Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет Компьютерного проектирования
Кафедра Инжинерной психологии и эргономики
Дисциплина Технологии программирования приложений

Лабораторная работа №1
«Activity: работа с элементами экрана, обработка нажатий кнопок»

Студент группы 310901 _____ Усов А.М.
(подпись)

Руководитель _____ Василькова А.Н.
(подпись)

Минск 2025

В рамках данной лабораторной работы разрабатывается приложение Calculator для платформы Android. Приложение представляет собой калькулятор с базовыми математическими операциями и функциями памяти.

Цель работы — формирование у студентов знаний и навыков работы с элементами экрана, обработки нажатий кнопок в Android-приложениях.

Скриншоты графических представлений приложения Calculator

0.1 Вертикальная ориентация экрана

Ниже представлен внешний вид приложения Calculator в вертикальной ориентации экрана в Android Studio.

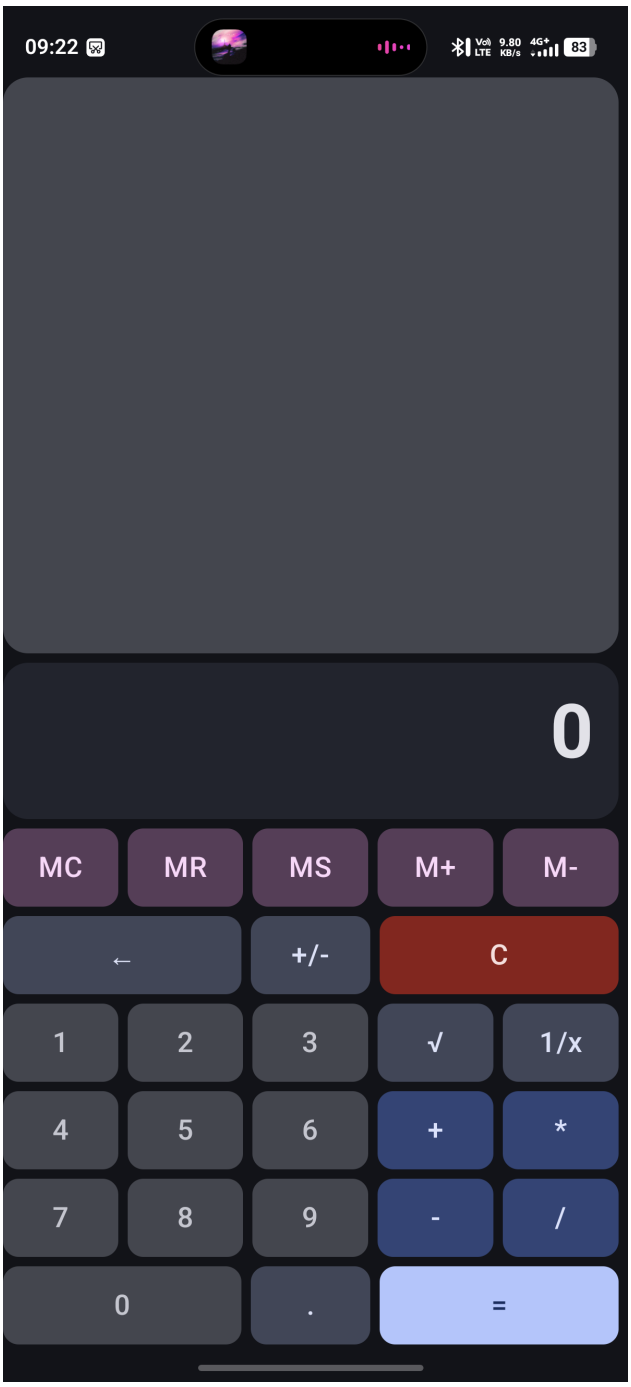

Рисунок 1 – Графическое представление приложения Calculator в вертикальной ориентации

На рисунке 1 показан интерфейс приложения Calculator в вертикальной ориентации экрана.

0.2 Горизонтальная ориентация экрана

Ниже представлен внешний вид приложения Calculator в горизонтальной ориентации экрана в Android Studio.

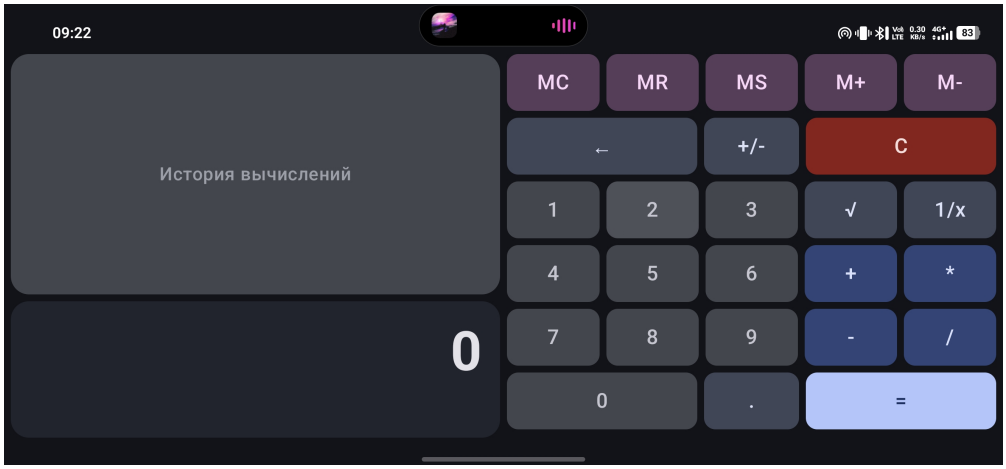

Рисунок 2 – Графическое представление приложения Calculator в горизонтальной ориентации

На рисунке 2 показан интерфейс приложения Calculator в горизонтальной ориентации экрана.

Код Activity приложения Calculator

1. Что такое нативное мобильное приложение, мобильная платформа?

Нативное мобильное приложение — это приложение, разработанное специально для конкретной мобильной платформы (Android, iOS) с использованием языков программирования и инструментов разработки, предоставляемых этой платформой. Нативные приложения имеют прямой доступ к API операционной системы и аппаратным возможностям устройства, что обеспечивает высокую производительность и полную интеграцию с платформой.

Мобильная платформа — это операционная система и набор инструментов разработки для мобильных устройств. Основные мобильные платформы: Android (Google), iOS (Apple), Windows Mobile (Microsoft).

2. Что собой представляет архитектура мобильной платформы Android?

Архитектура Android построена на основе Linux-ядра и состоит из нескольких уровней:

– **Linux Kernel** — нижний уровень, обеспечивающий работу драйверов устройств, управление памятью и процессами;
– **Hardware Abstraction Layer (HAL)** — уровень абстракции аппаратного обеспечения;
– **Native Libraries** — библиотеки на C/C++, включая SQLite, WebKit, OpenGL;
– **Android Runtime (ART)** — виртуальная машина для выполнения приложений;
– **Application Framework** — набор API для разработчиков (Activity Manager, Content Providers, View System и др.);
– **Applications** — уровень приложений пользователя.

3. Какие основные компоненты Android-приложения Вы знаете?

Основные компоненты Android-приложения:

– **Activity** — компонент, представляющий один экран с пользовательским интерфейсом;
– **Service** — компонент для выполнения длительных операций в фоновом режиме;
– **BroadcastReceiver** — компонент для обработки системных и пользовательских широковещательных сообщений;
– **ContentProvider** — компонент для управления доступом к структурированным данным приложения;
– **Fragment** — переиспользуемая часть пользовательского интерфейса внутри Activity.

4. Что собой представляет структура Android-проекта? Что содержит файл конфигурации AndroidManifest.xml, папка java, папка res?

Структура Android-проекта включает:

– **AndroidManifest.xml** — файл конфигурации, содержащий информацию о приложении: название, иконки, разрешения, объявление компонентов (Activity, Service и др.), минимальную и целевую версии Android SDK;
– **папка java/** — содержит исходный код приложения на Java/Kotlin, организованный по пакетам;
– **папка res/** — содержит ресурсы приложения: layouts (XML-разметки), drawable (изображения, иконки), values (строки, цвета, стили), mipmap (иконки приложения), menu (меню) и др.

5. Что такое графическое представление Activity?

Графическое представление Activity — это пользовательский

интерфейс, отображаемый на экране устройства. Оно создается с помощью XML-разметки (layout files) или декларативно с помощью Jetpack Compose. Графическое представление включает различные UI-элементы (View): кнопки, текстовые поля, изображения и др., организованные в контейнеры (Layout).

6. Что такое Layout? Какие существуют виды Layout?

Layout (макет) — это контейнер, определяющий структуру и расположение дочерних элементов интерфейса. Основные виды Layout:

- **LinearLayout** — располагает элементы линейно (вертикально или горизонтально);
- **RelativeLayout** — позволяет позиционировать элементы относительно друг друга;
- **ConstraintLayout** — гибкий layout с ограничениями для позиционирования;
- **FrameLayout** — простой контейнер, накладывающий элементы друг на друга;
- **GridLayout** — располагает элементы в виде сетки;
- **TableLayout** — организует элементы в виде таблицы.

В Jetpack Compose используются аналогичные компоновки: `Column`, `Row`, `Box`, `ConstraintLayout`.

7. Какие параметры (атрибуты) имеют View-элементы?

Основные атрибуты View-элементов:

- **android:id** — уникальный идентификатор элемента;
- **android:layout_width** и **android:layout_height** — размеры элемента (match_parent, wrap_content, конкретные значения);
- **android:layout_margin** — внешние отступы;
- **android:padding** — внутренние отступы;
- **android:gravity** — выравнивание содержимого внутри элемента;
- **android:text** — текст элемента;
- **android:textSize** — размер текста;
- **android:background** — фоновый цвет или изображение;
- **android:visibility** — видимость элемента;
- **android:enabled** — доступность элемента для взаимодействия.

8. Как создать Layout-файл для работы в горизонтальной ориентации экрана мобильного устройства? В каких случаях это необходимо?

Для создания Layout-файла для горизонтальной ориентации необходимо:

– создать папку `res/layout-land/` в проекте;
– разместить в ней XML-файл с тем же именем, что и для вертикальной ориентации (например, `activity_main.xml`);
– в файле использовать горизонтальную ориентацию (`android:orientation="horizontal"`) или другую компоновку, оптимизированную для широкого экрана.

Это необходимо в случаях, когда:

– требуется различное расположение элементов в разных ориентациях;
– нужно использовать дополнительное пространство экрана в горизонтальной ориентации;
– требуется оптимизация интерфейса под различные размеры экрана.

В Jetpack Compose адаптация к ориентации выполняется программно через проверку `LocalConfiguration.current` и условное построение UI.

9. Для чего нужны методы setContentView, findViewById?

– **setContentView()** — метод Activity, устанавливающий XML-разметку или Compose контент в качестве пользовательского интерфейса Activity. В традиционном подходе принимает ID ресурса layout: `setContentView(R.layout.activity_main)`. В Compose используется `setContent { ... }` для установки Compose UI.
– **findViewById()** — метод для поиска View-элемента по его ID в XML-разметке. Возвращает ссылку на элемент для дальнейшей работы с ним (изменение текста, установка обработчиков событий и т.д.). В Compose этот метод не используется, так как элементы создаются декларативно и доступны напрямую в коде.

10. Какие существуют способы обработки событий в Activity?

Основные способы обработки событий:

– **Анонимные классы** — создание объекта, реализующего интерфейс слушателя (например, `OnClickListener`) непосредственно в коде;
– **Именованные классы** — создание отдельного класса, реализующего интерфейс слушателя;
– **Реализация интерфейса в Activity** — Activity реализует интерфейс слушателя напрямую;
– **Lambda-выражения** — использование лямбда-функций для обработки событий (современный подход в Kotlin);

– **Ссылки на методы** — использование ссылок на методы через оператор `::`.

В Jetpack Compose обработка событий выполняется через функции-обработчики, передаваемые в качестве параметров в Composable функции (например, `onClick = { /* код */ }`).

Код Activity приложения Calculator

Ниже представлен код файла `MainActivity.kt`, содержащий логику обработки нажатий кнопок и вычислений с использованием Jetpack Compose:

```
package com.example.lt1

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            LT1Theme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    CalculatorScreen(
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}

@Composable
fun CalculatorScreen(modifier: Modifier = Modifier) {
    var displayValue by remember { mutableStateOf("0") }
    var previousValue by remember { mutableStateOf(0.0) }
    var operation by remember { mutableStateOf<Char?>(null) }
    var waitingForNewValue by remember { mutableStateOf(false) }
    var errorMessage by remember { mutableStateOf<String?>(null) }
    var memoryValue by remember { mutableStateOf(0.0) }
    var hasMemory by remember { mutableStateOf(false) }
    var history by remember { mutableStateOf<List<String>>(emptyList()) }

    val configuration = LocalConfiguration.current
    val isLandscape = configuration.screenWidthDp >
    ↪ configuration.screenHeightDp

    fun formatResult(result: Double): String {
        return if (result % 1.0 == 0.0) {
            result.toInt().toString()
        } else {
            String.format("%.10f", result).trimEnd('0').trimEnd('.')
        }
    }

    fun handleNumberInput(number: String) {
```

```
    if (errorMessage != null) {
        errorMessage = null
    }
    if (waitingForNewValue) {
        displayValue = number
        waitingForNewValue = false
    } else {
        displayValue = if (displayValue == "0" || displayValue == "Ошибка:
        ↪  деление на 0") {
            number
        } else {
            displayValue + number
        }
    }
}

fun handleOperation(op: Char) {
    if (errorMessage != null) {
        return
    }
    val currentValue = displayValue.toDoubleOrNull() ?: 0.0

    if (operation != null && !waitingForNewValue) {
        val result = when (operation) {
            '+' -> previousValue + currentValue
            '-' -> previousValue - currentValue
            '*' -> previousValue * currentValue
            '/' -> if (currentValue != 0.0) previousValue / currentValue
            ↪  else Double.NaN
            else -> currentValue
        }

        if (result.isNaN()) {
            displayValue = "Ошибка: деление на 0"
            errorMessage = "Ошибка: деление на 0"
            previousValue = 0.0
            operation = null
            waitingForNewValue = true
            return
        }

        displayValue = formatResult(result)
        previousValue = result
    } else {
        previousValue = currentValue
    }

    operation = op
    waitingForNewValue = true
}

fun handleEquals() {
    if (errorMessage != null || operation == null) {
        return
```

```
    }
    val currentValue = displayValue.toDoubleOrNull() ?: 0.0
    val operationSymbol = when (operation) {
        '+' -> "+"
        '-' -> "-"
        '*' -> "×"
        '/' -> "÷"
        else -> ""
    }
    val result = when (operation) {
        '+' -> previousValue + currentValue
        '-' -> previousValue - currentValue
        '*' -> previousValue * currentValue
        '/' -> if (currentValue != 0.0) previousValue / currentValue else
        ↪  Double.NaN
        else -> currentValue
    }

    if (result.isNaN()) {
        displayValue = "Ошибка: деление на 0"
        errorMessage = "Ошибка: деление на 0"
        history = history + "${formatResult(previousValue)}
        ↪  $operationSymbol ${formatResult(currentValue)} = Ошибка"
    } else {
        val formattedResult = formatResult(result)
        displayValue = formattedResult
        history = history + "${formatResult(previousValue)}
        ↪  $operationSymbol ${formatResult(currentValue)} =
        ↪  $formattedResult"
        if (history.size > 10) {
            history = history.takeLast(10)
        }
    }
    previousValue = 0.0
    operation = null
    waitingForNewValue = true
}

fun handleClear() {
    displayValue = "0"
    previousValue = 0.0
    operation = null
    waitingForNewValue = false
    errorMessage = null
}

fun handleBackspace() {
    if (errorMessage != null) {
        return
    }
    if (displayValue.length > 1 && displayValue != "0") {
        displayValue = displayValue.dropLast(1)
    } else {
        displayValue = "0"
```

```
    }
}

fun handleDecimalPoint() {
    if (errorMessage != null) {
        errorMessage = null
        displayValue = "0."
        waitingForNewValue = false
        return
    }
    if (waitingForNewValue) {
        displayValue = "0."
        waitingForNewValue = false
    } else if (!displayValue.contains(".")) {
        displayValue += "."
    }
}

fun handleSignChange() {
    if (errorMessage != null) {
        return
    }
    val currentValue = displayValue.toDoubleOrNull()
    if (currentValue != null) {
        displayValue = if (currentValue == 0.0) {
            "0"
        } else if (displayValue.startsWith("-")) {
            displayValue.substring(1)
        } else {
            "-$displayValue"
        }
    }
}

// Адаптивная компоновка в зависимости от ориентации
if (isLandscape) {
    Row(
        modifier = modifier
            .fillMaxSize()
            .background(MaterialTheme.colorScheme.background)
            .padding(6.dp),
        horizontalArrangement = Arrangement.spacedBy(6.dp)
    ) {
        Column(modifier = Modifier.weight(1f)) {
            // История вычислений
            Surface(
                modifier = Modifier
                    .fillMaxWidth()
                    .weight(1f),
                shape = RoundedCornerShape(16.dp),
                color = MaterialTheme.colorScheme.surfaceVariant
            ) {
                if (history.isEmpty()) {
                    Box(
```

```
                        modifier = Modifier.fillMaxSize(),
                        contentAlignment = Alignment.Center
                    ) {
                        Text("История вычислений", fontSize = 16.sp)
                    }
                } else {
                    LazyColumn(
                        modifier = Modifier.fillMaxSize().padding(8.dp),
                        reverseLayout = true
                    ) {
                        items(history.reversed()) { item ->
                            Text(item, fontSize = 14.sp, modifier =
                            ↪  Modifier.padding(vertical = 4.dp))
                        }
                    }
                }
            }
            Spacer(modifier = Modifier.height(6.dp))
            // Дисплей
            Surface(
                modifier = Modifier.fillMaxWidth().height(120.dp),
                shape = RoundedCornerShape(16.dp),
                color = MaterialTheme.colorScheme.surface
            ) {
                Box(modifier = Modifier.fillMaxSize().padding(16.dp)) {
                    Text(
                        text = displayValue,
                        fontSize = if (errorMessage != null) 24.sp else
                        ↪  48.sp,
                        fontWeight = FontWeight.Bold,
                        textAlign = TextAlign.End,
                        modifier = Modifier.align(Alignment.TopEnd)
                    )
                }
            }
        }
        // Кнопки калькулятора
        CalculatorNumpad(
            modifier = Modifier.weight(1f),
            onNumberClick = { handleNumberInput(it) },
            onOperationClick = { handleOperation(it) },
            onEqualsClick = { handleEquals() },
            onClearClick = { handleClear() },
            onBackspaceClick = { handleBackspace() },
            onDecimalClick = { handleDecimalPoint() },
            onSignClick = { handleSignChange() }
        )
    }
} else {
    Column(
        modifier = modifier
            .fillMaxSize()
            .background(MaterialTheme.colorScheme.background)
            .padding(6.dp)
```

```
        ) {
            // История вычислений
            Surface(
                modifier = Modifier.fillMaxWidth().weight(1f).heightIn(min =
                ↪  100.dp),
                shape = RoundedCornerShape(16.dp),
                color = MaterialTheme.colorScheme.surfaceVariant
            ) {
                LazyColumn(
                    modifier = Modifier.fillMaxSize().padding(8.dp),
                    reverseLayout = true
                ) {
                    items(history.reversed()) { item ->
                        Text(item, fontSize = 14.sp, modifier =
                        ↪  Modifier.padding(vertical = 4.dp))
                    }
                }
            }
            Spacer(modifier = Modifier.height(6.dp))
            // Дисплей и кнопки
            Column(modifier = Modifier.fillMaxWidth()) {
                Surface(
                    modifier = Modifier.fillMaxWidth().height(100.dp),
                    shape = RoundedCornerShape(16.dp),
                    color = MaterialTheme.colorScheme.surface
                ) {
                    Box(modifier = Modifier.fillMaxSize().padding(16.dp)) {
                        Text(
                            text = displayValue,
                            fontSize = if (errorMessage != null) 24.sp else
                            ↪  48.sp,
                            fontWeight = FontWeight.Bold,
                            textAlign = TextAlign.End,
                            modifier = Modifier.align(Alignment.TopEnd)
                        )
                    }
                }
                Spacer(modifier = Modifier.height(6.dp))
                CalculatorNumpad(
                    modifier = Modifier.fillMaxWidth(),
                    onNumberClick = { handleNumberInput(it) },
                    onOperationClick = { handleOperation(it) },
                    onEqualsClick = { handleEquals() },
                    onClearClick = { handleClear() },
                    onBackspaceClick = { handleBackspace() },
                    onDecimalClick = { handleDecimalPoint() },
                    onSignClick = { handleSignChange() }
                )
            }
        }
    }
}

@Composable
```

```
fun CalculatorNumpad(
    onNumberClick: (String) -> Unit,
    onOperationClick: (Char) -> Unit,
    onEqualsClick: () -> Unit,
    onClearClick: () -> Unit,
    onBackspaceClick: () -> Unit,
    onDecimalClick: () -> Unit,
    onSignClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier,
        verticalArrangement = Arrangement.spacedBy(6.dp)
    ) {
        // Строки кнопок с цифрами и операциями
        Row(modifier = Modifier.fillMaxWidth(), horizontalArrangement =
        ↪  Arrangement.spacedBy(6.dp)) {
            CalculatorButton(text = "1", modifier = Modifier.weight(1f)) {
            ↪  onNumberClick("1") }
            CalculatorButton(text = "2", modifier = Modifier.weight(1f)) {
            ↪  onNumberClick("2") }
            CalculatorButton(text = "3", modifier = Modifier.weight(1f)) {
            ↪  onNumberClick("3") }
            CalculatorButton(text = "+", modifier = Modifier.weight(1f),
            ↪  backgroundColor = MaterialTheme.colorScheme.primaryContainer)
            ↪  { onOperationClick('+') }
        }
        // ... остальные строки кнопок аналогично
    }
}

@Composable
fun CalculatorButton(
    text: String,
    modifier: Modifier = Modifier,
    backgroundColor: Color = MaterialTheme.colorScheme.surfaceVariant,
    textColor: Color = MaterialTheme.colorScheme.onSurfaceVariant,
    onClick: () -> Unit
) {
    Surface(
        modifier = modifier
            .height(50.dp)
            .clip(RoundedCornerShape(10.dp))
            .clickable { onClick() },
        color = backgroundColor
    ) {
        Box(modifier = Modifier.fillMaxSize(), contentAlignment =
        ↪  Alignment.Center) {
            Text(text, fontSize = 18.sp, fontWeight = FontWeight.Medium, color
            ↪  = textColor)
        }
    }
}
```

В классе `MainActivity` реализована логика калькулятора с использованием Jetpack Compose:

- использование `ComponentActivity` и функции `setContent()` для установки Compose UI;
- создание адаптивного интерфейса, который автоматически изменяется в зависимости от ориентации экрана;
- использование состояния через `remember` и `mutableStateOf` для управления данными калькулятора;
- обработка нажатий кнопок через функции-обработчики, передаваемые в Compose компоненты;
- реализация логики вычислений с поддержкой основных математических операций и функций памяти.

Интерфейс приложения создается декларативно с помощью Compose компонентов, что позволяет легко адаптировать его под различные ориентации экрана без использования XML-разметки.

Заключение

В ходе выполнения лабораторной работы было разработано Android-приложение Calculator, реализующее базовые математические операции. Приложение поддерживает вертикальную и горизонтальную ориентации экрана, используя Jetpack Compose для создания адаптивного интерфейса.

Были изучены и применены следующие технологии и подходы:

- создание пользовательского интерфейса с использованием Jetpack Compose;
- работа с Compose компонентами (`Column`, `Row`, `Surface`, `Text`);
- обработка событий нажатий кнопок через функции-обработчики в Compose;
- реализация логики калькулятора с поддержкой основных математических операций;
- использование состояния через `remember` и `mutableStateOf` для управления данными.

Приложение успешно обрабатывает нажатия кнопок и выполняет вычисления, демонстрируя навыки работы с элементами экрана и обработки пользовательского ввода в Android-приложениях с использованием современных технологий разработки.

Ответы на контрольные вопросы

1. Что такое нативное мобильное приложение, мобильная платформа?

Нативное мобильное приложение — это приложение, разработанное

специально для конкретной мобильной платформы (Android, iOS) с использованием языков программирования и инструментов разработки, предоставляемых этой платформой. Нативные приложения имеют прямой доступ к API операционной системы и аппаратным возможностям устройства, что обеспечивает высокую производительность и полную интеграцию с платформой.

Мобильная платформа — это операционная система и набор инструментов разработки для мобильных устройств. Основные мобильные платформы: Android (Google), iOS (Apple), Windows Mobile (Microsoft).

2. Что собой представляет архитектура мобильной платформы Android?

Архитектура Android построена на основе Linux-ядра и состоит из нескольких уровней:

- **Linux Kernel** — нижний уровень, обеспечивающий работу драйверов устройств, управление памятью и процессами;
- **Hardware Abstraction Layer (HAL)** — уровень абстракции аппаратного обеспечения;
- **Native Libraries** — библиотеки на C/C++, включая SQLite, WebKit, OpenGL;
- **Android Runtime (ART)** — виртуальная машина для выполнения приложений;
- **Application Framework** — набор API для разработчиков (Activity Manager, Content Providers, View System и др.);
- **Applications** — уровень приложений пользователя.

3. Какие основные компоненты Android-приложения Вы знаете?

Основные компоненты Android-приложения:

- **Activity** — компонент, представляющий один экран с пользовательским интерфейсом;
- **Service** — компонент для выполнения длительных операций в фоновом режиме;
- **BroadcastReceiver** — компонент для обработки системных и пользовательских широковещательных сообщений;
- **ContentProvider** — компонент для управления доступом к структурированным данным приложения;
- **Fragment** — переиспользуемая часть пользовательского интерфейса внутри Activity.

4. Что собой представляет структура Android-проекта? Что содержит файл конфигурации AndroidManifest.xml, папка java, папка

res?

Структура Android-проекта включает:

- **AndroidManifest.xml** — файл конфигурации, содержащий информацию о приложении: название, иконки, разрешения, объявление компонентов (Activity, Service и др.), минимальную и целевую версии Android SDK;
- **папка java/** — содержит исходный код приложения на Java/Kotlin, организованный по пакетам;
- **папка res/** — содержит ресурсы приложения: layouts (XML-разметки), drawable (изображения, иконки), values (строки, цвета, стили), mipmap (иконки приложения), menu (меню) и др.

5. Что такое графическое представление Activity?

Графическое представление Activity — это пользовательский интерфейс, отображаемый на экране устройства. Оно создается с помощью XML-разметки (layout files) или декларативно с помощью Jetpack Compose. Графическое представление включает различные UI-элементы (View): кнопки, текстовые поля, изображения и др., организованные в контейнеры (Layout).

6. Что такое Layout? Какие существуют виды Layout?

Layout (макет) — это контейнер, определяющий структуру и расположение дочерних элементов интерфейса. Основные виды Layout:

- **LinearLayout** — располагает элементы линейно (вертикально или горизонтально);
- **RelativeLayout** — позволяет позиционировать элементы относительно друг друга;
- **ConstraintLayout** — гибкий layout с ограничениями для позиционирования;
- **FrameLayout** — простой контейнер, накладывающий элементы друг на друга;
- **GridLayout** — располагает элементы в виде сетки;
- **TableLayout** — организует элементы в виде таблицы.

В Jetpack Compose используются аналогичные компоновки: `Column`, `Row`, `Box`, `ConstraintLayout`.

7. Какие параметры (атрибуты) имеют View-элементы?

Основные атрибуты View-элементов:

- **android:id** — уникальный идентификатор элемента;
- **android:layout_width** и **android:layout_height** — размеры элемента (match_parent, wrap_content, конкретные значения);

– **android:layout_margin** — внешние отступы;
– **android:padding** — внутренние отступы;
– **android:gravity** — выравнивание содержимого внутри элемента;
– **android:text** — текст элемента;
– **android:textSize** — размер текста;
– **android:background** — фоновый цвет или изображение;
– **android:visibility** — видимость элемента;
– **android:enabled** — доступность элемента для взаимодействия.

8. Как создать Layout-файл для работы в горизонтальной ориентации экрана мобильного устройства? В каких случаях это необходимо?

Для создания Layout-файла для горизонтальной ориентации необходимо:

– создать папку `res/layout-land/` в проекте;
– разместить в ней XML-файл с тем же именем, что и для вертикальной ориентации (например, `activity_main.xml`);
– в файле использовать горизонтальную ориентацию (`android:orientation="horizontal"`) или другую компоновку, оптимизированную для широкого экрана.

Это необходимо в случаях, когда:

– требуется различное расположение элементов в разных ориентациях;
– нужно использовать дополнительное пространство экрана в горизонтальной ориентации;
– требуется оптимизация интерфейса под различные размеры экрана.

В Jetpack Compose адаптация к ориентации выполняется программно через проверку `LocalConfiguration.current` и условное построение UI.

9. Для чего нужны методы setContentView, findViewById?

– **setContentView()** — метод Activity, устанавливающий XML-разметку или Compose контент в качестве пользовательского интерфейса Activity. В традиционном подходе принимает ID ресурса layout: `setContentView(R.layout.activity_main)`. В Compose используется `setContent { ... }` для установки Compose UI.
– **findViewById()** — метод для поиска View-элемента по его ID в XML-разметке. Возвращает ссылку на элемент для дальнейшей работы с ним (изменение текста, установка обработчиков событий и т.д.). В Compose этот метод не используется, так как элементы создаются

декларативно и доступны напрямую в коде.

10. Какие существуют способы обработки событий в Activity?

Основные способы обработки событий:

- **Анонимные классы** — создание объекта, реализующего интерфейс слушателя (например, `OnClickListener`) непосредственно в коде;
- **Именованные классы** — создание отдельного класса, реализующего интерфейс слушателя;
- **Реализация интерфейса в Activity** — Activity реализует интерфейс слушателя напрямую;
- **Lambda-выражения** — использование лямбда-функций для обработки событий (современный подход в Kotlin);
- **Ссылки на методы** — использование ссылок на методы через оператор `::`.

В Jetpack Compose обработка событий выполняется через функции-обработчики, передаваемые в качестве параметров в Composable функции (например, `onClick = { /* код */ }`).