, представляющую собой Row с иконкой и текстом («Корзина»). Кнопка имеет размеры по содержимому, фон, отступы и скруглённые углы. Обернуть эту кнопку в Box (не меняя её внешний вид). В правом верхнем углу Box разместите маленький бейдж – круг красного цвета диаметром 20 dp с белым текстом «3» внутри. Бейдж должен выступать за пределы кнопки. Для этого использовать Modifier.offset(x = 8.dp, y = (-8).dp). Бейдж отображается только если

количество товаров в корзине больше 0. Добавить кнопку «Добавить», увеличивающую счётчик, и кнопку «Убрать», уменьшающую его. Счётчик хранить в remember.

Вариант 16. Экраны с вкладками

Создать простую реализацию вкладок без использования стандартных компонентов. Верхняя часть – Row с горизонтальной прокруткой (horizontalScroll) и тремя вкладками: «Первый», «Второй», «Третий». Каждая вкладка – это Box с отступами, фоном при выборе. Ниже – Box, который отображает содержимое выбранной вкладки (используйте when). Содержимое первой вкладки – Column с текстом, второй – Row с кнопками, третьей – Box с картинкой. При нажатии на вкладку активная вкладка меняется, её стиль (фон, цвет текста) обновляется.

Вариант 17. Форма входа с группировкой полей

Создать экран входа в приложение. Используйте Column, которая занимает всю высоту экрана (fillMaxSize). В Column разместите:

Логотип (просто текст или иконку) в верхней части с отступом 32 dp.

Заголовок «Вход».

Поле ввода логина.

Поле ввода пароля.

Кнопку «Войти».

Текст «Забыли пароль?» (по центру).

Использовать Modifier.weight(1f) для создания промежутков: между логотипом и заголовком, между заголовком и полями, а также между кнопкой и текстом «Забыли пароль?». Кнопка должна быть растянута на всю ширину. Добавить валидацию: если поля пустые, кнопка неактивна.

Вариант 18. Сложный макет

Создать экран с тремя зонами: верхняя панель (шапка), основной контент (с прокруткой), нижняя панель (подвал). Использовать Column для вертикальной структуры. Шапка – Row высотой 56 dp с заголовком по центру и иконкой «Назад» слева. Подвал – Row высотой 50 dp с двумя кнопками («Отмена», «Сохранить») по краям. Основной контент – Column с verticalScroll, содержащая 15 строк текста. Между шапкой, основным контентом и подвалом нет фиксированных отступов – контент должен прокручиваться, а шапка и подвал всегда видны.

Лабораторная работа 5. Состояние компонентов

Цель работы: изучить основные механизмы управления состоянием в Jetpack Compose: remember, mutableStateOf, rememberSaveable, подъём состояния (state hoisting), производные состояния (derivedStateOf) и взаимодействие с ViewModel. Научиться создавать реактивные интерфейсы, правильно организовывать хранение и передачу данных между компонентами.

Теоретическая часть

В Jetpack Compose интерфейс обновляется автоматически при изменении состояния. Состояние (State) – это данные, которые могут изменяться во времени и вызывать перерисовку (рекомпозицию) зависимых частей UI.

Основные понятия

MutableStateOf − создаёт наблюдаемое состояние. При изменении значения все composable-функции, читающие его, перезапускаются (рекомпозиция).

Remember − сохраняет значение между рекомпозициями. Без remember состояние сбрасывалось бы при каждом перерисовывании.

RememberSaveable − аналогичен remember, но сохраняет состояние при повороте экрана и восстановлении Activity (через Bundle).

DerivedStateOf − создаёт состояние, которое вычисляется из других состояний и обновляется только тогда, когда изменяется его входные данные.

State<T> − интерфейс только для чтения. Используется для передачи состояния в composable-функции без возможности его изменения.

MutableState<T> − интерфейс для чтения и записи состояния.

Подъём состояния

Принцип, при котором состояние поднимается на уровень выше (в родительский компонент), а дочерние получают его как параметры. Это делает компоненты переиспользуемыми и без сохранения состояния.

```
// Без подъёма состояния (плохо)
@Composable
fun BadCounter() {
    var count by remember { mutableStateOf(0) }
    Button(onClick = { count++ }) { Text("$count") }
}

// С подъёмом состояния (хорошо)
@Composable
fun GoodCounter(count: Int, onIncrement: () -> Unit) {
    Button(onClick = onIncrement) { Text("$count") }
}
```

Работа с ViewModel

ViewModel хранит состояние, переживающее конфигурационные изменения (поворот экрана). Compose интегрируется с ViewModel через viewModel().

```
class MyViewModel : ViewModel() {
    private val _count = mutableStateOf(0)
    val count: State<Int> = _count

    fun increment() { _count.value++ }
}

@Composable
fun MyScreen(viewModel: MyViewModel = viewModel()) {
    val count by viewModel.count
    Button(onClick = { viewModel.increment() }) { Text("$count") }
}
```

Производные состояния

Используется, когда новое состояние зависит от другого состояния и не должно вызывать лишних рекомпозиций.

```
val list = remember { mutableStateListOf(1,2,3) }
val isEmpty by remember { derivedStateOf { list.isEmpty() } }
// isEmpty обновится только когда изменится пустота списка
```

Для простых типов используйте rememberSaveable. Для сложных – добавьте @Parcelize или используйте MapSaver/ListSaver.

Практическая часть

Вариант 1 Счётчик с шагом и ограничениями

Реализовать экран со счётчиком (целое число), который можно увеличивать и уменьшать. Начальное значение 0. Максимальное значение 10, минимальное 0. Добавьте поле ввода (TextField) для задания шага изменения (по умолчанию 1). При нажатии на кнопку «+» счётчик увеличивается на величину шага, при нажатии на «-» – уменьшается на величину шага. Шаг может быть любым целым числом от 1 до 5. Отображайте текущее значение счётчика крупным шрифтом. При достижении максимума или минимума соответствующие кнопки должны становиться неактивными (enabled = false). Состояние должно сохраняться при повороте экрана.

Вариант 2. Список задач с фильтрацией

Создать приложение «Список задач». Каждая задача имеет текст и статус (выполнена/не выполнена). Реализовать:

Поле ввода для добавления новой задачи и кнопку «Добавить».

Три кнопки фильтра: «Все», «Активные», «Выполненные».

Список задач, отображаемый согласно выбранному фильтру.

Чекбокс у каждой задачи для изменения статуса.

Кнопку «Удалить выполненные».

Состояние должно сохраняться при повороте.

Вариант 3. Таймер с управлением

Реализовать таймер, который отсчитывает секунды от заданного пользователем значения до 0. Используйте ViewModel для хранения состояния. Компоненты:

Поле ввода для установки начального времени (секунды, от 5 до 60).

Кнопки «Старт», «Пауза», «Сброс».

Отображение оставшегося времени.

При нажатии «Старт» запускается обратный отсчёт. «Пауза» останавливает отсчёт. «Сброс» останавливает таймер и устанавливает время в начальное значение. При достижении 0 таймер останавливается и выводится сообщение «Время вышло!». Состояние должно переживать поворот экрана.

Вариант 4. Форма регистрации с валидацией

Создать форму регистрации с полями: Имя (не менее 2 символов), Email (должен содержать '@' и '.'), Пароль (не менее 6 символов), Подтверждение пароля (должен совпадать с паролем). Для каждого поля под ним отображайте сообщение об ошибке, если поле не валидно. Кнопка «Зарегистрироваться» должна быть активна только тогда, когда все поля валидны. При нажатии на кнопку показывайте всплывающее сообщение (тост или текст) «Регистрация успешна!» и очищайте форму.

Вариант 5. Выбор даты с помощью кастомного селектора

Реализовать компонент выбора даты (день, месяц, год) без использования готовых диалогов. Создать три отдельные Column с числами (дни от 1 до 31, месяцы от 1 до 12, годы от 2020 до 2030), каждая с вертикальной прокруткой и выделением выбранного элемента. Реализовать подъём состояния: родительский компонент хранит выбранные день, месяц, год и предоставляет их дочерним компонентам через параметры, а также колбэки для изменения. Выбранные значения отображаются текстом под селекторами. Необходимо учитывать количество дней в месяце (февраль – 28 или 29, апрель/июнь/сентябрь/ноябрь – 30). При выборе месяца автоматически корректируйте день, если он выходит за пределы допустимого. Состояние сохраняйте при повороте.

Вариант 6. Кнопка с двойным нажатием

Реализовать кнопку (Box с текстом «Нажми дважды»), которая выполняет действие только при двух быстрых нажатиях подряд. Используйте состояние для хранения времени последнего нажатия. Если интервал между двумя нажатиями меньше 300 миллисекунд – считать двойным нажатием и выводить сообщение «Двойное нажатие!». При одиночном нажатии выводите «Одиночное нажатие». Учитывать, что после двойного нажатия не должно быть сообщения об одиночном. Состояние хранить в remember.

Вариант 7. Expandable текст (подробнее/скрыть) с анимацией

Создать компонент, который показывает короткий текст (до 100 символов) и кнопку «Подробнее». При нажатии на кнопку текст расширяется до полного (до 500 символов), и кнопка меняется на «Скрыть». Текст должен обрезаться по количеству символов в зависимости от состояния. Состояние (развёрнут/свёрнут) хранить в remember. Добавить возможность, чтобы текст был взят из переменной (статический для примера). При повороте экрана состояние должно сохраняться.

Вариант 8. Переключение между двумя экранами с сохранением состояния

Реализовать приложение с двумя «экранами», переключение между которыми происходит через кнопки «Экран 1» и «Экран 2». Использовать подъём состояния: текущий экран хранится в родительском компоненте. Каждый экран имеет своё внутреннее состояние (например, первый – счётчик, второй – список задач). При переключении между экранами их состояние должно сохраняться. Для этого состояние каждого экрана должно быть поднято на уровень родителя или храниться в remember на уровне каждого экрана, но тогда при переключении оно не будет сбрасываться, потому что composable не будет пересоздаваться. Реализовать так, чтобы состояние счётчика на первом экране не сбрасывалось при переходе на второй и обратно.

Вариант 9. Текст с изменяемым стилем

Создать экран с текстовым полем и тремя кнопками: «Жирный», «Курсив», «Подчёркнутый». Каждая кнопка переключает соответствующий стиль текста (включён/выключен). Текст должен отображаться с применёнными стилями. Состояние (три булевых флага) должно сохраняться при повороте экрана – использовать rememberSaveable. Дополнительно: отобразить предварительный просмотр текста, который меняет стиль в реальном времени.

Вариант 10. Слайдер с отображением значения и производным цветом

Реализовать горизонтальный слайдер (Slider) со значениями от 0 до 100. Отобразить текущее значение числом рядом. Цвет ползунка должен меняться в зависимости от значения: от зелёного (0-33) через жёлтый (34-66) к красному (67-100). Использовать derivedStateOf для вычисления цвета на основе значения слайдера. Дополнительно: добавить текстовое поле для ручного ввода числа, синхронизированное со слайдером (при изменении числа слайдер перемещается, и наоборот). Состояние значения храните в remember.

Вариант 11. Список выбранных элементов с счётчиком

Создать список из 10 элементов. Каждый элемент имеет чекбокс. Внизу экрана отображается количество выбранных элементов. Реализовать подъём состояния: родительский компонент хранит список с флагами выбора и предоставляет дочернему компоненту (элементу списка) состояние и колбэк для изменения. Использовать LazyColumn. При нажатии на чекбокс счётчик обновляется. Добавить кнопку «Выбрать всё» и «Снять всё». Кнопка «Выбрать всё» активна, если не все элементы выбраны, «Снять всё» – если выбрано хотя бы что-то. Использовать derivedStateOf для вычисления флагов активности кнопок. Состояние списка сохраняйте при повороте через rememberSaveable.

Вариант 12. Поисковая строка с дебаунсом

Реализовать поле поиска и список результатов. При вводе текста в поле поиска фильтрация списка должна происходить не мгновенно, а с задержкой 500 миллисекунд после последнего ввода. Использовать LaunchedEffect с key = text и delay(500L). Состояние для текста поиска, отфильтрованного списка и текущего запроса храните в remember. Отобразить количество найденных элементов. При повороте экрана состояние ввода и результатов должно сохраняться.

Вариант 13. Переключение темы приложения с сохранением

Создать экран, который позволяет переключать цветовую тему приложения между светлой и тёмной. Использоват CompositionLocalProvider. Реализовать rememberSaveable для хранения текущей темы (true – тёмная, false – светлая). При нажатии на кнопку «Сменить тему» фон экрана и цвет текста должны меняться. Для этого необходимо создать Box с Modifier.background(if (isDark) Color.Black else Color.White) и текст с соответствующим цветом. Также изменить цвет кнопки. Состояние темы должно переживать поворот экрана. Дополнительно: добавить третий вариант – «Системная тема» (определяется настройками устройства).

Вариант 14. Валидация пароля с индикатором сложности

Создать поле ввода пароля и под ним индикатор сложности. Индикатор должен показывать:

Слабый (менее 6 символов) – красный

Средний (6-8 символов, содержит буквы и цифры) – жёлтый

Сильный (более 8 символов, буквы в разных регистрах, цифры, спецсимволы) – зелёный

Использовать derivedStateOf для вычисления уровня сложности при каждом изменении пароля. Отобразить текстовое сообщение и цветной индикатор. Пароль скрыть точками. Добавить чекбокс «Показать пароль». Состояние пароля и чекбокса сохраняйте при повороте.

Вариант 15. Счётчик с возможностью сохранения моментального снимка

Реализовать счётчик с помощью ViewModel и SavedStateHandle, чтобы состояние сохранялось не только при повороте, но и при уничтожении процесса. Добавить кнопку «Сохранить снимок», которая запоминает текущее значение счётчика в отдельную переменную. Кнопка «Восстановить» – устанавливает счётчик в сохранённое значение. Кнопки «+1», «-1». Обычное значение счётчика и сохранённое – два разных состояния. Приложение должно корректно восстанавливать оба значения после полного перезапуска.

Вариант 16. Анимированный переключатель с состоянием

Создать переключатель вручную с помощью Box и анимации с использованием animateColorAsState для цвета фона и animateDpAsState для положения кружка. Состояние переключателя (вкл/выкл) хранить в remember. Добавить текстовую метку рядом (например, «Уведомления») и сделать всю строку кликабельной. При нажатии состояние переключателя меняется. Реализовать два переключателя на экране, их состояния должны быть независимыми и сохраняться при повороте.

Вариант 17. Список с возможностью редактирования названия

Создать список элементов. Каждый элемент отображается как Row с текстом, иконкой «Редактировать» и иконкой «Удалить». При нажатии на «Редактировать» текст заменяется на TextField с текущим значением, и появляются кнопки «Сохранить» и «Отмена». После сохранения текст обновляется. Использовать подъём состояния: родительский компонент хранит список и редактируемый элемент (id или индекс). Состояние редактирования (какой элемент редактируется, временное значение текста) должно быть локальным для элемента или поднято состояния. Реализовать так, чтобы можно было редактировать только один элемент одновременно. Состояние списка сохраняйте при повороте через rememberSaveable.

Вариант 18. Комплексное приложение: заметки с тегами

Разработать приложение для заметок. Каждая заметка имеет текст и список тегов (строки). Функциональность:

Поле ввода текста заметки, поле ввода тега, кнопка «Добавить тег» (теги добавляются к текущей заметке), кнопка «Сохранить заметку».

Список сохранённых заметок (отображаются текст и теги).

Фильтр по тегу: поле ввода для фильтра, отображаются только заметки, содержащие этот тег.

Использовать ViewModel для хранения списка заметок, текущей заметки (черновика) и фильтра. Для отфильтрованных заметок используйте derivedStateOf. Все операции (добавление заметки, добавление тега, изменение фильтра) – через методы ViewModel. Состояние должно полностью переживать поворот экрана. Реализовать удаление заметок (иконка корзины у каждой заметки).

Лабораторная работа 6. Визуальные компоненты

Цель работы: изучить основные визуальные компоненты Jetpack Compose: текстовые элементы, кнопки, поля ввода, чекбоксы, радиокнопки, иконки, плавающие кнопки, панели приложений, Scaffold, всплывающие сообщения, выдвижные панели, слайдеры, переключатели, диалоговые окна, выпадающие меню и индикаторы прогресса. Научиться применять их для построения современных Android-интерфейсов.

Теоретическая часть

1. Текстовые компоненты

Text – отображает текстовую строку. Позволяет задавать размер, цвет, начертание, выравнивание, ограничение по количеству строк.

TextField – поле ввода с подчёркиванием. Обязательно связывается с состоянием (value и onValueChange).

OutlinedTextField – вариант с контурной обводкой.

```
Text("Привет", fontSize = 20.sp, fontWeight = FontWeight.Bold)

var text by remember { mutableStateOf("") }
TextField(value = text, onValueChange = { text = it })
OutlinedTextField(value = text, onValueChange = { text = it }, label = { Text("Логин") })
```

2. Кнопки

Button – стандартная кнопка с текстом или иконкой.

IconButton – круглая кнопка-иконка.

FloatingActionButton (FAB) – круглая плавающая кнопка.

ExtendedFloatingActionButton – расширенная версия с текстом и иконкой.

```
Button(onClick = {}) { Text("Нажми") }
IconButton(onClick = {}) { Icon(Icons.Default.Favorite, null) }
FloatingActionButton(onClick = {}) { Icon(Icons.Default.Add, null) }
ExtendedFloatingActionButton(onClick = {}, icon = { Icon(...) }, text = { Text("Добавить") })
```

3. Компоненты выбора

Checkbox – флажок для бинарного выбора.

RadioButton – радиокнопка для выбора одного варианта из группы.

Switch – переключатель (вкл/выкл).

Slider – ползунок для выбора значения из диапазона.

```
var checked by remember { mutableStateOf(false) }
Checkbox(checked = checked, onCheckedChange = { checked = it })

var selected by remember { mutableStateOf("A") }
RadioButton(selected = selected == "A", onClick = { selected = "A" })

var switched by remember { mutableStateOf(false) }
Switch(checked = switched, onCheckedChange = { switched = it })

var value by remember { mutableStateOf(50f) }
Slider(value = value, onValueChange = { value = it })
```

4. Модификаторы toggleable и selectable

toggleable – делает любой компонент переключаемым (аналог чекбокса для строк).

selectable – делает элемент выбираемым (для списков, радиокнопок).

```
Row(modifier = Modifier.toggleable(value = checked, onValueChange = { checked = it })) { ... }
Row(modifier = Modifier.selectable(selected = isSelected, onClick = { ... })) { ... }
```

5. Контейнеры и панели

Scaffold – каркас экрана, объединяет верхнюю/нижнюю панели, FAB, контент, Snackbar.

TopAppBar – верхняя панель с заголовком, навигационной иконкой и действиями.

BottomAppBar – нижняя панель, часто содержит FAB с выемкой.

```
Scaffold(
topBar = { TopAppBar(title = { Text("Заголовок") }) },
bottomBar = { BottomAppBar { ... } },
floatingActionButton = { FAB { ... } }
) { padding -> /* контент с отступами */ }
```

6. Выдвижная панель (Drawer)

Боковое меню, реализуемое через ModalDrawerSheet и rememberDrawerState. Открывается по кнопке или жесту.

```
val drawerState = rememberDrawerState(DrawerValue.Closed)
Scaffold(modifier = Modifier.drawer(drawerState) { ModalDrawerSheet { ... } }) { ... }
```

7. Всплывающие сообщения (Snackbar)

Временное сообщение внизу экрана. Управляется через SnackbarHostState.

```
val snackState = remember { SnackbarHostState() }
Scaffold(snackbarHost = { SnackbarHost(snackState) }) { ... }
snackState.showSnackbar("Сообщение")
```

8. Диалоговые окна (AlertDialog)

Модальное окно с заголовком, текстом и кнопками.

```
if (showDialog) AlertDialog(
onDismissRequest = { showDialog = false },
title = { Text("Заголовок") },
text = { Text("Текст") },
confirmButton = { TextButton(onClick = {}) { Text("OK") } }
)
```

9. Выпадающее меню (DropdownMenu)

Появляется по нажатию на элемент, обычно на иконку.

```
var expanded by remember { mutableStateOf(false) }
IconButton(onClick = { expanded = true }) { Icon(Icons.Default.MoreVert, null) }
DropdownMenu(expanded = expanded, onDismissRequest = { expanded = false }) {
    DropdownMenuItem(text = { Text("Пункт") }, onClick = { ... })
}
```

10. Индикаторы прогресса

CircularProgressIndicator – круговой (неопределённый или детерминированный).

LinearProgressIndicator – линейный.

```
CircularProgressIndicator()                    // бесконечный
LinearProgressIndicator(progress = 0.5f)       // детерминированный
```

Практическая часть

Вариант 1. Экран входа с текстовыми полями и кнопкой

Создать экран входа в приложение, используя Scaffold с TopAppBar (заголовок «Вход»). В Scaffold содержимое разместить в Column с отступами. Добавить два OutlinedTextField: для логина и пароля (пароль скрыт точками). Под ними – Button с текстом «Войти». При нажатии на кнопку проверить, что поля не пустые. Если пустые – показать Snackbar с сообщением «Заполните все поля». Если заполнены – покажите AlertDialog с текстом «Добро пожаловать, [логин]!» и кнопкой «ОК». После закрытия диалога очистить поля. Для Snackbar использовать Scaffold с параметром snackbarHost.

Вариант 2. Настройки уведомлений с Checkbox, Switch и Slider

Реализовать экран настроек с помощью Scaffold и TopAppBar. Использовать Column с verticalScroll. Добавьте следующие элементы:

Checkbox с текстом «Включить уведомления» (при выключении остальные элементы становятся неактивными).

Switch с текстом «Звук уведомлений» (активен только если уведомления включены).

Slider от 0 до 100 для выбора громкости уведомлений (активен только если уведомления и звук включены). Отображайте текущее значение рядом.

RadioButton для выбора темы уведомлений: «Тёмная», «Светлая», «Системная» (активны, только если уведомления включены).

Все состояния хранить в rememberSaveable. При изменении настроек выводите Snackbar с кратким подтверждением («Настройки сохранены»).

Вариант 3. Список задач с чекбоксами и кнопкой добавления

Создать приложение «Список задач» с использованием Scaffold. В TopAppBar поместить заголовок «Мои задачи». В качестве кнопки добавления используйте FloatingActionButton с иконкой «+». При нажатии на кнопку добавления открывается AlertDialog с TextField для ввода названия задачи и кнопками «Добавить» / «Отмена». После добавления новая задача появляется в

LazyColumn списка. Каждая задача – это Row с Checkbox (отметка о выполнении) и текстом. При нажатии на чекбокс задача перечёркивается. Добавить возможность удаления задачи по долгому нажатию. Сохраняйте список задач через rememberSaveable.

Вариант 4. Выбор пола с RadioButton и подтверждением

Создать экран анкеты с Scaffold. В Column разместите:

Text «Ваш пол:»

Row с двумя RadioButton (Мужской, Женский) и соответствующими текстами.

Кнопку «Далее».

При выборе пола отображается под радиокнопками текст (например, «Выбран мужской пол»). При нажатии «Далее» показывайте AlertDialog с сообщением «Вы выбрали: [пол]» и кнопкой «ОК». Для группы радиокнопок использовать одно состояние. Сохраняйте выбор при повороте.

Вариант 5. Панель инструментов с иконками

Создать TopAppBar с заголовком «Редактор», навигационной иконкой (IconButton с Icons.Default.Menu) и двумя действиями справа: IconButton с иконкой поиска и IconButton с иконкой «три точки» (открывает меню). При нажатии на иконку меню показывайте DropdownMenu с пунктами «Настройки», «О программе», «Выход». При выборе пункта показывать Snackbar с соответствующим сообщением. Основной контент – просто Text «Здесь будет содержимое». Использовать Scaffold для интеграции панели.

Вариант 6. Плавная кнопка

Создать экран с Scaffold, где основное содержимое – LazyColumn из 20 элементов (номера 1..20). В качестве кнопки добавления использовать ExtendedFloatingActionButton с иконкой Icons.Default.Add и текстом «Добавить элемент». При нажатии на кнопку добавления в начало списка добавляется новый элемент. Реализовать BottomAppBar с простым текстом «Элементов: [количество]». Кнопка добавления должна быть встроена в Scaffold через параметр floatingActionButton. При добавлении элемента прокручивается список к началу.

Вариант 7. Экран с выдвижной панелью

Реализовать приложение с Scaffold, включающим TopAppBar с иконкой меню. При нажатии на иконку открывать выдвижную панель слева. В панели разместите Column с пунктами: «Главная», «Профиль», «Настройки», «Выход». При выборе пункта панель закрывается и в основном контенте меняется текст. Использовать rememberDrawerState и DrawerValue.Closed. Для обработки выбора использовать подъём состояния. Панель должна корректно открываться и закрываться жестом.

Вариант 8. Слайдер для выбора размера шрифта

Создать экран, на котором отображается образец текста («Пример текста для настройки размера»). Под текстом разместить Slider с диапазоном от 10 до 30 sp. Текущее значение отображайте рядом. Размер шрифта текста должен меняться синхронно с перемещением ползунка. Добавить Text с отображением текущего размера в sp. Использовать remember для хранения значения. Дополнительно:

добавить кнопку «Сброс» для установки размера по умолчанию. Обеспечить сохранение размера при повороте через rememberSaveable.

Вариант 9. Диалог подтверждения удаления

Реализовать список элементов. Каждый элемент имеет иконку корзины. При нажатии на корзину показывается AlertDialog с заголовком «Удалить?», сообщением «Вы уверены, что хотите удалить [имя элемента]?» и кнопками «Удалить» и «Отмена». При нажатии «Удалить» элемент исчезает из списка, показывается Snackbar «Элемент удалён». При нажатии «Отмена» диалог закрывается. Список хранится в mutableStateListOf. Использовать Scaffold для размещения списка и Snackbar.

Вариант 10. Индикаторы прогресса при загрузке данных

Создать экран с кнопкой «Загрузить данные». При нажатии на кнопку показывается CircularProgressIndicator, имитируя загрузку. Во время загрузки кнопка неактивна. После загрузки отображается Text «Данные загружены: [случайное число]». Индикатор прогресса должен исчезать. Реализовать два типа индикатора: круговой и линейный, переключаемые через Switch.

Вариант 11. Выпадающее меню из иконки

Создать TopAppBar с иконкой «три точки». При нажатии на неё открывается меню. В меню добавьте пункты: «Сортировать по имени», «Сортировать по дате», «Сброс фильтра». При выборе пункта в основном контенте меняется порядок отображения или фильтрацию. Реализовать простой список строк. Для сортировки использовать derivedStateOf. Меню должно закрываться после выбора. Использовать состояние expanded для управления видимостью меню.

Вариант 12. Поле ввода с чекбоксом «Показать пароль»

Создать экран с OutlinedTextField для пароля. Добавьте Row с Checkbox и текстом «Показать пароль». При установке чекбокса пароль должен отображаться открыто, иначе скрыт. Под полем отображается текст-индикатор сложности пароля (слабый/средний/сильный). Использовать derivedStateOf для вычисления сложности. Состояние пароля и чекбокса сохраняется при повороте.

Вариант 13. Экран с BottomAppBar и центральной кнопкой добавления

Создать Scaffold с BottomAppBar . В BottomAppBar разместите два IconButton (например, «Избранное» и «Профиль»). В качестве кнопки добавления использовать FloatingActionButton с иконкой «+», расположенную с выемкой (cutout) в BottomAppBar. При нажатии на кнопку добавления показывается Snackbar «Добавлено». При нажатии на кнопки в BottomAppBar меняется содержимое экрана. Реализовать управление состоянием текущего экрана.

Вариант 14. Переключатель Switch для режима «Не беспокоить»

Создать экран с Switch и текстом «Режим «Не беспокоить»». При включении переключателя все другие элементы управления становятся неактивными (enabled = false). При выключении – возвращаются в активное состояние. Использовать derivedStateOf для вычисления enabled для дочерних компонентов. Также при включении режима показывается Snackbar с сообщением «Режим «Не беспокоить» активирован». Сохраняется состояние переключателя при повороте.

Вариант 15. Кастомный переключатель с toggleable (модификатор)

Создать переключатель с помощью Box и модификатора Modifier.toggleable. Реализовать прямоугольник (60×30 dp) с кругом внутри. При нажатии на прямоугольник состояние меняется (вкл/выкл), цвет фона меняется, круг перемещается. Добавить текстовую метку «Wi-Fi» слева от переключателя, и сделать всю строку (Row) кликабельной через toggleable. Использовать одно состояние для переключателя. Сохраняется состояние переключателя при повороте.

Вариант 16. Выбор одного элемента из списка с selectable

Создать список из 4 элементов. Реализовать выбор одного элемента с помощью модификатора Modifier.selectable. Каждый элемент – Row с текстом и RadioButton, но радиокнопка должна быть только визуальным индикатором, а выбор происходит при нажатии на всю строку. Использовать состояние selectedIndex. При выборе другого элемента предыдущий снимается. Под списком отобразить текст «Выбрано: [название]». Сохранение выбор при повороте.

Вариант 17. Экран с прогрессом загрузки файла

Создать экран, имитирующий загрузку файла. Добавить кнопку «Начать загрузку». При нажатии запускается анимированный прогресс (от 0 до 100 за 5 секунд). Используйте LaunchedEffect и animateFloatAsState для значения прогресса. Отображается LinearProgressIndicator с текущим прогрессом, а также текст «Загружено X%». Во время загрузки кнопка «Начать загрузку» неактивна, появляется кнопка «Отмена». При нажатии «Отмена» загрузка прерывается, прогресс сбрасывается. Реализовать CircularProgressIndicator с неопределённым режимом (progress = null) на время, пока загрузка не начата. Управляние состояниями через remember.

Вариант 18. Комплексный экран профиля

Разработать экран профиля пользователя, используя Scaffold с TopAppBar (заголовок «Профиль», иконка редактирования). Контент в Column:

Text (имя пользователя) – изначально «Имя не задано».

OutlinedTextField для ввода имени.

Button «Сохранить» – обновляет имя.

Switch «Получать уведомления».

Checkbox «Согласен с условиями».

Slider для выбора уровня опыта (0-100).

RadioButton для выбора темы (светлая/тёмная).

ExtendedFloatingActionButton с текстом «Поделиться» (при нажатии – Snackbar).

Все состояния должны сохраняться при повороте. Кнопка «Сохранить» активна только если имя не пустое и чекбокс согласия отмечен. При сохранении показывать AlertDialog с подтверждением.

Лабораторная работа 7. Ресурсы в Jetpack Compose

Цель работы: изучить способы работы с ресурсами приложений Android в Jetpack Compose: строковые ресурсы, размерные ресурсы (dimension) и цветовые ресурсы. Научиться извлекать и использовать значения из strings.xml, dimens.xml и colors.xml для создания адаптивных и локализуемых интерфейсов.

Теоретическая часть

1. Ресурсы строк

Строковые ресурсы хранятся в файле res/values/strings.xml. Они позволяют локализовать приложение и упрощают поддержку текста.

```xml
<!-- res/values/strings.xml -->
<resources>
    <string name="app_name">Моё приложение</string>
    <string name="greeting">Привет, %s!</string>
    <string name="welcome_message">Добро пожаловать в %1$s, %2$s</string>
</resources>
```

В Compose для получения строки используется функция stringResource():

```
Text(text = stringResource(R.string.app_name))

// С форматированием
Text(text = stringResource(R.string.greeting, "Анна"))
Text(text = stringResource(R.string.welcome_message, "Jetpack Compose", "Иван"))
```

2. Ресурсы размеров

Размерные ресурсы хранятся в res/values/dimens.xml. Они позволяют задавать отступы, размеры шрифтов, ширину/высоту элементов централизованно.

```xml
<!-- res/values/dimens.xml -->
<resources>
    <dimen name="padding_small">8dp</dimen>
    <dimen name="padding_medium">16dp</dimen>
    <dimen name="text_title">20sp</dimen>
    <dimen name="button_width">200dp</dimen>
</resources>
```

В Compose размер из ресурсов получается через dimensionResource(). Возвращаемое значение имеет тип Dp.

```
val paddingSmall = dimensionResource(R.dimen.padding_small)
val titleSize = dimensionResource(R.dimen.text_title)

Text(
    text = "Заголовок",
    fontSize = titleSize,
    modifier = Modifier.padding(paddingSmall)
)
```

3. Ресурсы цвета

Цветовые ресурсы хранятся в res/values/colors.xml. Это удобно для поддержки единой цветовой схемы и тем.

В Compose цвет из ресурсов получается через colorResource(), который возвращает Color.

```
val primaryColor = colorResource(R.color.primary)
val bgColor = colorResource(R.color.background)

Box(
    modifier = Modifier
        .background(bgColor)
        .padding(16.dp)
) {
    Text("Цветной текст", color = primaryColor)
}
```

Функции stringResource(), dimensionResource(), colorResource() являются composable-функциями и могут использоваться только внутри composable-контекста или с LocalContext.current.resources.

Для использования вне композиции (например, в ViewModel) можно получить ресурсы через context.resources.

Практическая часть

Реализовать приложение с общим шаблоном ресурсов по заданному варианту
Образец шаблона:

```xml
<!-- strings.xml -->
<resources>
<string name="app_name">Название приложения</string>
<string name="title">Заголовок</string>
<string name="counter">Счётчик: %d</string>
<plurals name="items_count">
<item quantity="one">%d элемент</item>
<item quantity="few">%d элемента</item>
<item quantity="many">%d элементов</item>
</plurals>
</resources>

<!-- dimens.xml -->
<resources>
<dimen name="padding_small">8dp</dimen>
<dimen name="padding_medium">16dp</dimen>
<dimen name="text_title">20sp</dimen>
<dimen name="button_height">48dp</dimen>
</resources>

<!-- colors.xml (светлая тема) -->
<resources>
<color name="primary">#2196F3</color>
<color name="background">#FFFFFF</color>
<color name="on_background">#000000</color>
</resources>
```

Вариант 1. Счётчик калорий

Пользователь вводит название продукта и количество калорий. Список продуктов отображается с итоговой суммой калорий. Используются строки для названий, размеры для отступов и полей ввода, цвета для выделения продуктов с высокой калорийностью.

Вариант 2. Список дел

Добавление/удаление задач, отметка о выполнении. Задачи хранятся в списке. Ресурсы: заголовки, кнопки, сообщения (например, «Задача добавлена»). Плюрализация для количества задач.

Вариант 3. Таймер обратного отсчёта

Пользователь задаёт время (минуты/секунды), запускает таймер. Отображение оставшегося времени, звуковой сигнал по окончании. Ресурсы: подписи полей, сообщения о завершении, форматирование времени.

Вариант 4. Конвертер валют

Выбор валюты из списка, ввод суммы, расчёт по курсу. Курсы можно задать константами. Ресурсы: названия валют, форматированная строка результата («X рублей = Y долларов»).

Вариант 5. Викторина

5 вопросов с вариантами ответов. После ответа показывается правильный/неправильный и итоговый счёт. Ресурсы: вопросы, варианты ответов, сообщения о результате.

Вариант 6. Калькулятор чаевых

Ввод суммы счёта, процента чаевых, количества гостей. Расчёт суммы чаевых и общей суммы на каждого. Ресурсы: подписи полей, форматированная строка результата.

Вариант 7. Генератор случайных чисел

Выбор диапазона (мин/макс), кнопка «Сгенерировать». Отображение случайного числа и истории генераций. Ресурсы: заголовки, сообщения об ошибках (если мин > макс).

Вариант 8. Заметки с тегами

Добавление заметки и тегов к ней. Фильтр по тегам. Список заметок. Ресурсы: названия полей, сообщения о пустом теге, плюрализация для количества заметок по тегу.

Вариант 9. Бюджет (расходы)

Добавление расходов с категориями (еда, транспорт, развлечения). Отображение суммы по категориям и общей суммы. Ресурсы: названия категорий, **форматирование даты.**

Вариант 10. Погодное приложение

Выбор города (из списка или ввод), отображение температуры, описания погоды. Данные можно сгенерировать. Ресурсы: названия городов, описания погоды («солнечно», «дождливо»).

Вариант 11. Планировщик привычек

Список привычек с отметкой о выполнении за сегодня. Отображение прогресса (процент выполнения). Ресурсы: названия привычек, сообщения о достижении цели.

Вариант 12. Матрица умножения

Генерация случайных примеров на умножение, поле ввода ответа, проверка. Подсчёт правильных ответов. Ресурсы: пример («%d × %d = ?»), сообщения о результате.

Вариант 13. Список книг (библиотека)

Отображение списка книг (название, автор, год). Возможность добавить/удалить. Ресурсы: заголовки колонок, названия кнопок, плюрализация для количества книг.

Вариант 14. Трекер воды

Кнопки «Выпить стакан» (200 мл) и «Сброс». Отображение выпитого объёма и дневной нормы (2 л). Ресурсы: сообщение «Выпито X мл из Y мл», прогресс-бар.

Вариант 15. Парольный менеджер

Сохранение паролей для сайтов (название сайта, логин, пароль). Список сохранённых записей. Ресурсы: подписи полей, сообщения о сохранении.

Вариант 16. Рецепты (кулинарная книга)

Список рецептов (название, время приготовления). При нажатии – подробное описание (из строкового ресурса с переносами). Ресурсы: названия рецептов, описания.

Вариант 17. Игра «Угадай число»

Компьютер загадывает число от 1 до 100, пользователь вводит догадки. Подсказки «больше», «меньше». Ресурсы: сообщения подсказок, поздравления.

Вариант 18. Профиль спортсмена

Имя, вид спорта, достижения (медали, рекорды). Возможность редактировать достижения. Ресурсы: названия видов спорта, форматирование рекордов, плюрализация для медалей.

Лабораторная работа 8. Работа с изображениями

Цель работы: изучить способы отображения и обработки изображений в Jetpack Compose. Научиться загружать изображения из ресурсов, из интернета, применять модификаторы для масштабирования, обрезки, наложения эффектов, создавать простые графические элементы через Canvas.

Теоретическая часть

1. Основные компоненты для работы с изображениями

Image – отображение растровых изображений

```
// Из ресурсов drawable
Image(
painter = painterResource(R.drawable.ic_avatar),
contentDescription = "Аватар пользователя",
modifier = Modifier.size(64.dp)
)

// Из интернета (библиотека Coil)
// Добавить зависимость: implementation("io.coil-kt:coil-compose:2.6.0")
Image(
painter = rememberAsyncImagePainter("https://example.com/image.jpg"),
contentDescription = "Изображение из сети"
)
```

2. Модификаторы для изображений

Modifier.size() − задание размера;

Modifier.clip(CircleShape) − обрезка до круга;

Modifier.aspectRatio(1f) − сохранение пропорций;

Modifier.alpha(0.5f) − прозрачность;

Modifier.blur(radius) − размытие;
Modifier.colorFilter(ColorFilter.tint(Color.Red)) − наложение цвета.

```
Image(
    painter = painterResource(R.drawable.photo),
    contentDescription = null,
    modifier = Modifier
        .size(100.dp)
        .clip(CircleShape)
        .border(2.dp, Color.White, CircleShape)
)
```

3. Работа с Bitmap и кастомная отрисовка

```
Canvas(modifier = Modifier.fillMaxSize()) {
    drawCircle(color = Color.Blue, radius = 100f, center = center)
    drawRect(color = Color.Red, topLeft = Offset(50f, 50f), size = Size(200f, 100f))
    drawImage(image = bitmap, topLeft = Offset.Zero)
}
```

4. Загрузка изображений из интернета
Зависимость:

```
implementation("io.coil-kt:coil-compose:2.6.0")
```

```
Image(
painter = rememberAsyncImagePainter(
model = "https://example.com/image.jpg",)
```

Практическая часть

Все приложения должны использовать изображения (из drawable-ресурсов или из интернета). Для каждого приложения необходимо создать ресурсы: строки, размеры, цвета. Условия:

Все тексты – из strings.xml;

Все размеры (отступы, размеры изображений) – из dimens.xml;

Все цвета – из colors.xml (с поддержкой светлой/тёмной темы);

Изображения для ресурсов можно использовать стандартные или добавить свои в res/drawable.

Вариант 1. Галерея изображений

Создать экран с горизонтальным списком из 5 изображений. Каждое изображение имеет размер 120×120 dp, скруглённые углы 8 dp. При нажатии на изображение оно увеличивается до 150×150 dp (анимация) и показывается Snackbar

с названием изображения (из строкового ресурса). Отступы между изображениями – из dimens.xml. Цвет фона экрана – из colors.xml.

Вариант 2. Аватар с возможностью выбора из галереи

Экран профиля: круглый аватар (размер 100 dp) с иконкой фотоаппарата поверх. При нажатии на аватар открывается диалог с выбором: «Сделать фото» или «Выбрать из галереи» (пока просто заглушка – меняем аватар на другой drawable-ресурс). Используйте плюрализацию для сообщений. Все строки и размеры – из ресурсов.

Вариант 3. Карточка товара с изображением

Создать карточку товара: изображение товара (из ресурсов, 200×150 dp), название, цена, кнопка «Купить». Изображение должно занимать всю ширину карточки с сохранением пропорций (Modifier.aspectRatio(4f/3f)). Цена выделена цветом из colors.xml. При нажатии на кнопку – Snackbar с форматированной строкой («Товар «%s» добавлен»). Реализовать тёмную тему.

Вариант 4. Фоторедактор

Отобразить изображение из ресурсов. Добавить три кнопки: «Красный фильтр», «Зелёный фильтр», «Сброс». При нажатии на кнопку применить Modifier.colorFilter(ColorFilter.tint(...)) к изображению. Размер изображения – 300 dp по ширине, высота – пропорционально. Цвета фильтров – из ресурсов. Кнопки должны менять свою активность (выбранный фильтр подсвечивается).

Вариант 5. Список пользователей с аватарами из интернета

Загрузить список из 5 пользователей (имя, статус, URL аватара). Отобразить аватар (40×40 dp, круглый), имя и статус. При ошибке загрузки показывает стандартную иконку из ресурсов. Реализовать LazyColumn. Размеры и цвета – из ресурсов.

Вариант 6. Панорама (горизонтальное масштабирование)

Отобразить широкое изображение (из ресурсов, ширина 800 dp) внутри Box с Modifier.horizontalScroll. Пользователь может прокручивать изображение пальцем. Добавить текст «Панорама» над изображением. Использовать Modifier.aspectRatio(2f/1f) для сохранения пропорций. Все отступы – из dimens.xml.

Вариант 7. Анимированный баннер (смена изображений)

Создать карусель из 3 изображений, которые автоматически сменяются каждые 3 секунды. Отображение индикатора текущего слайда (три точки или кружка). При нажатии на изображение останавливается автопереключение. Для анимации использовать animateFloatAsState и Modifier.graphicsLayer. Ресурсы: строки для описаний, размеры индикаторов.

Вариант 8. Кастомный QR-код

Сгенерировать изображение QR-кода (нарисовать чёрно-белые квадраты в Canvas размером 200×200 dp). QR-код кодирует текст из строкового ресурса.

Добавить кнопку «Сохранить». Используйте drawRect в Canvas. Размеры и цвета – из ресурсов (чёрный, белый).

Вариант 9. Отражение изображения

Отобразить изображение из ресурсов. Добавить кнопку «Отразить по горизонтали». При нажатии применять Modifier.graphicsLayer { scaleX = -1f }. При повторном нажатии – возврат. Анимация поворота. Цвет кнопки – из colors.xml. Размер изображения – 200×200 dp, с закруглёнными углами 16 dp.

Вариант 10. Фоторамка

Создать изображение (из ресурсов) с рамкой: Modifier.border(4.dp, Color.Gray, RoundedCornerShape(12.dp)) и тенью (Modifier.shadow(8.dp)). Добавить подпись «Моё фото» под изображением. Отступы и размер шрифта – из ресурсов. Реализовать возможность переключения рамки между серой и золотой через RadioButton (цвета из ресурсов).

Вариант 11. Прогресс загрузки изображения

Загрузить изображение из интернета по URL. Пока загружается – показывает CircularProgressIndicator. При ошибке – показывает изображение-заглушку из ресурсов и текст ошибки из строковых ресурсов. Использовать rememberAsyncImagePainter с обработкой состояний.

Вариант 12. Сравнение двух изображений

Отобразить два изображения в одном контейнере. Добавить вертикальную разделительную линию, которую можно перетаскивать горизонтально. Левое изображение обрезается справа по положению разделителя, правое – слева.

Вариант 13. Список книг – с обложками

Отображение списка книг (название, автор, год). Каждая книга имеет миниатюру обложки (из ресурсов drawable – разные цвета или абстрактные рисунки). При долгом нажатии на книгу показывается диалог с увеличенной обложкой. При добавлении книги можно выбрать цвет обложки из палитры. Плюрализация для количества книг.

Вариант 14. Трекер воды

Кнопки «Выпить стакан» (200 мл) и «Сброс». Отобразить выпитый объём и дневную норму в виде уровня жидкости в графическом стакане. Уровень воды анимированно поднимается при добавлении. Рядом – иконка капли, меняющая цвет от красного (мало воды) до зелёного (норма).

Вариант 15. Парольный менеджер

Сохранение паролей для сайтов. Каждый сайт имеет логотип-иконку (из ресурсов: google, facebook, github). При добавлении нового сайта пользователь выбирает иконку из списка. Список сохранённых записей отображается с иконками. При нажатии на запись иконка анимированно вращается (показывает, что пароль скопирован).

Вариант 16. Рецепты – с фото блюд

Список рецептов (название, время приготовления). Каждый рецепт имеет миниатюру блюда. При нажатии – подробное описание с увеличенным фото. Фото можно сделать с закруглёнными углами и тенью. Добавить возможность ставить «лайк» рецепту – иконка сердца меняет цвет. Ресурсы: названия рецептов, описания (из строковых ресурсов с переносами).

Вариант 17. Игра «Угадай число»

Компьютер загадывает число от 1 до 100. Пользователь вводит догадки. Подсказки «больше»/«меньше» отображаются в виде стрелок-иконок (вверх/вниз). При правильном ответе показывайте анимированную звёздочку или салют. Термометр показывает, насколько близко предположение к загаданному числу (чем ближе – тем выше уровень ртути). Ресурсы: сообщения подсказок, поздравления.

Вариант 18. Профиль спортсмена

Имя, вид спорта, достижения (медали, рекорды). Аватар – круглое фото из ресурсов. Медали отображаются как иконки (золото, серебро, бронза). При редактировании достижений можно выбрать иконку медали из списка. Рекорды отображаются на фоне спортивного кубка. Плюрализация для количества медалей. Реализуйте возможность смены аватара (из галереи-заглушки).

Лабораторная работа 9. Архитектурные шаблоны и паттерны

Цель работы: изучить принципы построения архитектуры Android-приложений с использованием ViewModel, управление UI состоянием (StateFlow, Compose State), реализацию однонаправленного потока данных (Unidirectional Data Flow – UDF). Научиться отделять бизнес-логику от UI, корректно обрабатывать события, сохранять состояние при повороте экрана и тестировать логику.

Теоретическая часть

Основные компоненты архитектуры

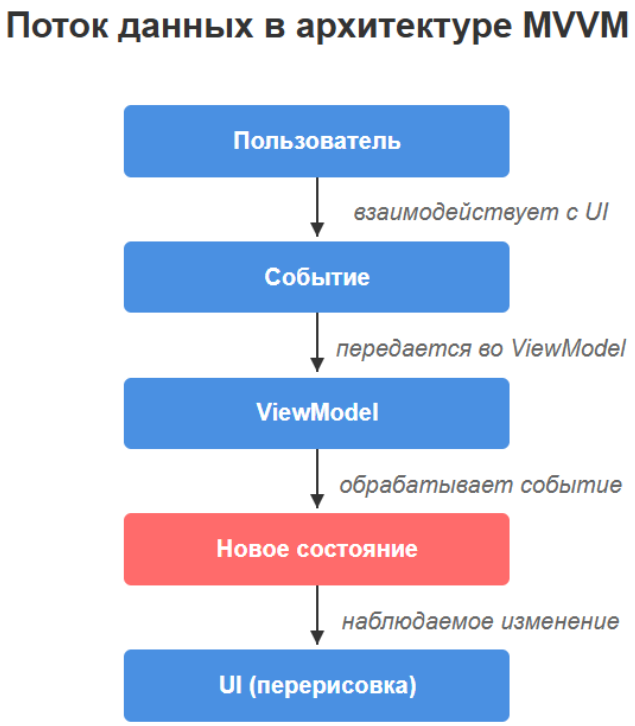
UI (Composable) – отображает состояние и отправляет события (действия пользователя) во ViewModel.

ViewModel – хранит и обрабатывает состояние, не зависит от жизненного цикла UI, переживает поворот экрана.

State – неизменяемые данные, описывающие экран в конкретный момент.

Event – действия пользователя (нажатие кнопки, ввод текста), которые вызывают изменение состояния.

Однонаправленный поток данных



Состояние хранится во ViewModel как StateFlow или MutableStateFlow.
UI подписывается на состояние через collectAsState().
UI отправляет события вызовом методов ViewModel.

ViewModel в Compose

```
    private val _uiState = MutableStateFlow(MyUiState())
    val uiState: StateFlow<MyUiState> = _uiState.asStateFlow()

    fun handleEvent(event: MyEvent) {
        when (event) {
            is MyEvent.OnButtonClick -> {
                _uiState.update { it.copy(counter = it.counter + 1) }
            }
        }
    }
}

data class MyUiState(val counter: Int = 0, val isLoading: Boolean = false)

@Composable
fun MyScreen(viewModel: MyViewModel = viewModel()) {
    val uiState by viewModel.uiState.collectAsState()
    // отображение uiState, вызов viewModel.handleEvent()
}
```

Практическая часть

Реализовать полноценный экран с соблюдением архитектуры MVVM: отдельный ViewModel, управление состоянием через StateFlow, передача событий, обработка побочных эффектов. Все строки, размеры, цвета – в ресурсах.

Вариант 1 Счётчик

Реализовать экран-счётчик с кнопками «+1», «-1», «Сброс». Состояние хранится во ViewModel как StateFlow<Int>. При нажатии кнопок отправляется события во ViewModel. Счётчик не должен сбрасываться при повороте экрана. Добавить ограничение: минимальное значение 0. При попытке уменьшить ниже 0 – показывается Snackbar с сообщением из ресурсов**.**

Вариант 2 Форма входа с валидацией

Поля: логин, пароль, кнопка «Войти». Состояние хранит логин, пароль, флаг валидности формы. Валидация выполняется во ViewModel при каждом изменении полей. При успешном входе отправьте однократное событие ShowSnackbar("Добро пожаловать").

Вариант 3 Бронирование столиков в ресторане

Приложение для выбора даты, времени и количества гостей. Пользователь выбирает дату (календарь), время (список: 18:00, 19:00, 20:00) и количество гостей (1-10). По кнопке «Забронировать» проверяется, не забронировано ли уже это время на выбранное количество гостей (имитация: если время 19:00 и гостей >4 – занято). При успехе – показывается Snackbar «Столик забронирован». При неудаче – сообщение об ошибке.

Вариант 4 Калькулятор чаевых с разделением на компанию

Ввод суммы счёта, процента чаевых (Slider 0-30), количества человек. Отображение суммы чаевых на каждого, общей суммы на каждого и общей суммы счёта. Добавить возможность округления суммы до целого (чекбокс «Округлить вверх»). При изменении любого параметра – мгновенный пересчёт.

Вариант 5 Список дел с приоритетами и сортировкой

Задача имеет текст, приоритет (высокий, средний, низкий) и статус выполнения. Можно добавить задачу, удалить, отметить выполненной. Сортировка: по приоритету, по дате добавления, по статусу. Фильтр: показать только высокий приоритет. Количество задач отображается с плюрализацией.

Вариант 6 Тренажёр английских слов

Показывается слово на русском, пользователь вводит перевод на английском. При проверке – правильно/неправильно. После ответа – следующее слово. Словарь из 10 слов (задан в ресурсах). В конце – количество правильных ответов. Кнопка «Сброс» для начала заново.

Вариант 7 Планировщик тренировок (упражнения с таймером)

Список упражнений (приседания, отжимания, планка). Для каждого – кнопка «Старт», запускающая таймер на заданное количество секунд (10, 15, 20). Во время таймера показывается обратный отсчёт, кнопка «Стоп». По окончании – звук (вибрация) и отметка о выполнении. Можно сбросить все выполнения.

Вариант 8 Шахматная доска

Сетка 8×8, каждая клетка – Box с цветом. При нажатии на клетку она выделяется рамкой. Если выделена одна клетка, а затем нажата другая – «фигура» перемещается (цвета меняются местами). Отображается текущая выбранная клетка.

Вариант 9 Приложение «Мои заметки» с шифрованием

Список заметок (заголовок, текст). При добавлении заметки текст шифруется сдвигом (Caesar cipher, ключ 3). При просмотре – расшифровывается. Возможность удалить. Поиск по заголовкам.

Вариант 10 Счётчик калорий с дневной нормой и графиком

Пользователь добавляет приёмы пищи (название, калории). Отображается сумма за сегодня, остаток до нормы (2000 ккал). Прогресс-бар. История по дням (можно выбрать дату из календаря).

Вариант 11 Меню ресторана с корзиной и оформлением заказа

Список блюд (название, цена, количество в корзине). Кнопки «+»/«-» меняют количество в корзине. Отображается итоговая сумма. Кнопка «Оформить заказ» – диалог подтверждения, после чего корзина очищается и показывается Snackbar.

Вариант 12 Игра «Сапёр» (упрощённая, 4×4)

Поле 4×4, 2 мины. При нажатии на клетку – показывается число (количество мин по соседству) или «мина» (игра заканчивается). При открытии всех безопасных клеток – победа. Счётчик попыток.

Вариант 13 Конвертер температуры с историей

Ввод температуры, выбор шкалы (Цельсий, Фаренгейт, Кельвин). При нажатии «Конвертировать» отображается результат во всех трёх шкалах. История всех конвертаций (входное значение + результат). Кнопка «Очистить историю».

Вариант 14 Чат-бот

Пользователь вводит сообщение, бот отвечает по правилам: если есть слово «привет» – «Здравствуйте!», «как дела» – «Хорошо, спасибо», иначе – «Я вас не понял». История сообщений отображается в виде пузырей (слева – бот, справа – пользователь).

Вариант 15 Трекер сна

Кнопки «Лёг спать» и «Проснулся». При нажатии запоминается время (часы, минуты). Отображается продолжительность последнего сна и средняя за последние 7 дней. История записей.

Вариант 16 Генератор QR-кода для текста

Поле ввода текста, кнопка «Сгенерировать». На Canvas рисуется QR-код (имитация – просто чёрно-белые квадраты, можно без реального кодирования, просто случайный паттерн). Кнопка «Сохранить» – показывается Snackbar «Изображение сохранено».

Вариант 17 Экран «О приложении» с раскрывающимися разделами

Accordion-список разделов: «Разработчики», «Версия», «Лицензия», «Политика конфиденциальности». При нажатии на раздел раскрывается содержимое (анимация). Содержимое – строки из ресурсов. Возможность только одного открытого раздела одновременно.

Лабораторная работа 10. Работа с базами данных

Цель работы: изучить основы работы с базами данных в Android на примере SQLite. Научиться создавать базу данных, выполнять операции CRUD (Create, Read, Update, Delete), отображать данные в списках, использовать SQLiteOpenHelper, Cursor и ContentValues.

Теоретическая часть

SQLite – встраиваемая реляционная база данных с открытым исходным кодом, используемая в Android по умолчанию. Поддерживает основные типы: INTEGER, REAL, TEXT, BLOB. Все операции выполняются через SQL-запросы или вспомогательные методы.

SQLiteOpenHelper

Абстрактный класс для управления созданием и обновлением базы данных. Обязательные методы:

onCreate(SQLiteDatabase db) – вызывается при первом создании БД (создание таблиц).

onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) – вызывается при изменении версии БД.

Основные операции

Вставка: insert(String table, String nullColumnHack, ContentValues values);

Обновление: update(String table, ContentValues values, String whereClause, String[] whereArgs);

Удаление: delete(String table, String whereClause, String[] whereArgs);

Запрос: query(...) или rawQuery(String sql, String[] selectionArgs).

В современной Android-разработке рекомендуется использовать библиотеку Room для работы с SQLite, но для понимания основ полезно знать чистый SQLite через SQLiteOpenHelper.

Пример создания таблицы

```
class DatabaseHelper(context: Context) :
    SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {

    companion object {
        private const val DATABASE_NAME = "students.db"
        private const val DATABASE_VERSION = 1

        // Имя таблицы и колонки
        const val TABLE_STUDENTS = "students"
        const val COLUMN_ID = "_id"
        const val COLUMN_NAME = "name"
        const val COLUMN_GROUP = "group_name"
        const val COLUMN_SCORE = "score"
    }

    // Создание таблицы (вызывается 1 раз при отсутствии БД)
    override fun onCreate(db: SQLiteDatabase) {
        val createTable = """
            CREATE TABLE $TABLE_STUDENTS (
                $COLUMN_ID INTEGER PRIMARY KEY AUTOINCREMENT,
                $COLUMN_NAME TEXT NOT NULL,
                $COLUMN_GROUP TEXT,
                $COLUMN_SCORE INTEGER DEFAULT 0
            )
        """.trimIndent()
        db.execSQL(createTable)
    }

}
```

CRUD операции

Вставка записи

```
fun addStudent(name: String, group: String, score: Int): Long {
    val db = writableDatabase
    val values = ContentValues().apply {
        put(COLUMN_NAME, name)
        put(COLUMN_GROUP, group)
        put(COLUMN_SCORE, score)
    }
    return db.insert(TABLE_STUDENTS, null, values)
}
```

Чтение всех записей

```
fun getAllStudents(): Cursor {
    val db = readableDatabase
    return db.query(
        TABLE_STUDENTS,
        null,           // все колонки
        null,           // selection
        null,           // selectionArgs
        null,           // groupBy
        null,           // having
        "$COLUMN_NAME ASC"  // sortOrder
    )
}
```

Обновление записи

```
fun updateStudent(id: Long, newName: String, newGroup: String, newScore: Int): Int {
    val db = writableDatabase
    val values = ContentValues().apply {
        put(COLUMN_NAME, newName)
        put(COLUMN_GROUP, newGroup)
        put(COLUMN_SCORE, newScore)
    }
    return db.update(TABLE_STUDENTS, values, "$COLUMN_ID = ?", arrayOf(id.toString()))
}
```

Удаление записи

```
fun deleteStudent(id: Long): Int {
    val db = writableDatabase
    return db.delete(TABLE_STUDENTS, "$COLUMN_ID = ?", arrayOf(id.toString()))
}
```

Практическая часть

Реализовать приложение, работающее с базой данных SQLite. Требования общие для всех вариантов:

Использовать SQLiteOpenHelper.

Реализовать все операции CRUD (добавление, чтение, удаление, обновление) согласно условию.

Интерфейс должен быть удобным: список элементов, поля для ввода, кнопки.

Все строки, размеры и цвета вынести в ресурсы.

Приложение должно корректно работать при повороте экрана.

Вариант 1. Записная книжка (контакты)

Создать приложение для хранения контактов. Каждый контакт содержит имя, телефон, email. Реализовать:

Добавление нового контакта (форма с тремя полями).

Отображение всех контактов в списке (RecyclerView или ListView).

Возможность удаления контакта по долгому нажатию.

Возможность редактирования контакта (по нажатию – открывается диалог с полями).

Вариант 2. Список покупок

Приложение для ведения списка покупок. Поля: название продукта, количество, цена за единицу. Реализовать:

Добавление продукта.

Отображение списка с итоговой суммой по каждому продукту (цена × количество) и общей суммой.

Возможность отметить продукт как «купленный» (чекбокс, меняет цвет или зачёркивание).

Удаление продукта.

Редактирование количества.

Вариант 3. Библиотека (книги)

Хранение списка книг. Поля: название, автор, год издания, прочитана (да/нет). Реализовать:

Добавление книги.

Список всех книг (название, автор).

Фильтр: показывать только непрочитанные (кнопка «Показать все» / «Непрочитанные»).

Возможность отметить книгу как прочитанную.

Удаление книги.

Плюрализация для количества книг.

Вариант 4. Дневник тренировок

Приложение для записи тренировок. Поля: дата, вид тренировки (бег, плавание, тренажёрный зал), длительность (минуты), сожжённые калории. Реализовать:

Добавление тренировки.

Список тренировок, отсортированный по дате (последние сверху).

Возможность удалить тренировку.

Отображение итоговой суммы калорий за выбранный месяц.

Вариант 5. Расходы

Учёт расходов. Поля: сумма, категория (еда, транспорт, развлечения, другое), дата, комментарий. Реализовать:

Добавление расхода.

Список расходов с суммой и категорией.

Фильтр по категории.

Итоговая сумма по текущему фильтру.

Удаление расхода.

Вариант 6. Заметки с напоминаниями

Заметки, к которым можно привязать дату-время напоминания. Поля: заголовок, текст заметки, дата напоминания (может быть пустой). Реализовать:

Добавление/редактирование заметки.

Список заметок (заголовок и дата напоминания).

При наступлении даты напоминания (при запуске приложения проверять) показывать уведомление.

Удаление заметки.

Возможность отключить напоминание.

Вариант 7. Студенты и оценки

База данных студентов. Поля: ФИО, группа, список оценок (можно хранить в отдельной таблице или в виде строки). Реализовать:

Добавление студента.

Список студентов.

Добавление оценки студенту (диалог с выбором оценки).

Отображение среднего балла каждого студента.

Удаление студента.

Вариант 8. Рецепты

Хранение кулинарных рецептов. Поля: название блюда, категория (суп, салат, горячее, десерт), список ингредиентов (текст), инструкция (текст). Реализовать:

Добавление рецепта.

Список рецептов с названием и категорией.

Фильтр по категории.

Просмотр подробностей (ингредиенты, инструкция) в отдельном экране или диалоге.

Удаление рецепта.

Вариант 9. Список фильмов

База данных фильмов. Поля: название, режиссёр, год, жанр, оценка (от 1 до 5). Реализовать:

Добавление фильма.

Список фильмов с названием, годом и оценкой.

Сортировка по году или по названию.

Фильтр по жанру.

Удаление фильма.

Возможность изменить оценку.

Вариант 10. Маршруты

Хранение маршрутов походов. Поля: название маршрута, регион, протяжённость (км), сложность (1-5), дата прохождения. Реализовать:

Добавление маршрута.

Список маршрутов (название, сложность, протяжённость).

Фильтр по сложности (выше/ниже заданной).

Возможность отметить маршрут как «пройденный».

Удаление маршрута.

Вариант 11. Планы на день

Задачи с приоритетом (высокий, средний, низкий) и датой выполнения. Поля: задача, приоритет, дата, выполнено (да/нет). Реализовать:

Добавление задачи.

Список задач, сгруппированный по дате.

Сортировка по приоритету.

Отметка о выполнении.

Удаление выполненных задач за один раз (кнопка «Очистить»).

Вариант 12. Инвентаризация

Учёт товаров на складе. Поля: наименование, артикул, количество, цена закупки. Реализовать:

Добавление товара.

Список товаров с количеством и общей стоимостью (количество × цена).

Возможность увеличить/уменьшить количество (кнопки «+» / «-»).

Поиск по наименованию или артикулу.

Удаление товара.

Вариант 13. Менеджер паролей

Хранение паролей от сайтов. Поля: название сайта, логин, пароль (желательно зашифровать простым шифром). Реализовать:

Добавление записи.

Список записей (сайт, логин).

Просмотр пароля по нажатию (диалог).

Возможность скопировать пароль в буфер обмена.

Удаление записи.

Поиск по названию сайта.

Вариант 14. Расписание занятий

Хранение расписания на неделю. Поля: день недели, время начала, название предмета, аудитория, преподаватель. Реализовать:

Добавление/редактирование занятия.

Список занятий, сгруппированный по дням недели (Monday…Friday).

Возможность выбрать неделю (числитель/знаменатель).

Удаление занятия.

Экспорт расписания в текстовый файл.

Вариант 15. Медицинская карта

Хранение записей о посещении врача. Поля: дата приёма, врач, диагноз, назначения. Реализовать:

Добавление записи.

Список записей, отсортированный по дате.

Фильтр по врачу.

Возможность редактировать диагноз/назначения.

Удаление записи.

Вариант 16. Автопарк (учёт автомобилей)

Список автомобилей с их характеристиками. Поля: марка, модель, год выпуска, пробег, номерной знак. Реализовать:

Добавление автомобиля.

Список автомобилей (марка, модель, номер).

Возможность добавить запись о техническом обслуживании (новая таблица или отдельное поле).

Удаление автомобиля.

Поиск по номеру.

Вариант 17. Цитатник

Хранение цитат с авторами и тегами. Поля: текст цитаты, автор, тег (юмор, мотивация, философия и т.п.). Реализовать:

Добавление цитаты.

Список цитат (текст, автор).

Фильтр по тегу.

Возможность отметить цитату как «избранное».

Удаление цитаты.

Случайная цитата (кнопка «Показать случайную»).

Вариант 18. Журнал оценок учеников

Две таблицы: Students (id, ФИО) и Grades (id_student, предмет, оценка, дата). Реализовать:

Добавление ученика.

Список учеников.

Добавление оценки выбранному ученику (выбор предмета, оценка, дата).

Просмотр всех оценок ученика с подсчётом среднего балла по каждому предмету.

Удаление ученика.

Удаление оценки.

Лабораторная работа 11. Работа с сетью

Цель работы: изучить основные методы взаимодействия с сетью на платформе Android. Научиться выполнять HTTP-запросы, обрабатывать ответы в форматах JSON/XML.

Теоретическая часть

Архитектура клиент-сервер

В классической модели клиент-сервер:

Сервер – программа, которая ожидает входящие запросы, выполняет бизнес-логику (вычисления, работа с данными) и возвращает результат.

Клиент – программа, которая отправляет запросы к серверу, получает ответы и отображает информацию пользователю.

Формат обмена данными – JSON.

JSON (JavaScript Object Notation) – текстовый формат обмена данными, основанный на синтаксисе JavaScript. Пример JSON-объекта.

В Android для парсинга JSON используется библиотека Gson (преобразует JSON в Kotlin-классы и обратно).

В Java есть встроенный легковесный HTTP-сервер в пакете com.sun.net.httpserver. Он позволяет быстро создать сервер без внешних зависимостей.

Основные классы:

HttpServer – создание сервера, привязка к порту.

HttpHandler – интерфейс с методом handle(HttpExchange exchange), который обрабатывает входящие запросы.

HttpExchange – предоставляет доступ к методу запроса, заголовкам, телу, а также позволяет сформировать ответ.

Пример простейшего сервера, обрабатывающего GET-запрос:

```
import com.sun.net.httpserver.HttpServer;
import java.io.*;
import java.net.InetSocketAddress;

class SimpleServer {
    public static void main(String[] args) throws IOException {
        HttpServer server = HttpServer.create(new InetSocketAddress( port: 8081),  backlog: 0);
        server.createContext( path: "/hello",  HttpExchange exchange -> {
            String response = "Hello, client!";
            exchange.sendResponseHeaders( rCode: 200, response.length());
            OutputStream os = exchange.getResponseBody();
            os.write(response.getBytes());
            os.close();
        });
        server.start();
        System.out.println("Сервер запущен на порту 8081");
    }
}
```

Практическая часть

Вариант 1

Разработать приложение-калькулятор для совершения простейших арифметических операций.

Исходные параметры (два числа) и тип операции (+, -, ×, ÷) вводятся на клиентской части и передаются серверу. Сервер вычисляет результат и возвращает его клиенту. При делении на ноль сервер возвращает сообщение об ошибке**.**

Вариант 2

Разработать приложение-чат.

На сервере хранится история сообщений. Клиент отправляет сообщение (текст + имя отправителя) на сервер, сервер добавляет его в общий список и возвращает все сообщения за последние 10 минут. Клиент отображает историю чата с метками времени.

Вариант 3

Разработать приложение-генератор случайных чисел.

На клиентской части вводится целое положительное число N и количество чисел K. Клиент передаёт N и K на сервер. Сервер генерирует массив из K случайных чисел в диапазоне от 1 до N и возвращает его клиенту.

Вариант 4

Разработать приложение-поисковик слов.

На сервере хранится определённый текст (например, несколько абзацев). На клиентской части вводится слово для поиска и передаётся серверу. Сервер находит все предложения, в которых встречается это слово (без учёта регистра), и возвращает их клиенту.

Вариант 5

Разработать приложение-счётчик букв.

На клиентской части вводится строка и передаётся серверу. Сервер подсчитывает количество гласных и согласных букв (русский или английский алфавит – на выбор) и возвращает клиенту JSON с результатами.

Вариант 6

Разработать приложение-определитель матрицы.

На клиентской части вводится квадратная матрица произвольного порядка (например, в виде строки или таблицы) и передаётся серверу. Сервер вычисляет определитель этой матрицы и возвращает результат клиенту.

Вариант 7

Разработать приложение для нахождения обратной матрицы размером 3×3.

Исходная матрица 3×3 вводится на клиентской части (9 чисел) и передаётся серверу. Сервер вычисляет обратную матрицу (если она существует) и возвращает её клиенту. Если матрица вырождена, возвращается сообщение об ошибке.

Вариант 8

Разработать приложение для определения счастливчика лотереи.

На сервере хранятся номера билетов (от 1 до 100). На каждом билете имеются 10 случайных чисел от 1 до 100. На клиентской части пользователь вводит 10 своих

чисел. Сервер определяет номер билета, в котором больше всего совпадений с введёнными числами, и возвращает этот номер (или несколько при равном совпадении).

Вариант 9

Разработать приложение для определения призовых мест на соревнованиях по прыжкам в длину.

На сервере хранятся фамилии участников и их идентификационные номера. На клиентской части вводятся результаты прыжков (по каждому идентификационному номеру). Сервер сортирует участников по дальности прыжка и возвращает фамилии спортсменов, занявших 1, 2 и 3 места.

Вариант 10

Разработать приложение для расчёта подоходного налога.

На клиентской части вводятся заработные платы сотрудников предприятия (список чисел) и передаются серверу. Сервер применяет ставки: для зарплаты < 100 000 руб. – 5%, от 100 000 до 500 000 руб. – 10%, > 500 000 руб. – 15%. Сервер возвращает сумму налога для каждого сотрудника и общую сумму налога.

Вариант 11

Разработать приложение по поиску квартиры для покупки.

Стоимости квартир и их адреса хранятся на сервере (предопределённый список). На клиентской части вводится предельная сумма для покупки. Сервер возвращает клиенту адреса всех квартир с ценой, не превышающей указанную сумму.

Вариант 12

Разработать приложение для поиска минимального и максимального элементов матрицы.

На клиентской части вводится матрица произвольного порядка (числа). Сервер определяет индекс строки, в которой находится минимальный элемент, и индекс столбца, в котором находится максимальный элемент. Результат (номера строки и столбца) возвращается клиенту.

Вариант 13

Разработать приложение для вычисления отношения диагоналей матрицы.

На клиентской части вводится квадратная матрица. Сервер вычисляет среднее арифметическое элементов на главной диагонали и среднее арифметическое элементов на побочной диагонали, затем находит отношение первого ко второму и возвращает результат клиенту.

Вариант 14

Разработать приложение «Расписание занятий» с возможностью редактирования.

На сервере хранится информация о расписании (день недели, время, предмет, преподаватель, аудитория). Клиентская часть имеет возможность просматривать расписание, добавлять новые записи, редактировать существующие и удалять их. Все изменения отправляются на сервер, который обновляет своё хранилище.

Вариант 15

Разработать приложение для расчёта себестоимости продукции.

На клиентской части пользователь вводит: основную заработную плату, дополнительную заработную плату, стоимость материалов, прочие затраты. Сервер рассчитывает полную себестоимость (сумма всех затрат) и, дополнительно, процент каждой статьи затрат в общей сумме. Результат возвращается клиенту в виде таблицы.

Вариант 16

Разработать приложение «Телефонный справочник» с поиском.

На сервере хранятся записи: фамилия, имя, телефон, адрес. Клиент может отправить запрос на поиск по фамилии (или части фамилии). Сервер возвращает все подходящие записи. Также клиент может добавить новую запись или удалить существующую по идентификатору.

Вариант 17

Разработать приложение «Конвертер валют» с актуальными курсами.

Сервер хранит текущие курсы валют (заданные в коде или обновляемые из внешнего источника). Клиент вводит сумму и выбирает исходную валюту (рубли, доллары, евро) и целевую валюту. Сервер выполняет конвертацию и возвращает результат. Дополнительно сервер может возвращать список доступных валют.

Вариант 18

Разработать приложение «Анализатор текста».

Клиент отправляет на сервер произвольный текст. Сервер выполняет статистический анализ: подсчитывает количество символов (с пробелами и без), количество слов, количество предложений, частоту встречаемости каждой буквы. Результат возвращается клиенту в виде JSON и отображается на экране.

III. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ВОПРОСОВ ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ (ЭКЗАМЕН)

1. Основные языки разработки под Android.
2. История развития и достоинства языка Kotlin.
3. Сборщики проектов. Gradle.
4. Основные элементы языка Kotlin: Типы данных. Переменные и функции.
5. Основные элементы языка Kotlin: Перечисления. Классы и свойства.
6. Основные элементы языка Kotlin: Управляющие конструкции.
7. Исключения.
8. Создание и использование коллекций.
9. Использование регулярных выражений.
10. Понятие анонимной функции и лямбда-выражения. Синтаксис лямбда выражений.
11. Лямбда-выражения и коллекции.
12. Перегрузка арифметических операторов.
13. Перегрузка логических операторов.
14. Перегрузка операторов для работы с коллекциями.
15. Понятие корутин. Преимущества использования корутин по сравнению с потоками. Жизненный цикл корутин.
16. Декларирование и применение аннотаций. Мета-аннотации. Классы в качестве параметров аннотации.
17. Классы для упрощения работы с базами данных.
18. Создание базы данных и таблиц.
19. Получение, модификация и запись данных в базу данных.
20. Структура Android проекта.
21. Структура Android проекта. Android Manifest.
22. Структура Android проекта. Ресурсы Android приложения.
23. Разработка UI Android приложения.
24. Меню Android приложения. Создание меню параметров.
25. Меню Android приложения. Создание контекстного меню.
26. Меню Android приложения. Создание всплывающего меню.
27. Меню Android приложения. Создание групп меню.
28. Жизненный цикл Activity.
29. Элементы экрана и их свойства.
30. Отладка Android приложений. LogCat.
31. Обработка исключений в Android.
32. База данных SQLite. SQLite OpenHelper.
33. База данных SQLite. Методы UPDATE и DELETE.
34. База данных SQLite. Метод QUERY.
35. База данных SQLite. Content provider (Контент-провайдер)
36. JSON в Android. Структура данных в JSON-формате.

IV. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

ОСНОВНАЯ ЛИТЕРАТУРА

1. Эккель, Б. Философия Java / Эккель Брюс. – 4-е полное изд. – Спб. : Питер, 2018 – 1168 с.: ил.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

2. Прата, С. Язык программирования C++. Лекции и упражнения : пер. с англ. / Стивен Прата. – 6-е изд. – М. : ООО “И.Д. Вильямс”, 2018. – 1248 с.

3. Маран, М. М. Программная инженерия : учебное пособие для вузов / М. М. Маран. – 3-е изд., стер. – СПб. : Лань, 2022. – 196 с. – Текст : электронный // Лань : электронно-библиотечная система. – URL: https://eJanbook.com/book/189470.

4. Черткова, Е. А. Программная инженерия. Визуальное моделирование программных систем : учебник для академического бакалавриата / Е. А. Черткова. – 2-е изд., испр. и доп. – М. : Юрайт, 2017. – 168 с. – (Бакалавр. Академический курс – Текст : электронный // Образовательная платформа Юрайт [сайт]. – URL: https://urait.ru/bcode/408215

5. Диков А.В. Клиентские технологии веб-программирования: JavaScript и DOM: учебное пособие, Издательство "Лань", 2020.- 124 с. – Текст : электронный // Лань : электронно-библиотечная система. – URL: https://eJanbook.com/book/126934.

6. Никсон, Р. Создаем динамические веб-сайты с помощью PHP, MySQL, JavaScript, CSS и HTML 5 / Робин Никсон ; [пер. с англ. Н.Вильчинского]. – 4-е изд. – СПб. [и др.] : Питер, 2018. – 766 с. – URL: https://ibooks.ru/bookshelf/352698.

7. Васильев, Н. П. Введение в гибридные технологии разработки мобильных приложений / Н. П. Васильев, А. М. Заяц. – 3-е изд., стер. – СПб. : Лань, 2022. – 160 с.. – Текст : электронный // Лань : электронно-библиотечная система. – URL: https://eJanbook.com/book/230387

8. Фримен, Э. Head First. Паттерны проектирования = Head First. Design Patterns / Эрик Фримен, Элизабет Робсон при участии Кэтти Сьерра и Берта Бейтса ; [пер. с англ. Е. Матвеева]. – Обновленное юбилейное изд. – СПб. [и др.] : Питер, 2019. – 651 с. – URL: https://ibooks.ru/bookshelf/377150.

9. Робсон Э. Изучаем HTML, XHTML и CSS. 2-е изд. / Э. Робсон, Э. Фримен. – СПб. : Питер, 2018. – 720 с. – ISBN 978-5-49600653-8. – URL: https://ibooks.ru/bookshelf/342692

10. Гриффитс Дэвид. Head First. Программирование для Android. 2-е изд. – СПб. : Питер, 2018. – 912 с. – URL: https://ibooks.ru/bookshelf/358143