Лабораторная работа № 8.
Построение UI интерфейса с помощью Compose.

1. Теоретический материал.

1.1. Язык Kotlin

1.1.1 Циклы

Циклы представляют вид управляющих конструкций, которые позволяют в зависимости от определенных условий выполнять некоторое действие множество раз.

Цикл while

```
var i = 10
while(i > 0){
    println(i*i)
    i--;
}
```

Цикл do-while

```
var i = -1
do{
    println(i*i)
    i--;
}
while(i > 0)
```

Цикл for с интервалами

```
    for(i in 1..15)
        println(i)
}
```

```
val range = 1..5
for(i in range){
    println("Элемент $i")
}
```

Цикл for – обход по коллекциям

List

```
val list = listOf("C", "A", "T")
for (letter in list) {
    print(letter)
}
```

Map

```
val capitals = mapOf(
    "China" to "Beijing ",
    "England" to "London",
    "Russia" to "Moscow"
)
```

```
for ((country, capital) in capitals) {
    println("The capital of $country is $capital")
}
```

Досрочный выход из цикла.

```
for(n in 1..5){
    if(n == 5) break;
    println(n * n)
}
```

Встроенная функция для повторения команд заданное число раз.

```
repeat(3){
    Text("Element")
}
```

1.1.2 Аннотации

Аннотации являются специальной формой синтаксических метаданных, добавленных в исходный код.

Пример аннотации перед функцией Hello. Аннотации начинаются с @.

```
@Preview(showSystemUi = true)
fun Hello() {
```

Чтобы объявить аннотацию, достаточно добавить модификатор annotation перед классом:

annotation class CustomAnnotation

(подробнее про аннотации: https://habr.com/ru/companies/surfstudio/articles/823906/)

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

Jetpack Compose предлагает декларативный API, который является более интуитивным.

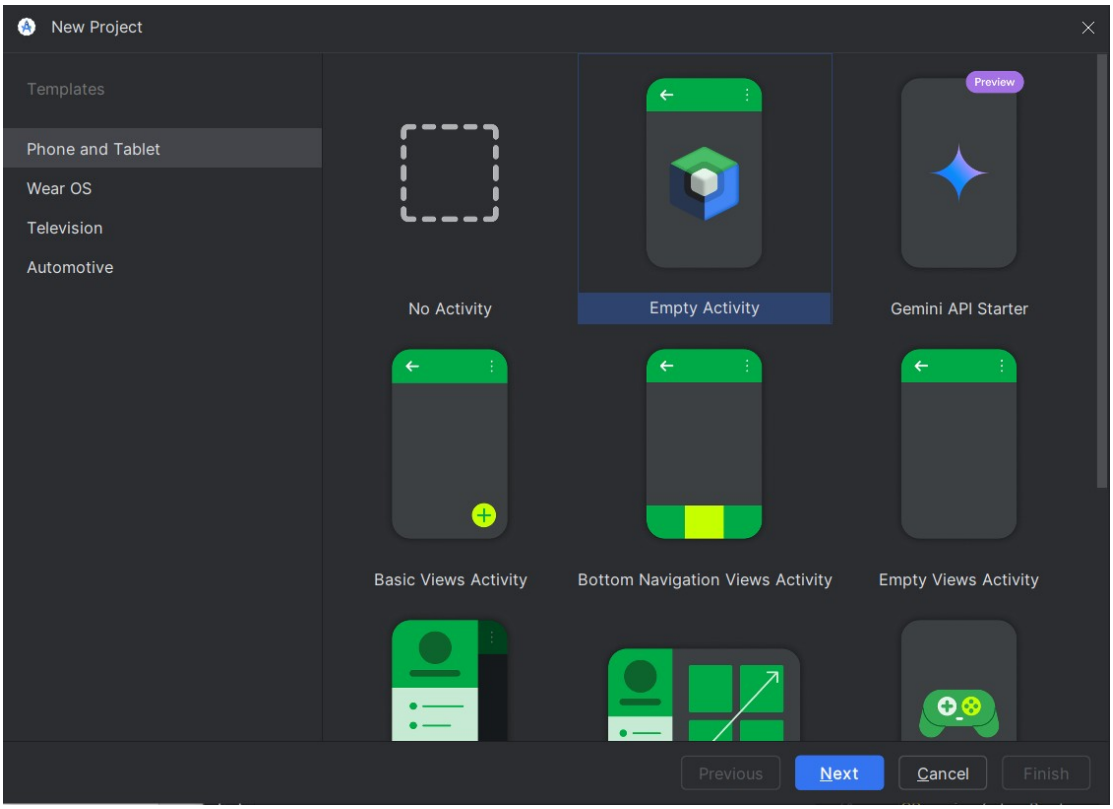
Jetpack Compose ориентирован на данные (data-driven-подход). В частности, Compose применяет концепцию состояния (state). Данные сохраняются как состояние, что гарантирует, что при любые изменения в данных автоматически отразяться на изменении пользовательского интерфейса. Соответственно не потребуется 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.

(подробнее: https://metanit.com/kotlin/jetpack/1.1.php)

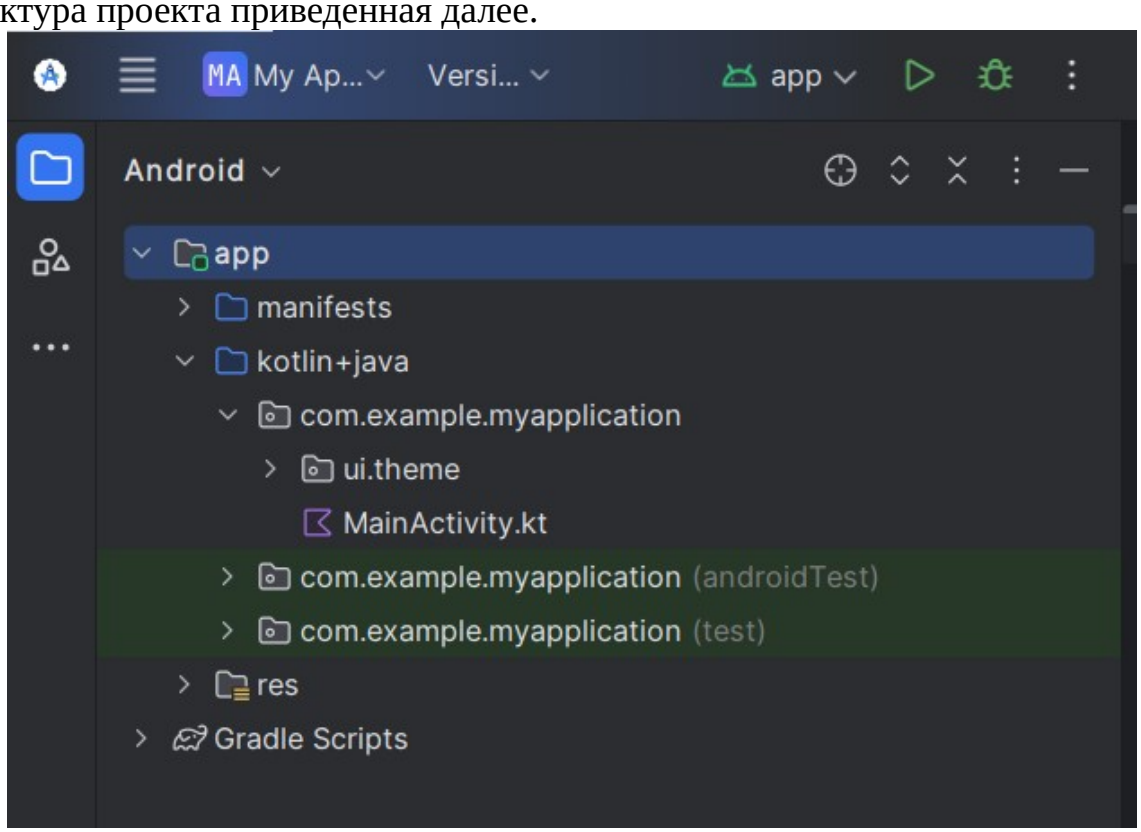
2. Ход выполнения проекта.

2.1 Создание проекта

Выбираем Empty Activity. Далее действуем по аналогии с Лабораторными работами №№ 1-2.



После создания проекта и выполнения всех скриптов Gradle, будет создана структура проекта приведенная далее.



В классе MainActivity.kt будет содержаться код по умолчанию.

Также в этой папке определен подкаталог **ui.theme**, который содержит ряд

файлов на языке Kotlin: **Color.kt, Theme.kt и Type.kt**, которые определяют ряд вспомогательных типов, функций, переменных, применяемых для создания графического интерфейса.

2.2. Создание простого UI

Сгенерированный Android Studio код удалим оставив только приведённый в рисунке № 1.

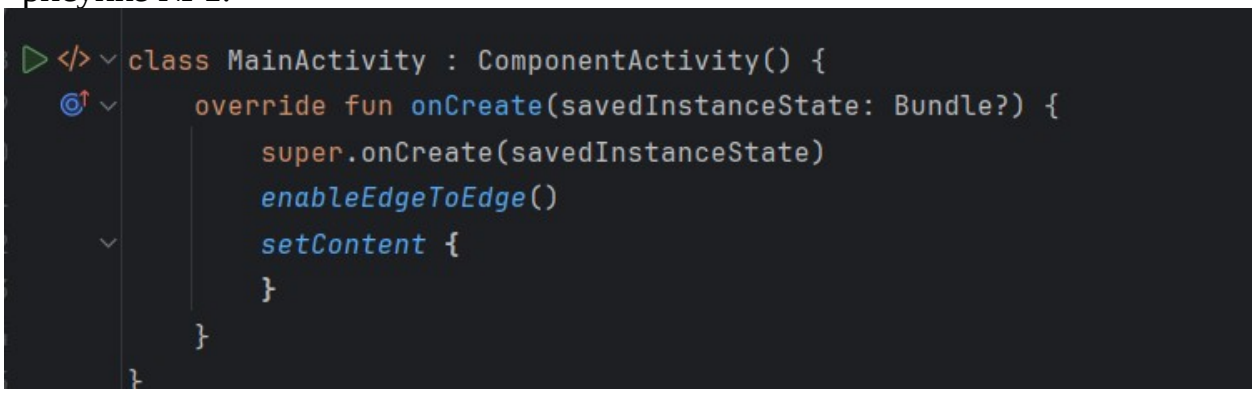


Рисунок № 1.

Корневой элемент устанавливается в методе **setContent**. **setContent**() является функцией расширения типа **ComponentActivity**. Функции расширения добавляют классу дополнительную функциональность без изменения его исходного кода. Это означает, что метод **setContent**() можно использовать для любого **ComponentActivity** или его подклассов, например **AppCompatActivity**. Вызов **setContent**() устанавливает функцию из параметра content в качестве корневого компонента, который выступает в качестве контейнера для всех остальных компонентов и в который можно добавить произвольное количество других компонентов.

Добавим первый UI компонент в приложение – **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.

sp (scale-independent pixels) — независимые от масштабирования пиксели. Допускают настройку размеров, производимую пользователем. При установке размера текста используются единицы измерения sp, так как они наиболее корректно отображают шрифты. В остальных случаях рекомендуется использовать dp.

Если какой-то элемент выделен красным, то можно поставить на него курсор и нажать сочетание клавиш «Alt+Enter».

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

После запуска должно вывестись надпись как на рисунке № 2.

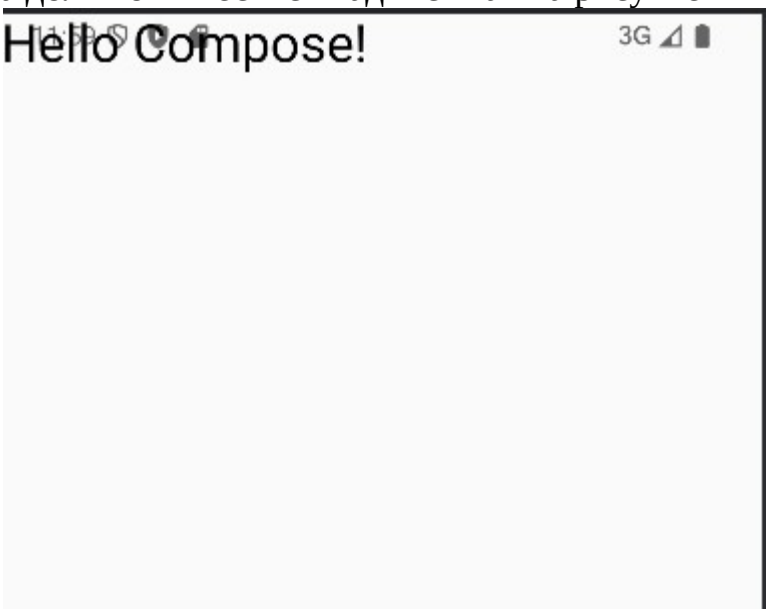


Рисунок № 2.

2.3. Создание компонентов Composable

Основой пользовательского интерфейса в Jetpack Compose являются компоненты, которые оформлены в виде функции с аннотацией **@Composable**. При вызове **@Composable**-компонента обычно передаются некоторые данные и набор свойств, которые определяют отображение и поведение определенной части пользовательского интерфейса.

Создадим **@Compose**-компонент:

```
@Composable
@Preview
fun Hello() {
    Text(text = "Hello Compose!",
        style = TextStyle(
            fontSize = 28.sp
        )
    )
}
```

И вызовем его в setContent вместо предыдущего кода:

```
setContent {
    Hello()
}
```

После запуска ни чего не поменяется.

С помощью параметров аннотации **@Preview** можно настраивать предварительный просмотр. Отображаться элемент будет в специальном окне, которое нужно выбрать, как показано на рисунке № 3.

Рисунок № 3.

Если надо отобразить остальную часть экрана, то можно установить опцию showSystemUi = true.

Добавим стилиации тексту. Изменения в предварительном просмотре отобразятся автоматически.

```
style = TextStyle(
    fontSize = 28.sp,
    color = Color.Cyan,
    fontWeight = FontWeight.Bold,
    textAlign = TextAlign.Center
)
```

2.4. Использование Modifier

Для настройки внешнего вида и стилизации большинства встроенных компонентов в Jetpack Compose применяются так называемые модификаторы. Модификаторы представляют функции, которые задают какой-то отдельный аспект для компонентов (или иными словами "модифицируют" внешний вид компонента), например, установка размеров компонента или его фонового цвета и т.д.

Большинство встронных компонентов поддерживают применение модификаторов через параметр modifier.

Тип Modifier предоставляет огромное количество встроенных функций-модификаторов, которые изменяют (модифицируют) отдельные аспекты компонентов.

К компоненту можно применять множество модификаторов по цепочке:

Modifier.модификатор1().модификатор2().модификатор3()...модификаторN()

Порядок следования модификаторов важен.

Передадим модификатор функции из пункта 2.3. UI будет выглядеть как на рисунке № 4. Теперь попробуйте убрать первый padding, а затем вернуть его и убрать второй. Какие изменения происходят и на, что это влияет?

```
@Composable
@Preview(showSystemUi = true)
fun Hello() {
    Text(text = "Hello Compose!",
        style = TextStyle(
            fontSize = 28.sp,
            color = Color.Cyan,
            fontWeight = FontWeight.Bold,
            textAlign = TextAlign.Center
        ),
        modifier = Modifier.padding(30.dp).back-
ground(Color.LightGray).padding(30.dp)
    )
}
```

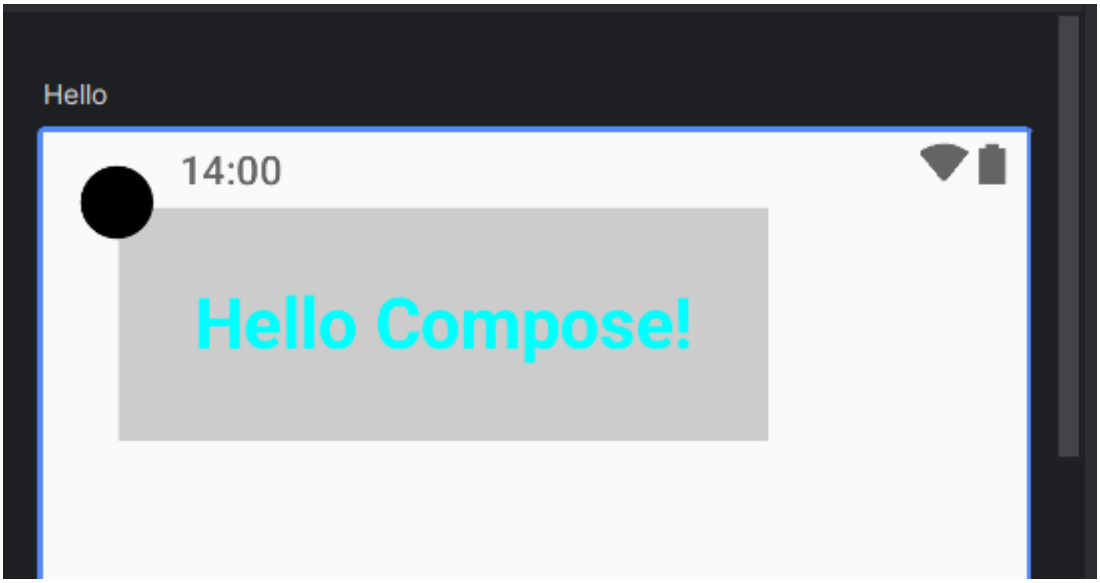


Рисунок № 4.

2.5. Использование Layout

При разработке приложений с помощью Compose мы можем как определять свои собственные компоненты, так и использовать встроенные.

Встроенные компоненты, которые по умолчанию входят в состав Compose, можно разделить на три категории: **Layout**, **Foundation** и **Material Design**.

В категорию Layout входят компоненты, которые определяют макет приложения, позволяют определить расположение других компонентов на экране:

- **Box**
- **BoxWithConstraints**
- **Column**
- **ConstraintLayout**
- **Row**

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

- **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

```
@Composable
```

```
@Preview(showSystemUi = true)
fun Hello() {
    val columnElementsStyle = TextStyle(
        fontSize = 28.sp,
        color = Color.Cyan,
        fontWeight = FontWeight.Bold,
        textAlign = TextAlign.Center
    )
    val columnElementModifier = Modifier.fillMaxWidth().height(40.dp).padding(5.dp)
        .clip(CircleShape)
        .background(color = Color.White)
        .border(width = 2.dp, color = Color.Yellow, shape = CircleShape)
    Column (modifier = Modifier.height(600.dp)
        .padding(10.dp)
        .background(Color.LightGray)
        .fillMaxWidth()
        .padding(30.dp),
        verticalArrangement = Arrangement.Top,
        horizontalAlignment = Alignment.CenterHorizontally) {
        Text("Element 1", style = columnElementsStyle, modifier = columnElementModifier)
        Text("Element 2", style = columnElementsStyle, modifier = columnElementModifier)
        Text("Element 3", style = columnElementsStyle, modifier = columnElementModifier)
        Text("Element 4", style = columnElementsStyle, modifier = columnElementModifier)
    }}
```

В результате получится окно как на рисунке № 5.

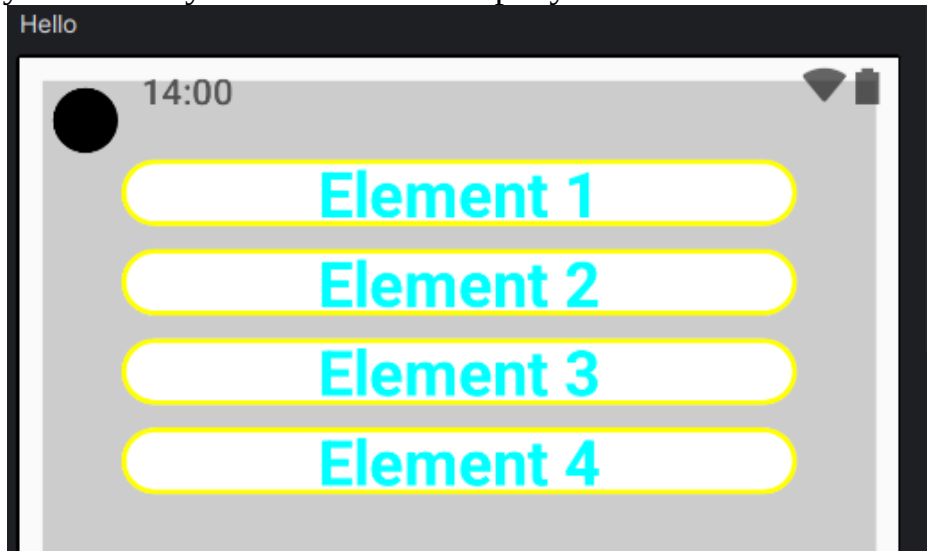


Рисунок № 5.

2.6. Добавляем обработку нажатий

Jetpack Compose предоставляет концепцию состояние или state. Состояние представляет некоторое значение, которое хранится в приложении и которое в процессе его работы может изменяться.

Измененяемое состояние в Jetpack Compose представлено объектом интерфейса **MutableState<T>**.

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()**.

Чтобы сохранить состояние используется функция remember. Эта функция применяется для сохранения некоторого объекта в памяти. Она может хранить как изменяемые (mutable), так и неизменяемые (immutable) объекты.

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**.

```
@Composable
@Preview(showSystemUi = true)
fun Hello() {
    val columnElementsStyle = TextStyle(
        fontSize = 28.sp,
        color = Color.Cyan,
        fontWeight = FontWeight.Bold,
        textAlign = TextAlign.Center
    )
    val columnElementModifier = Modifier.fill-
MaxWidth().height(40.dp).padding(5.dp)
        .clip(CircleShape)
        .background(color = Color.White)
        .border(width = 2.dp, color = Color.Yellow, shape
= CircleShape)
    Column (modifier = Modifier.height(600.dp)
        .padding(10.dp)
        .background(Color.LightGray)
        .fillMaxWidth()
        .padding(30.dp),
        verticalArrangement = Arrangement.Top,
        horizontalAlignment = Alignment.CenterHorizon-
tally) {
        for (i in 1..5) {
            val clicksState = remember{mutableStateOf(0)}
```

```
            val onClicksChange = { value : Int ->
                clicksState.value = value
            }
            Text("Element " + i.toString() + " clicked: ${clicksState.value}", style = columnElementsStyle,
                modifier = columnElementModifier.clickable {
                    onClicksChange(clicksState.value+1)
                }
            )
        }
    }
}
```

Результат работы вышеуказанного кода приведен на рисунке № 6.

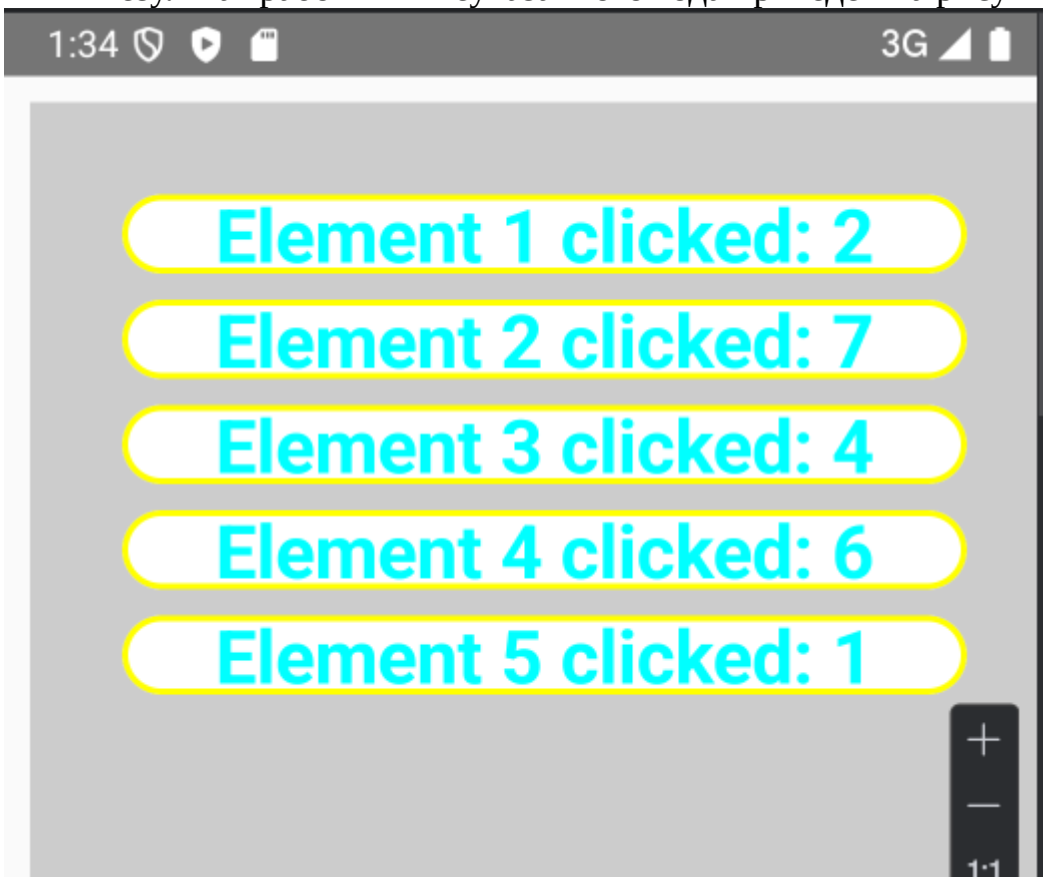


Рисунок № 6.

2.7. Работа с ресурсами.

Одним из наиболее применяемых типов ресурсов являются ресурсы строк. 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.

Каждый отдельный строковый ресурс определяется с помощью элемента string, а его атрибут name содержит название ресурса.

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

R.string.название_ресурса

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.

```
Text(
        text = stringResource(R.string.app_name),
        fontSize = 28.sp
    )
```

2.8. Lazy Column.

При большом количестве элементов использование стандартных контейнеров **Column** или **Row** может вызвать проблемы с производительностью, так как все вложенные элементы будут скомпонованы внутри контейнера независимо от того, видимы они или нет. Для более эффективной работы со вложенными компонентами Jetpack Compose предоставляет такие компоненты-контейнеры как **LazyColumn** и **LazyRow**. Элементы в них подгружаются по мере необходимости.

За отображение содержимого отвечают функции типа LazyListScope.() -> Unit:

- **LazyListScope.item()**: для добавления одного элемента
- **LazyListScope.items()**: для добавления нескольких элементов
- **LazyListScope.itemsIndexed()**: для добавления нескольких элементов с использованием индексов

Создадим список элементов в MainActivity

```
val elements = listOf("Element 0", "Element 1", "Element 2", "Element 3", "Element 4", "Element 5")
setContent {
    Hello(elements)
}
```

Перепишем функцию Hello с использованием LazyColumn

```
@Composable
fun Hello(elements: List<String>) {
    val columnElementsStyle = TextStyle(
        fontSize = 28.sp,
        color = Color.Cyan,
        fontWeight = FontWeight.Bold,
        textAlign = TextAlign.Center
    )
    val columnElementModifier = Modifier.fillMaxWidth().height(40.dp).padding(5.dp)
        .clip(CircleShape)
        .background(color = Color.White)
```

```
        .border(width = 2.dp, color = Color.Yellow, shape
= CircleShape)
    LazyColumn (modifier = Modifier.height(600.dp)
        .padding(10.dp)
        .background(Color.LightGray)
        .fillMaxWidth()
        .padding(30.dp),
        verticalArrangement = Arrangement.Top,
        horizontalAlignment = Alignment.CenterHorizon-
tally) {
            item { Text("Elements:", fontSize = 32.sp) }
            items(elements){ e -> val clicksState = re-
member { mutableStateOf(0) }
                val onClicksChange = { value: Int ->
                    clicksState.value = value
                }
                Text(e + " clicked: ${clicksState.-
value}",
                style = columnElementsStyle,
                modifier = columnElementModifier.click-
able {
                    onClicksChange(clicksState.value + 1)
                }
            )}
    }
}
```

На рисунке № 8 видно, что никаких изменений за исключением добавленного заголовка не наблюдается.

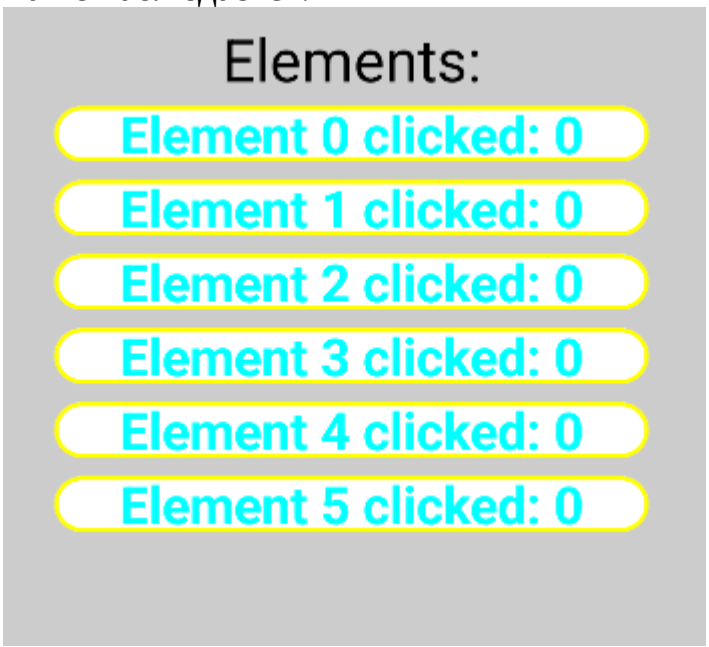


Рисунок № 8.

3. Задание для самостоятельной реализации

Реализовать как минимум окна «Регистрация» и «Администрирование» которые были созданы в лабораторных работах № № 2-6 с использованием фреймворка Compose.