Лабораторная работа № 3
Списки LazyColumn и LazyRow.
Работа с изображениями. Анимация

Контейнеры компоновки

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

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

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 определение их расположения;

Создание своего алгоритма вычисления размеров компонентов внутри компонента-контейнера, правильная 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.

Box

Представляет некоторую область экрана.

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

modifier: объект Modifier, который позволяет настроить внешний вид и поведение компонента с помощью модификаторов

contentAlignment: объект Alignment, 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

Пример простейшего элемента Box с вложенным элементом Text:

```
Box {
    Text("Hello BSUIR!", fontSize = 28.sp,)
}
```

Настройка contentAlignment 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

Если Box содержит несколько вложенных компонентов, то по умолчанию они будут накладываться друг на друга в порядке следования (последний компонент располагается поверх предыдущих)

Column

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

horizontalAlignment: объект Alignment.Horizontal, который устанавливает выравнивание компонента по горизонтали. По умолчанию имеет значение Alignment.Start

content: объект интерфейса BoxScope, который представляет вложенное содержимое.

```
setContent {
    Column {
        Text("Kotlin", fontSize = 28.sp)
        Text("Java", fontSize = 28.sp)
        Text("JavaScript", fontSize = 28.sp)
        Text("Python", fontSize = 28.sp)
    }
}
```

Если высота контейера Column больше суммы высот его вложенных компонентов, то для позиционирования этих компонентов может применяться параметр verticalArrangement, который может принимать следующие значения:

Arrangement.Center: расположение по центру

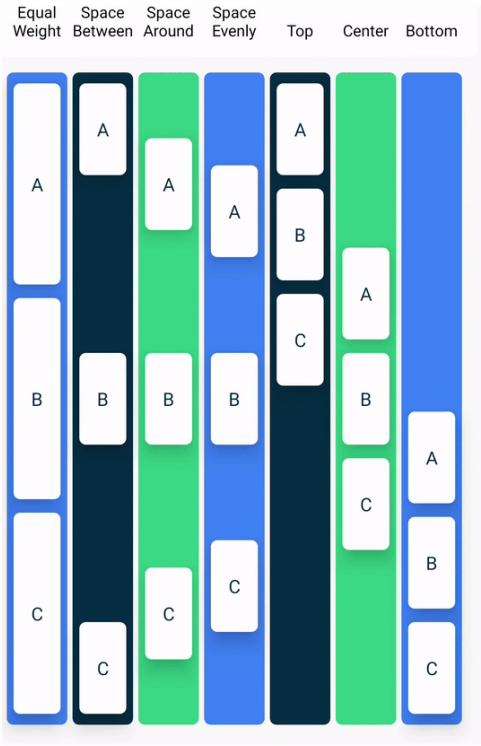
Arrangement.Bottom: расположение внизу

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 равны 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.



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

Модификаторы ColumnScope

Поскольку вложенные компоненты в Column 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 контейнере, который предполагает вертикальную прокрутку, то веса компонентов игнорируются, поскольку общее пространство по вертикали условно бесконечно.

```
Column {
  Box(modifier = Modifier.background(Color.Red).fillMaxWidth().weight(1f))
  Box(modifier = Modifier.background(Color.DarkGray).fillMaxWidth().weight(3f))
  Box(modifier = Modifier.background(Color.Blue).fillMaxWidth().weight(2f))
}
```

Row

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

verticalAlignment: объект Alignment.Vertical, который устанавливает выравнивание компонента по вертикали. По умолчанию имеет значение Alignment.Top

content: объект интерфейса RowScope, который представляет вложенное содержимое

Если ширина контейнера 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

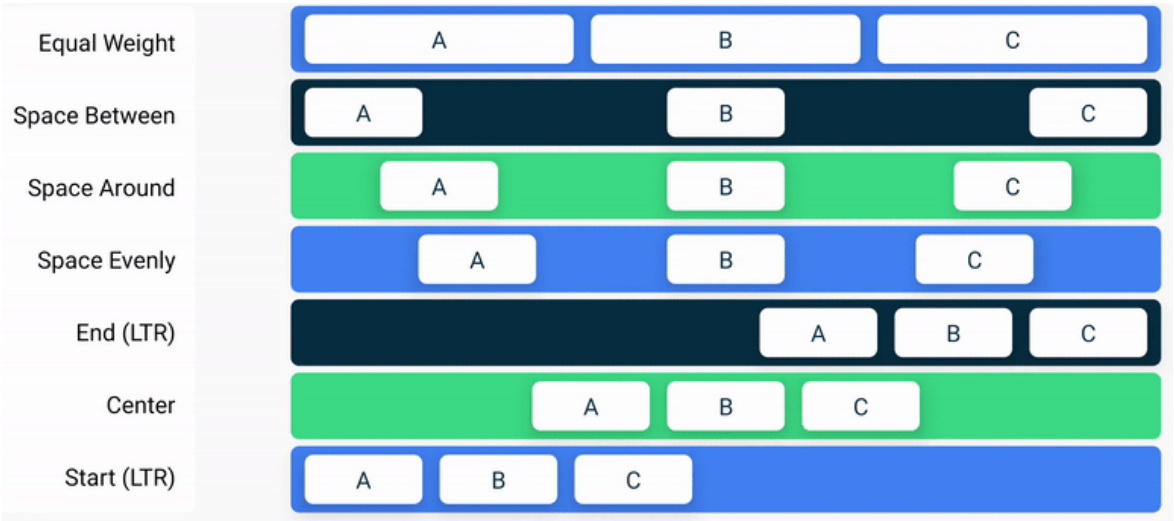
Arrangement.End: расположение в конце

Arrangement.Start: расположение в начале

Arrangement.SpaceAround: компоненты равномерно распределяются по всей ширине с равномерными отступами между элементами, при этом отступы между первым и последним элементами и границами контейнера равны 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, можно использовать одно из следующих значений:

Top: выравнивание по верху

CenterVertically: выравнивание по центру

Bottom: выравнивание по нижнему краю контейнера

Модификаторы RowScope

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

Вес (weight) позволяет назначить вложенным компонентам ширину в соответствии с некоторым коэффициентом. Для указания веса применяется модификатор RowScope.weight. Стоит учитывать, что если контейнер Row обеспечивает горизонтальную прокрутку или располагается в контейнере, который предполагает горизонтальную прокрутку, то веса компонентов игнорируются, поскольку общее пространство по горизонтали условно бесконечно.

Используя различные типы контейнеров, можно создавать более сложные композиции компоновок.

Surface

Компонент Surface является ключевым компонентом компоновки в Material Design, предоставляя для вложенного содержимого множество стилизаций по умолчанию. Он имеет несколько версий. Самая простая из них:

```
@Composable
@NonRestartableComposable
fun Surface(
    modifier: Modifier = Modifier,
    shape: Shape = RectangleShape,
    color: Color = MaterialTheme.colorScheme.surface,
    contentColor: Color = contentColorFor(color),
    tonalElevation: Dp = 0.dp,
```

```
        shadowElevation: Dp = 0.dp,
        border: BorderStroke? = null,
        content: @Composable () -> Unit
    ): Unit
```

Данная версия функции компонента принимает следующие параметры:
modifier: применяемые к контейнеру функции модификатора
shape: форма компонента в виде объекта Shape
color: фоновый цвет
contentColor: предпочитаемый цвет содержимого
tonalElevation: эффект анимации при нажатии
shadowElevation: высота тени
border: параметры границы в виде объекта BorderStroke

```
}
```

Если необходимо отобразить большое количество элементов (или список неизвестной длины), использование стандартных контейнеров как 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. Они компонуют и добавляют только те элементы, которые видны в окне просмотра компонента. При прокрутке в контейнер компонуются новые элементы, а старые удаляются. При обратной прокрутке происходит повторная компоновка старых элементов.

LazyColumn

LazyColumn создает список с вертикальной прокруткой и имеет следующие параметры:

```
@Composable
fun LazyColumn(
    modifier: Modifier = Modifier,
    state: LazyListState = rememberLazyListState(),
    contentPadding: PaddingValues = PaddingValues(0.dp),
    reverseLayout: Boolean = false,
    verticalArrangement: Arrangement.Vertical = if (!reverseLayout)
Arrangement.Top else Arrangement.Bottom,
    horizontalAlignment: Alignment.Horizontal = Alignment.Start,
    flingBehavior: FlingBehavior = ScrollableDefaults.flingBehavior(),
    userScrollEnabled: Boolean = true,
    content: LazyListScope.() -> Unit
): Unit
```

modifier: применяемые к контейнеру модификаторы
state: объект состояния LazyListState, применяемый для управления состоянием контейнера
contentPadding: отступы вокруг содержимого
reverseLayout: при значении true располагает элементы в обратном порядке
verticalArrangement: настройки расположения элементов по вертикали

horizontalAlignment: выравнивание элементов по горизонтали

flingBehavior: описывает поведение при таком типе прокрутки, когда пользователь быстро перетаскивает что-то и поднимает палец. Представляет объект FlingBehavior

userScrollEnabled: указывает, доступна ли прокрутка жестами либо через специальные инструменты управления доступом

content: устанавливает содержимое контейнера с помощью функции типа LazyListScope.() -> Unit.

Одно из отличий lazy-контейнеров, в частности, LazyColumn от других контейнеров, например, от Column состоит в принципе установки содержимого. За это отвечает функция типа LazyListScope.() -> Unit. Внутри блока этой функции можно использовать специальные методы для добавления других компонентов:

LazyListScope.item(): для добавления одного элемента

LazyListScope.items(): для добавления нескольких элементов

LazyListScope.itemsIndexed(): для добавления нескольких элементов с использованием индексов

Пример применения функции LazyListScope.item() для добавления в LazyColumn одного элемента

```
LazyColumn( Modifier.fillMaxSize()){
    item { Text("Hello BSUIR!", fontSize = 28.sp) }
}
```

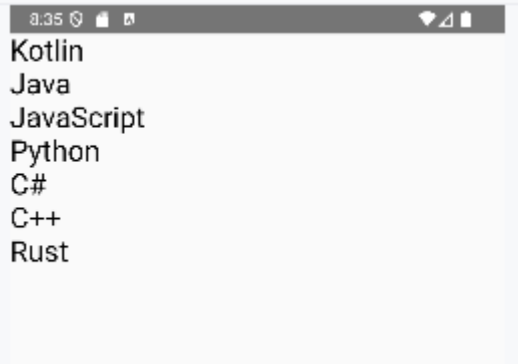
В данном случае добавляется компонент Text.

Функция LazyListScope.items() принимает список значений для каждого из которых создается элемент в LazyColumn:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.ui.Modifier
import androidx.compose.material3.Text
import androidx.compose.ui.unit.sp
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items


class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val langs = listOf("Kotlin", "Java", "JavaScript", "Python", "C#",
"C++", "Rust")
            LazyColumn(
                Modifier.fillMaxSize()
            ){
                items(langs){lang -> Text(lang, fontSize = 24.sp)}
            }
        }
```

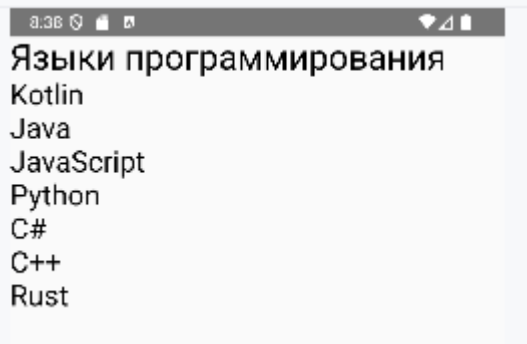
```
    }
}
```

Здесь для создания содержимого в LazyColumn применяется список langs. Функция items принимает этот список в качестве параметра и для каждого его элемента вызывает функцию @Composable() (LazyItemScope.(item: T) -> Unit). В примере в качестве такой функции выступает функция {lang -> Text(lang, fontSize = 24.sp)}, которая для каждого элемента списка создает компонент Text



Причем для создания элементов можно сочетать сразу несколько функций item()/items()

```
setContent {
    val langs = listOf("Kotlin", "Java", "JavaScript", "Python", "C#", "C++",
"Rust")
    LazyColumn(
        Modifier.fillMaxSize()
    ){
        item { Text("Языки программирования", fontSize = 29.sp) }
        items(langs){lang -> Text(lang, fontSize = 24.sp)}
    }
}
```

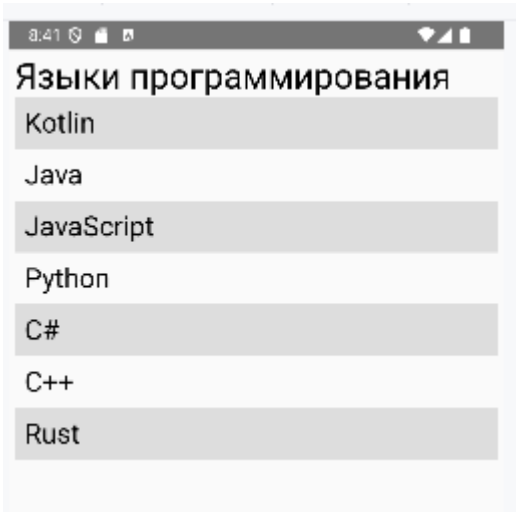


Еще одна функция itemsIndexed() аналогично items() принимает список/массив элементов, но при переборе позволяет получить их индекс.

Пример применения индекса для определения фонового цвета компонента

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
```

```
import androidx.compose.ui.Modifier
import androidx.compose.material3.Text
import androidx.compose.ui.unit.sp
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.itemsIndexed
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
import androidx.compose.foundation.layout.fillMaxWidth

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val langs = listOf("Kotlin", "Java", "JavaScript", "Python", "C#",
"C++", "Rust")
            LazyColumn(
                modifier = Modifier.fillMaxSize(),
                contentPadding = PaddingValues(5.dp)
            ){
                item { Text("Языки программирования", fontSize = 29.sp) }
                itemsIndexed(langs){index,lang -> Text(lang, fontSize = 23.sp,
                    modifier=Modifier.background(
                        if(index%2==0) Color(0xffdddddd) else Color.Transparent
                    ).padding(8.dp).fillMaxWidth())}
            }

        }
    }
}
```



LazyRow

Контейнер LazyRow создает список с горизонтальной прокруткой. Он имеет следующие параметры:

```
@Composable
fun LazyRow(
    modifier: Modifier = Modifier,
    state: LazyListState = rememberLazyListState(),
    contentPadding: PaddingValues = PaddingValues(0.dp),
    reverseLayout: Boolean = false,
    horizontalArrangement: Arrangement.Horizontal = if (!reverseLayout)
Arrangement.Start else Arrangement.End,
    verticalAlignment: Alignment.Vertical = Alignment.Top,
```

```
        flingBehavior: FlingBehavior = ScrollableDefaults.flingBehavior(),
        userScrollEnabled: Boolean = true,
        content: LazyListScope.() -> Unit
    ): Unit
```

modifier: применяемые к контейнеру модификаторы

state: объект состояния LazyListState, применяемый для управления состоянием контейнера

contentPadding: отступы вокруг содержимого

reverseLayout: при значении true располагает элементы в обратном порядке

horizontalArrangement: настройки расположения элементов по горизонтали

verticalAlignment: выравнивание элементов по вертикали

flingBehavior: описывает поведение при таком типе прокрутки, когда пользователь быстро перетаскивает что-то и поднимает палец. Представляет объект FlingBehavior

userScrollEnabled: указывает, доступна ли прокрутка жестами либо через специальные инструменты управления доступом

content: устанавливает содержимое контейнера с помощью функции типа LazyListScope.() -> Unit.

В целом LazyRow похож на LazyColumn, таким же образом устанавливает элементы, только располагает их по горизонтали.

Пример простейшего горизонтального списка

```
val langs = listOf("Kotlin", "Java", "JavaScript", "Python")
LazyRow {
    items(langs) {lang -> Text(lang)}

}
```

Также это может быть более сложное содержимое:

```
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.Box
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.lazy.LazyRow
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Text
import androidx.compose.ui.Alignment
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 {
            val langs = listOf(
```

```
                Language("Kotlin", 0xff16a085),
                Language("Java", 0xff2980b9),
                Language("JavaScript", 0xffd35400),
                Language("Python", 0xff2c3e50)
            )
            LazyRow {
                items(langs) { lang ->
                    Column(
                        Modifier.padding(8.dp),
                        horizontalAlignment = Alignment.CenterHorizontally
                    ) {

Box(Modifier.size(100.dp).background(Color(lang.hexColor)))
                        Text(lang.name, fontSize = 24.sp, modifier =
Modifier.padding(8.dp))
                    }
                }

            }
        }
    }
}
data class Language(val name:String, val hexColor: Long)
```

Программная прокрутка

Для программной прокрутки списков LazyColumn и LazyRow применяются функции объекта LazyListState, который можно получить с помощью вызова функции rememberLazyListState():

```
val listState = rememberLazyListState()
```

Затем этот объект передается параметру state:

```
LazyColumn(state = listState....
```

Для собственно прокрутки вызываются следующие методы LazyListState:

animateScrollToItem(index: Int) – плавная прокрутка к указанному элементу списка (где 0 — первый элемент)

scrollToItem(index: Int) – мгновенная прокрутка к указанному элементу списка (где 0 – первый элемент)

Это тоже suspend-функции, которые должны запускаться из корутин.

Пример

```
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.padding
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
import androidx.compose.material3.Text
import androidx.compose.ui.unit.sp
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.foundation.lazy.rememberLazyListState
```

```
import androidx.compose.foundation.lazy.LazyColumn
import kotlinx.coroutines.launch
import androidx.compose.foundation.clickable


class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent{
            val listState = rememberLazyListState()
            val coroutineScope = rememberCoroutineScope()
            LazyColumn(state=listState) {
                item{Text("В конец",
Modifier.padding(8.dp).background(Color.DarkGray).padding(5.dp).clickable {
                    coroutineScope.launch() {
                        listState.animateScrollToItem(19)
                    }
                }, fontSize = 28.sp, color = Color.White)
                }
                items(20){
                    Text("Item $it", Modifier.padding(8.dp), fontSize = 28.sp)
                }
            }

        }
    }

}
```

В данном случае создается аналогичный список с 20 компонентами Text плюс начальный Text, который выполняет роль кнопки перехода к 19-му элементу в списке.

Прикрепленные заголовки

LazyColumn и LazyRow позволяют использовать прикрепленные заголовки или sticky headers. То есть можно сгруппировать элементы списка под соответствующим заголовком. И прикрепленные заголовки остаются видимыми на экране во время прокрутки текущей группы. Как только группа прокручивается из поля зрения, ее место занимает заголовок следующей группы.

Для создания прикрепленных заголовков применяется функция LazyListScope.stickyHeader(). А содержимое списка должно храниться в массиве или списке, который трансформируется в группы с помощью функции Kotlin groupBy(). Функция groupBy() принимает лямбду-селектор, которая определяет, как данные должны быть сгруппированы. Этот селектор затем служит ключом для доступа к элементам каждой группы.

Пример 1 списка, содержащего модели смартфонов

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.ExperimentalFoundationApi
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.PaddingValues
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.lazy.LazyColumn
```

```
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.Text
import androidx.compose.ui.unit.sp
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 @OptIn(ExperimentalFoundationApi::class){
                // начальные данные
                val phones = listOf("Apple iPhone 15 Pro", "Realme 11 PRO",
"Google Pixel 5", "Samsung Galaxy S24 Ultra", "Google Pixel 6",
                    "Samsung Galaxy S21 FE", "Apple iPhone 15 Pro Max", "Xioami
Redmi Note 12", "Xiaomi Redmi 12",
                    "Apple iPhone 13", "Google Pixel 6", "Apple iPhone 14",
                    "Realme C30s", "Realme Note 50")
                // создаем группы
                val groups = phones.groupBy { it.substringBefore(" ") }
                LazyColumn(
                    contentPadding = PaddingValues(5.dp)
                ){
                    groups.forEach { (brand, models) ->
                        stickyHeader {
                            Text(
                                text = brand,
                                fontSize = 28.sp,
                                color = Color.White,
                                modifier =
Modifier.background(Color.Gray).padding(5.dp).fillMaxWidth()
                            )
                        }
                        items(models) { model ->
                            Text(model, Modifier.padding(5.dp), fontSize = 28.sp)
                        }
                    }
                }
        }
    }

}
```

На данный момент функциональность LazyListScope.stickyHeader() является экспериментальной, поэтому компонент, который использует данную функцию, надо предварять аннотацией @OptIn(ExperimentalFoundationApi::class)

```
setContent @OptIn(ExperimentalFoundationApi::class){
...
}
```

Сами выводимые данные представляют список phones, который содержит названия моделей смартфонов. С помощью функцию groupBy() элементы этого списка группируются по начальной подстроке:

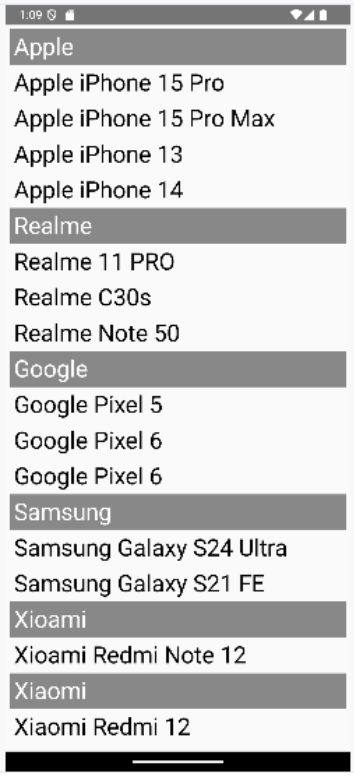
```
val groups = phones.groupBy { it.substringBefore(" ") }
```

Здесь it – это каждая строка из списка, соответственно выражение it.substringBefore(" ") получает в каждой строке списка подстроку, которая идет до пробела. Например, из строки "Apple iPhone 15 Pro" получается подстрока

"Apple". В пример условно эта подстрока представляет производителя смартфона. И функция groupBy группирует все данные по полученным подстрокам-производителям. В итоге получается словарь типа Map<String, List<String>>, где ключами являются подстроки-производители, а значениями – подсписок смартфонов конкретно данного производителя.

Далее идет перебор всего этого словаря:

```
groups.forEach { (brand, models) ->
    stickyHeader {
        Text(
            text = brand,
            fontSize = 28.sp,
            color = Color.White,
            modifier =
Modifier.background(Color.Gray).padding(5.dp).fillMaxWidth()
        )
    }
    items(models) { model ->
        Text(model, Modifier.padding(5.dp), fontSize = 28.sp)
    }
}
```



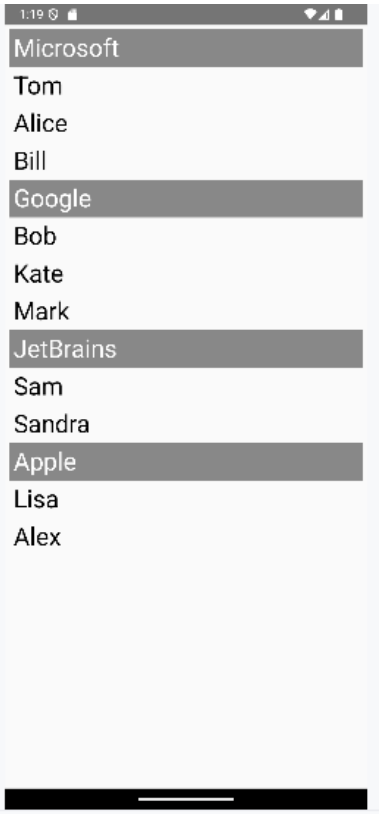
Для ключа каждого элемента словаря используется функция stickyHeader() – в ней создается компонент Text, который будет представлять собственно заголовок и будет отображать ключ – производителя смартфонов. Также для каждого элемента словаря применяется функция items(), которая отображает список смартфонов конкретного данного производителя. В итоге получится следующее приложение:

Пример 2

```
...
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent
@OptIn(ExperimentalFoundationApi::class){
            // начальные данные
            val people = listOf(
                Person("Tom", "Microsoft"), Person("Alice", "Microsoft"),
                Person("Bob", "Google"), Person("Sam", "JetBrains"),
                Person("Kate", "Google"), Person("Mark", "Google"),
                Person("Bill", "Microsoft"), Person("Sandra", "JetBrains"),
                Person("Lisa", "Apple"), Person("Alex", "Apple")
            )
            // создаем группы
            val groups = people.groupBy { it.company }
            LazyColumn(
                contentPadding = PaddingValues(5.dp)
            ){
                groups.forEach { (company, employees) ->
                    stickyHeader {
                        Text(
```

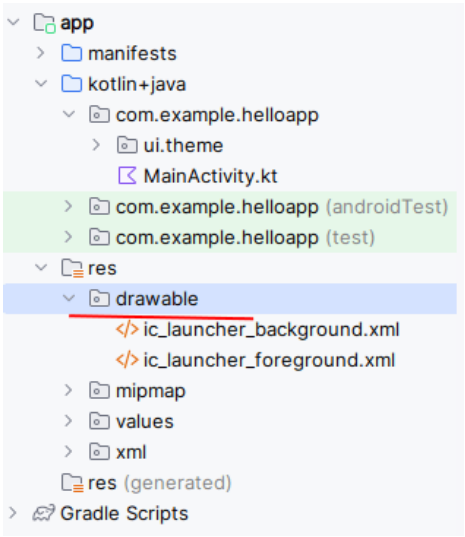
```
                                text = company,
                                fontSize = 28.sp,
                                color = Color.White,
                                modifier =
Modifier.background(Color.Gray).padding(5.dp).fillMaxWidth()
                                )
                        }
                        items(employees) { employee ->
                            Text(employee.name, Modifier.padding(5.dp), fontSize =
28.sp)
                        }
                    }
                }
            }
        }
}

data class Person(val name:String, val company: String)
```

Здесь данные представляют сложные данные объекта Person, который определяет два свойства: name и company. Группировка идет по свойству company:



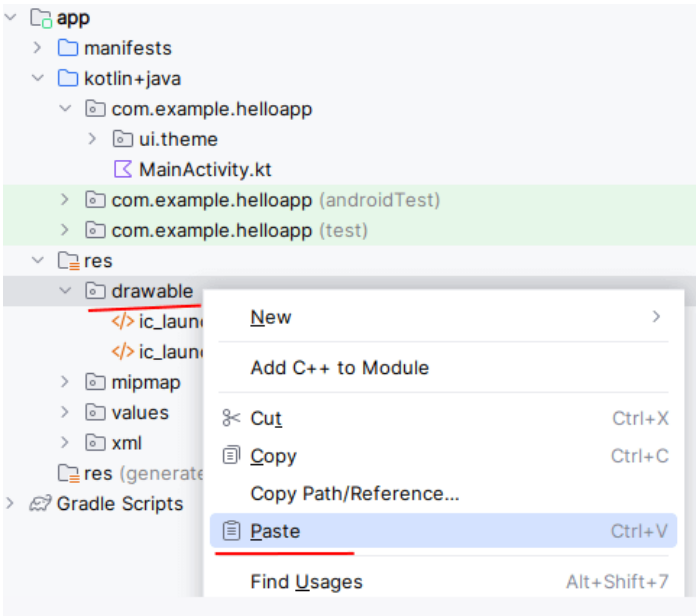
Ресурсы изображений

Изображение, которое необходимо отобразить в приложении, может располагаться в различных местах – это может быть сетевой проект, а 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.

Для добавления в эту папку какого-нибудь файла изображения необходимо скопировать добавляемый файл (с расширением png или jpg) и с помощью стандартной комбинации клавиш Ctrl+V добавить его в папку res/drawable. Или можно нажать на папку правой кнопкой мыши и в появившемся меню выбрать пункт Paste:



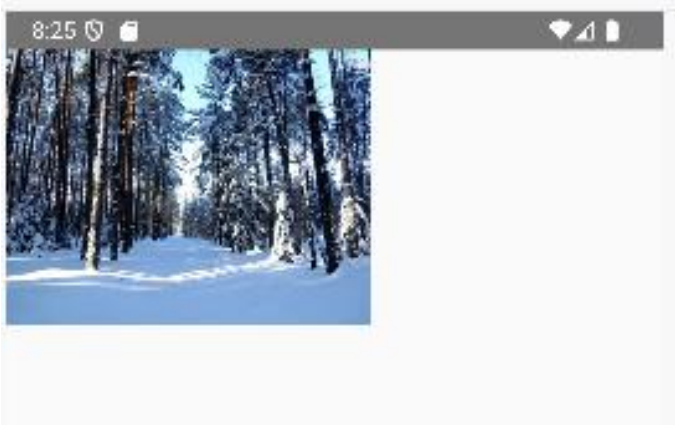
Далее при копировании файла будет предложено установить для него новое имя (по умолчанию подставляется текущее имя файла). Можно изменить название файла, а можно оставить как есть.

При добавлении графических файлов в папку drawable для каждого из них Android создает ресурс Drawable. После этого можно обратиться к ресурсу следующим образом в коде Kotlin: `R.drawable.имя_файла`

ImageBitmap

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.

```
Image(
      bitmap = ImageBitmap.imageResource(R.drawable.forest),
      contentDescription = "Зимний лес"
)
```



BitmapPainter

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:

```
Image(
     painter = BitmapPainter(ImageBitmap.imageResource(R.drawable.forest)),
     contentDescription = "Зимний лес"
     modifier = Modifier.fillMaxWidth()
)
```

В этом примере (R.drawable.forest) изображение отображается с помощью функции Image composable. Модификатор fillMaxWidth гарантирует, что изображение займет всю доступную ширину родительского макета.

Векторная графика

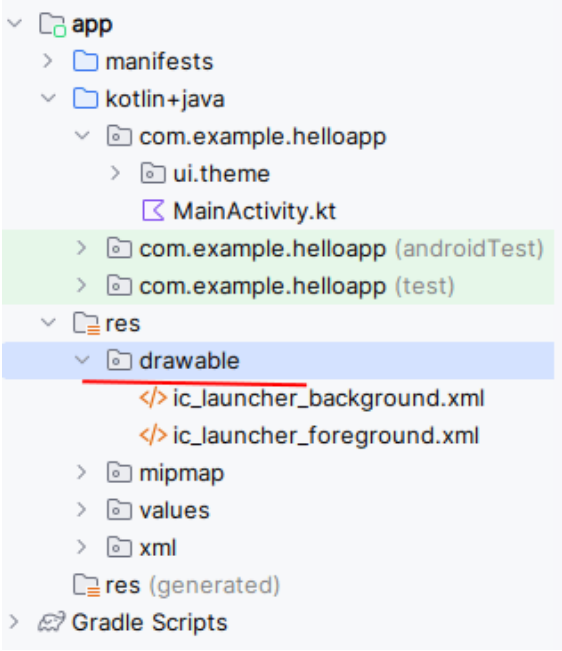
Для отображения векторной графики в компоненте Image применяется параметр imageVector, который представляет объект класса ImageVector.

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

```
Image(
      imageVector = Icons.Filled.Email,
      contentDescription = "Значок электронной почты",
        modifier = Modifier.size(200.dp, 150.dp)
 )
```

Ресурсы векторной графики

Также можно 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:



По умолчанию в этой папке уже имеется ресурсы векторной графики в виде файлов

ic_launcher_background.xml

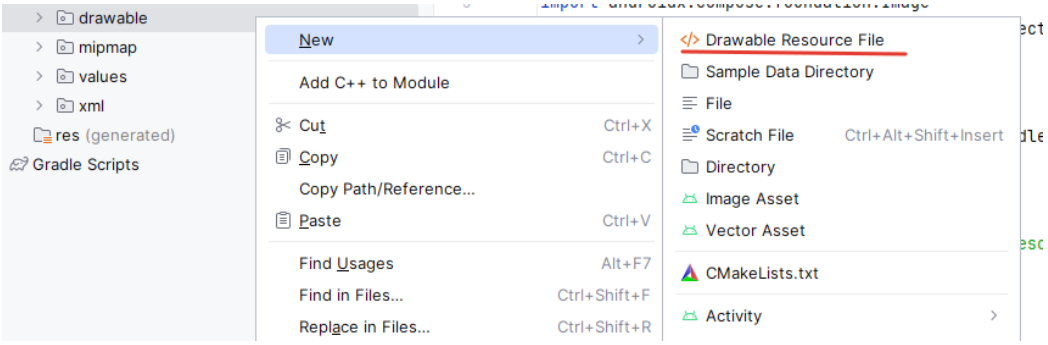
ic_launcher_foreground.xml,

которые применяются для создания иконок приложения.

Для загрузки ресурса векторной графики и создания из него объекта ImageVector применяется функция ImageVector.vectorResource(), в которую передается идентификатор загружаемого ресурса.

```
Image(
    imageVector = ImageVector.vectorResource(R.drawable.ic_launcher_background),
    contentDescription = "Android"
)
```

Для того, чтобы определить свой ресурс векторной график, в папку res/drawable необходимо добавим новый ресурс. По ПКМ на папке res/drawable из меню следует выбрать пункт New → Drawable Resource File:



Далее в появившемся окошке в поле File name следует ввести имя. Остальные настройки можно оставить по умолчанию. После нажатия ОК в папку res/drawable

будет добавлен файл в формате xml, где кодом можно описать графику (см. лекцию 9).

Изображение в овальной форме / кружочке

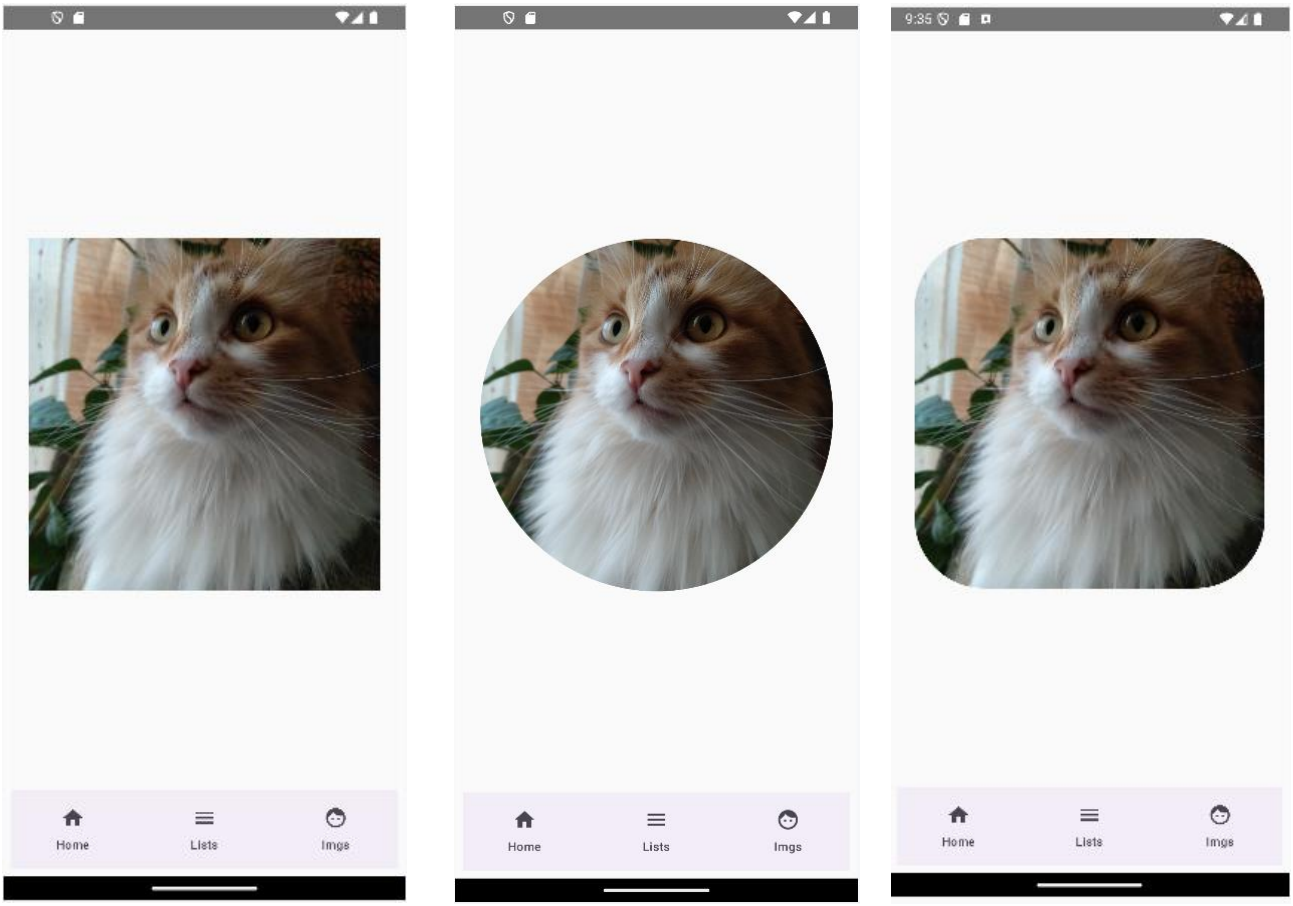
```
@Composable
fun CircleImage() {
    Column (modifier = Modifier.fillMaxSize(),
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Center)
    {
        Image(
            painter = painterResource(id = R.drawable.homka),
            contentScale = ContentScale.Crop,
            contentDescription = "My pet",
            modifier = Modifier.size(360.dp).clip(CircleShape)

        )
    }
}
```

Здесь модификатор clip придает форму круга и contentScale.Crop пропорционально масштабирует содержимое, чтобы заполнить границы.

Контейнер Column позволяет разместить изображение по центру.

Изображение в прямоугольнике с закругленными углами

```
modifier = Modifier.size(360.dp).clip(RoundedCornerShape(20))
```



Анимация

Для применения анимации в приложении Jetpack Compose предоставляет специальный API − Animation API. Этот API состоит из классов и функций, которые предоставляют широкие возможности по созданию анимации.

Compose Animation API предоставляет ряд анимаций состояния компонентов. В частности, это функции анимации для значений типов Bounds, Color, Dp, Float, Int, IntOffset, IntSize, Offset, Rect и Size. Подобные функции покрывают большинство потребностей в анимации компонентов.

Они используют одно и то же соглашение об именах. В частности, все они называются по шаблону:

```
animate*AsState
```

где символ * представляет тип состояния, которое анимируется. Например, если нужно анимировать изменение цвета (например, цвета фона компонента), то применяется функция animateColorAsState(). В реальности, функции передается целевое (конечное) значение, которое должно получить состояние. И функция анимирует переход от текущего значения к целевому значению.

Анимация Dp. animateDpAsState

Функция animateDpAsState() выполняет анимацию значений Dp, которые могут применяться для установки размеров, отступов и т.д. Она имеет следующие параметры:

```
@Composable
public fun animateDpAsState(
    targetValue: Dp,
    animationSpec: AnimationSpec<Dp> = dpDefaultSpring,
    label: String = "DpAnimation",
    finishedListener: ((Dp) -> Unit)? = null
): State<Dp>
```

targetValue: значение Dp, к которому надо выполнить переход

animationSpec: применяемая анимация в виде объекта AnimationSpec

label: название анимации

finishedListener: функция, которая выполняется при завершении анимации

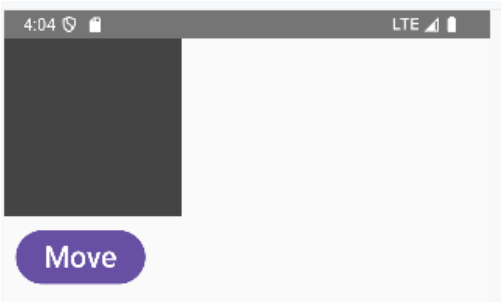
Обязательным параметром является только targetValue.

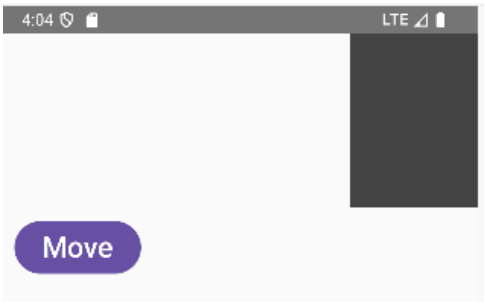
Пример анимации отступа слева, за счет чего возникает иллюзия движения объекта:

```
setContent {
  val startOffset = 0    // начальная позиция
  val endOffset = 300    // конечная позиция
  val boxWidth = 150       // ширина компонента

  var boxState by remember { mutableStateOf(startOffset)}
  val offset by animateDpAsState(targetValue = boxState.dp)
  Column(Modifier.fillMaxWidth()) {
      Box(Modifier.padding(start=offset)
                  .size(boxWidth.dp)
                  .background(Color.DarkGray))
      Button({boxState = if (boxState==startOffset){endOffset} else {startOffset}},
             Modifier.padding(10.dp)) {
```

```
                Text("Move", fontSize = 25.sp)
            }
        }
    }
}
```

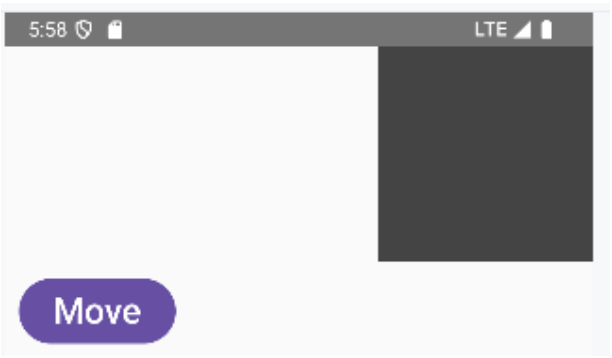
Здесь по нажатию на кнопку происходит перемещение компонента с помощью анимации значений Dp:





Чтобы box не заходил за экран, нужно для вычисления конечной позиции применить свойство LocalConfiguration.current.screenWidthDp, которое возвращает ширину экрана и, следовательно, переменную endoffset определить следующим образом (с учетом ширины компонента):

```
...
val boxWidth = 150      // ширина компонента
val endOffset = LocalConfiguration.current.screenWidthDp - boxWidth
...
```



Функция tween. Время и сглаживание анимации

Большинство функций, которые устанавливают анимацию, имеют параметр animationSpec, который представляет интерфейс AnimationSpec. С помощью этого параметра можно настроить продолжительность анимации, задержку, эффект отскок, повторение и замедление анимации и ряд других моментов.

Для создания объекта AnimationSpec Jetpack Compose предоставляет множество специальных функций и значений. Наиболее используемая из них – функция tween(). Она имеет следующую сигнатуру:

```
public fun <T> tween(
    durationMillis: Int = DefaultDurationMillis,
    delayMillis: Int = 0,
    easing: Easing = FastOutSlowInEasing
```

```
    ): TweenSpec<T>
```

Она принимает следующие параметры:
durationMillis: длительность анимации в миллисекундах
delayMillis: устанавливает начальную задержку
easing: позволяет ускорять и замедлять анимацию

Наиболее распространенный сценарий использования функции tween() – настройка времени анимации.

```
...
   val offset by animateDpAsState(
        targetValue = boxState.dp,
        animationSpec = tween(durationMillis = 5000) // анимация длится 5 секунд
    )
...
```

Параметр targetValue на основании boxState определяет позицию, к которой надо выполнить переход. А параметру animationSpec присваивается значение функции tween(), в которую передается число 5000. То есть анимация будет длиться 5 миллисекунд.

easing

Параметр easing в функции tween() позволяет ускорять и замедлять анимацию. В качестве значения можно передать одно из значений, определенных в пакете androidx.compose.animation.core.:

FastOutSlowInEasing
LinearOutSlowInEasing
FastOutLinearEasing
LinearEasing
CubicBezierEasing

```
val offset by animateDpAsState(
     targetValue = boxState.dp,
     animationSpec = tween(durationMillis = 5000, easing = FastOutSlowInEasing))
val offset1 by animateDpAsState(
     targetValue = boxState.dp,
    animationSpec = tween(durationMillis = 5000, easing = LinearOutSlowInEasing))
```

В каждом из примеров применяется анимация в течение 5 секунд, однако значение параметров easing отличается и это повлияет на принцип изменение отступа:

Если для параметра easing применяется значение CubicBezierEasing, то в конструктор надо передать координаты двух контрольных точек кривой Безье, на основании которой рассчитывается анимация:

```
import androidx.compose.animation.core.CubicBezierEasing
.....................
val offset by animateDpAsState(
    targetValue = boxState.dp,
    animationSpec = tween(durationMillis = 5000, easing = CubicBezierEasing(0f,
1f, 0.5f,1f))
)
```

Функция repeatable и повторение анимации

Для повторения анимации применяется класс RepeatableSpec (который реализует интерфейс AnimationSpec). Объект этого класса можно получить с помощью функции repeatable():

```
public fun <T> repeatable(
    iterations: Int,
    animation: DurationBasedAnimationSpec<T>,
    repeatMode: RepeatMode = RepeatMode.Restart,
    initialStartOffset: StartOffset = StartOffset(0)
): RepeatableSpec<T>
```

Она принимает следующие параметры:

iterations: количество повторений

animation: спецификация анимации. Можно использовать другие функции, которые возвращают DurationBasedAnimationSpec, например, функцию tween()

repeatMode: значение RepeatMode, которое указывает, должна ли анимация выполняться от начала до конца (значение RepeatMode.Restart) или должна выполняться в обратном порядке – от конца к началу (значение RepeatMode.Reverse)

initialStartOffset: смещение относительно начала

Для анимации с трехразовым повтором

```
val offset by animateDpAsState(
     targetValue = boxState.dp,
     animationSpec = repeatable(3, animation = tween(2000))
)
```

Параметр targetValue на основании boxState определяет позицию, к которой надо выполнить переход. А параметру animationSpec присваивается значение функции repeatable(). Первый аргумент функции – число 3 указывает на количество повторений. А параметр animation получает результат функции tween(), которая устанавливает время анимации – 2000 миллисекунд.

Параметр repeatMode при значении RepeatMode.Reverse позволяет задать воспроизведение анимации в обратном порядке:

```
...
     val offset by animateDpAsState(
      targetValue = boxState.dp,
      animationSpec = repeatable(3, animation = tween(2000), repeatMode =
RepeatMode.Reverse))
...
```

Функция spring и эффект отскока

Встроенная функция spring() позволяет добавить к анимации эффект отскока, подобно тому, как мяч ударяется об землю и после этого еще несколько раз подпрыгивает, делая несколько отскоков, пока совсем не остановит свое движение. Функция spring() имеет следующие параметры:

```
public fun <T> spring(
    dampingRatio: Float = Spring.DampingRatioNoBouncy,
```

```
        stiffness: Float = Spring.StiffnessMedium,
        visibilityThreshold: T? = null
    ): SpringSpec<T>
```

dampingRatio: степень затухания − определяет скорость, с которой затухает эффект подпрыгивания. Определяется как значение типа Float где 1.0 представляет отсутствие отскока, а 0.1 − самый высокий отскок. Вместо использования значений Float при настройке степени затухания также доступны следующие предопределенные константы:

DampingRatioHighBouncy
DampingRatioLowBouncy
DampingRatioMediumBouncy
DampingRatioNoBouncy

stiffness: жесткость при отскоке. При использовании меньшей жесткости диапазон движения при подпрыгивании будет больше. Для определения жесткости можно использовать ряд встроенных констант:

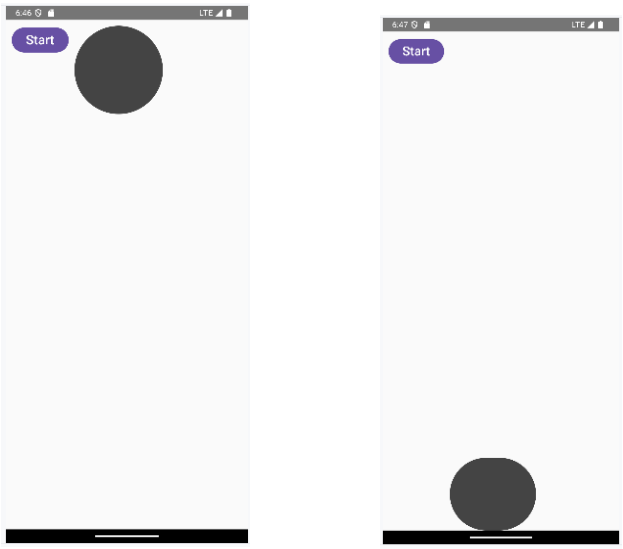
StiffnessHigh
StiffnessLow
StiffnessMedium
StiffnessMediumLow
StiffnessVeryLow

visibilityThreshold: предел видимости

```
...
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
    val circleHeight = 150      // высота (диаметр) круга
    val startOffset = 10       // начальный отступ
    val endOffset = LocalConfiguration.current.screenHeightDp - circleHeight
// предельная позиция
    var circleOffset by remember { mutableStateOf(startOffset)}
    val offset: Dp by animateDpAsState(
         targetValue = circleOffset.dp,
         animationSpec =if (circleOffset==endOffset) {
                           spring(dampingRatio = 0.3f)      // сильный отскок
                        }
                        else {
                           spring(dampingRatio = 1.0f)      // отсутствие отскока
                        }
    )
    Row(Modifier.fillMaxSize()) {
        Button({circleOffset =
          if (circleOffset==startOffset) endOffset else startOffset },
          Modifier.padding(10.dp)) {Text("Start", fontSize = 22.sp)}
        Box(Modifier.padding(top=offset)
                    .size(circleHeight.dp)
                    .clip(CircleShape)
                    .background(Color.DarkGray))
    }
  }
```

```
}}
```

Здесь определяется компонент Box в виде круга, и по нажатию на кнопку запускается анимация, в результате которой компонент движется вниз, пока не ударится о низ устройства.



Параметр animationSpec, который задает анимацию, использует условное выражение. Если значение, к которому надо перейти, является конечным – endOffset, то с помощью функции spring() в конце формируется отскок. Значение 0.3f можно характеризовать как сильный отскок. Если круг находится на конечной позиции (т.е. он упал вниз), и его движение с отскоком завершилось, то круг просто возвращается в начальную точку. В этом случае отскок не нужен, поэтому в функцию spring() передается значение 1.0f

В предыдущем примере в функцию spring() передавались числовые значения, но также можно передавать в нее и предустановленные константы:

```
import androidx.compose.animation.core.Spring.DampingRatioHighBouncy
import androidx.compose.animation.core.Spring.DampingRatioNoBouncy
.....................................

val offset: Dp by animateDpAsState(
    targetValue = circleOffset.dp,
    animationSpec =if (circleOffset==endOffset) {
        spring(dampingRatio = DampingRatioHighBouncy)      // сильный отскок
    } else {
        spring(dampingRatio = DampingRatioNoBouncy)      // отсутствие отскока
    }
)
```

Аналогичным образом можно использовать другие параметры функции spring().

Установка жесткости

```
val offset: Dp by animateDpAsState(
      targetValue = circleOffset.dp,
      animationSpec =if (circleOffset==endOffset) {
            spring(dampingRatio = DampingRatioMediumBouncy,
```

```
            stiffness = StiffnessVeryLow)   // жесткость StiffnessVeryLow
            } else {
              spring(dampingRatio = DampingRatioNoBouncy)
            }
    )
```

Функция keyframes и анимация по ключевым кадрам

Ключевые кадры (keyframes) позволяют применять различные значения длительности и замедления в определенных точках временной шкалы анимации. Ключевые кадры применяются к анимации через параметр animationSpec и определяются с помощью функции keyframes():

```
public fun <T> keyframes(
    init: KeyframesSpec.KeyframesSpecConfig<T>.() -> Unit
): KeyframesSpec<T>
```

Эта функция возвращает объект KeyframesSpec. Он принимает другую функцию, которая содержит данные о ключевых кадрах.

Определение анимации по ключевым кадрам содержит свойства durationMillis (общее время анимации) и delayMillis (задержка анимации – необязательное свойство), а также определения ключевых кадров. Каждый ключевой кадр содержит метку времени, которая указывает, какая часть общей анимации должна быть завершена в этот момент в зависимости от типа единицы состояния (например, Float, Dp, Int и т.д.). Эти временные метки создаются посредством вызовов функции at(). Например:

```
animationSpec = keyframes {
    durationMillis = 1000
    100.dp.at(10)
    110.dp.at(500)
    200.dp.at(700)
}
```

Здесь общее время анимации (durationMillis) 1000 миллисекунд. Для этой анимации задается три ключевых кадра. К примеру, первый кадр 100.dp.at(10) указывает, что смещение в 100.dp надо достигнуть через 10 миллисекунд. При 500 миллисекундах смещение должно составлять 110dp и, наконец, 200dp по истечении 700 миллисекунд. Это оставляет 300 миллисекунд для завершения оставшейся анимации.

```
    ...
val offset: Dp by animateDpAsState(
          targetValue = boxState.dp,
          animationSpec = keyframes {
              durationMillis = 1000
                if (boxState==endOffset) {
                        100.dp.at(100)
                        110.dp.at(500)
                        200.dp.at(800)
                    }
              }
           )
```

targetValue – значение, к которому надо перейти в процессе анимации. Параметру animationSpec передается результат функции keyframes(). В ней устанавливается общее время анимации – 1000 миллисекунд. И если надо перейти к конечному отступу (в boxState хранится endOffset), т.е. box находится вверху, то задаются три ключевых кадра. Если Box находится внизу, то ключевые кадры отсутствуют, а Box будет двигаться на начальную позицию равномерно.

В качестве альтернативы для установки ключевых кадров в функции keyframes() можно использовать другой синтаксис, где функция at() применяется аналогично операторам:

```
animationSpec = keyframes {
    durationMillis = 1000
    if (boxState==endOffset) {
        100.dp at 100
        110.dp at 500
        200.dp at 800
    }
}
```

Анимация цвета. animateColorAsState

Функция animateColorAsState() анимирует значение типа Color, то есть цвет. Она имеет следующие параметры:

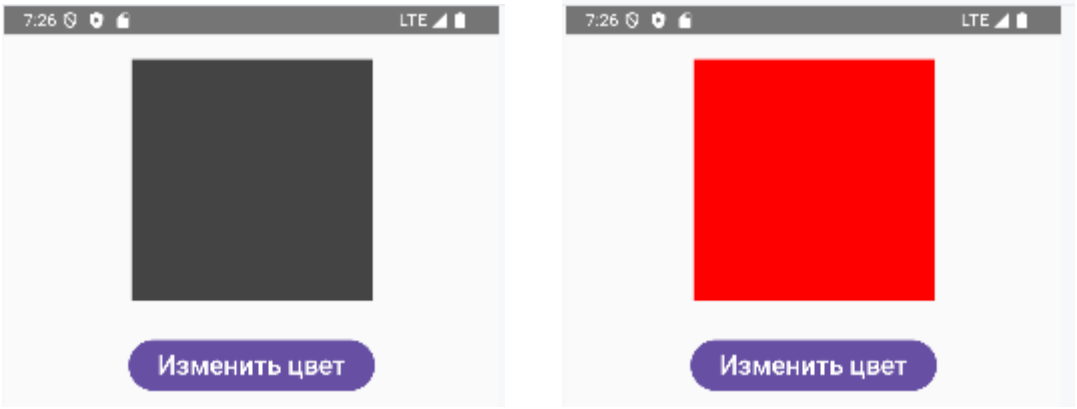
```
@Composable
public fun animateColorAsState(
    targetValue: Color,
    animationSpec: AnimationSpec<Color> = colorDefaultSpring,
    label: String = "ColorAnimation",
    finishedListener: ((Color) -> Unit)? = null
): State<Color>
```

targetColor: целевой цвет, к которому надо выполнить переход
animationSpec: применяемая анимация в виде объекта AnimationSpec
label: название анимации
finishedListener: функция, которая выполняется при завершении анимации

```
setContent {
     var colorState by remember { mutableStateOf(Color.DarkGray) }
      val animatedColor: Color by animateColorAsState(
           targetValue = colorState,
           animationSpec = tween(5000)
      )

      Column(Modifier.fillMaxWidth(),
             horizontalAlignment= Alignment.CenterHorizontally) {
            Box(Modifier.padding(20.dp)
                        .size(200.dp)
                        .background(animatedColor))
            Button(
                {colorState = if (colorState == Color.Red) {Color.DarkGray}
                              else {Color.Red}},
                 Modifier.padding(10.dp))
             {
               Text("Изменить цвет", fontSize = 22.sp)
             }
```

```
        }
}
```

Здесь происходит анимация цвета компонента Box от темно-серого к красному и наоборот.



Для этого вначале определяется состояние, которое будет хранить текущий цвет:

```
var colorState by remember { mutableStateOf(Color.DarkGray) }
```

По умолчанию это темно-серый цвет. Далее определяется сама анимация цвета:

```
val animatedColor: Color by animateColorAsState(
    targetValue = colorState,
    animationSpec = tween(5000)
)
```

Параметр targetValue указывает на значение, к которому надо перейти. И здесь просто передается значение colorState, то есть текущий цвет. Когда произойдет нажатие на кнопку, и изменится значение colorState, то targetValue получит новое значение. И функция выполнит переход к новому целевому цвету.

С помощью параметра animationSpec определяются настройки анимации. В данном случае с помощью функции tween() задается время выполнения анимации – 5000 миллисекунд. Т.е. переход от одного цвета к другому будет выполняться 5 секунд.

Функция animateColorAsState() возвращает объект типа State<Color>, а с помощью оператора by будет получен из него сам объект Color. Для демонстрации определен компонент Box, фон которого привязан к цвету, полученному из функции animateColorAsState

```
Box(Modifier.padding(20.dp).size(200.dp).background(animatedColor))
```

А с помощью кнопки переключается цвет в colorState:

```
Button(
    {colorState = if (colorState == Color.Red) {Color.DarkGray} else
```

```
{Color.Red}},
    Modifier.padding(10.dp)) {
    Text("Изменить цвет", fontSize = 22.sp)
}
```

Здесь спецификация анимации устанавливается с помощью функции tween(), но также можно использовать и другие функции, которые возвращают объект AnimationSpec.

```
...
 val animatedColor: Color by animateColorAsState(
     targetValue = colorState,
     animationSpec = keyframes {
          durationMillis = 3000
          Color.Blue at 500
          Color.Green at 1500
          Color.Yellow at 2500
      }
     )

...
```

В данном случае цвет компонента Box опять же изменяется с темно-серого на красный, однако в течение всего времени анимации (3 секунды) он также получает промежуточные цвета, которые описываются ключевыми кадрами анимации.

Таким образом, через 500 миллисекунд после начала анимации компонент Box получит цвет Color.Blue, через 1500 миллисекунд − Color.Green, и через 2500 миллисекунд после начала анимации − цвет Color.Yellow.

Анимация числовых значений и animateFloatAsState

Функция animateFloatAsState() применяется для анимации значений типа Float. Она имеет следующие параметры:

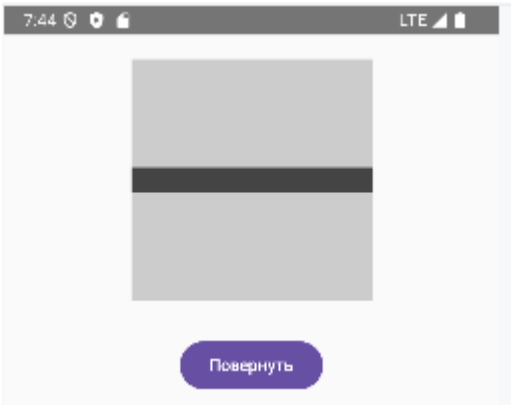
```
@Composable
public fun animateFloatAsState(
    targetValue: Float,
    animationSpec: AnimationSpec<Float> = defaultAnimation,
    visibilityThreshold: Float = 0.01f,
    label: String = "FloatAnimation",
    finishedListener: ((Float) -> Unit)? = null
): State<Float>
```

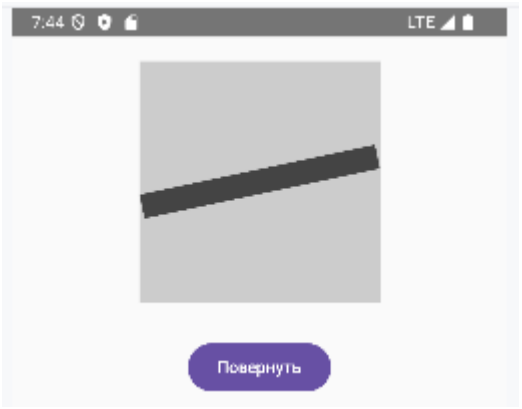
Здесь фактически те же самые параметры, что и в animateColorAsState() или animateDpAsState(), только вместо типа Color/Dp применяется тип Float. Например, параметр targetValue указывает на число, к которому надо выполнить переход.

```
setContent {
   var rotated by remember { mutableStateOf(false) }
   val angle by animateFloatAsState(
       targetValue = if (rotated) 360f else 0f,
       animationSpec = tween(4000)
   )
  Column(Modifier.fillMaxWidth(),
        horizontalAlignment = Alignment.CenterHorizontally) {
          Box(Modifier.padding(20.dp)
                      .size(200.dp)
```

```
                    .background(Color.LightGray)
                    .rotate(angle),
                Alignment.Center){
                Box(Modifier.size(width=200.dp,height=20.dp)
                        .background(Color.DarkGray))}
                Button({rotated = !rotated},
                    Modifier.padding(10.dp)) {
                  Text(text = "Повернуть")
                }
        }
}
```

Здесь суть приложения заключается в повороте компонента с помощью анимации Float:





Для поворота сначала определяется состояние:

```
var rotated by remember { mutableStateOf(false) }
```

Это значение указывает, в какую сторону вращать компонент. Далее на основе этого состояния определяется анимация:

```
val angle by animateFloatAsState(
    targetValue = if (rotated) 360f else 0f,
    animationSpec = tween(4000)
)
```

Если rotated равно true, то целевое значение анимации устанавливается на 360 градусов, в противном случае оно устанавливается на 0. Результат анимации сохраняется в переменную angle.

Затем значение angle передается в модификатор rotate(), который принимает угол поворота:

```
Box(Modifier.padding(20.dp).size(200.dp).background(Color.LightGray).rotate(angle
), Alignment.Center){
    Box(Modifier.size(width=200.dp,height=20.dp).background(Color.DarkGray))
}
```

И для запуска анимации определяется кнопка, по нажатию на которую переключается значение rotated, что, в свою очередь, приведет к изменению угла вращения и анимации:

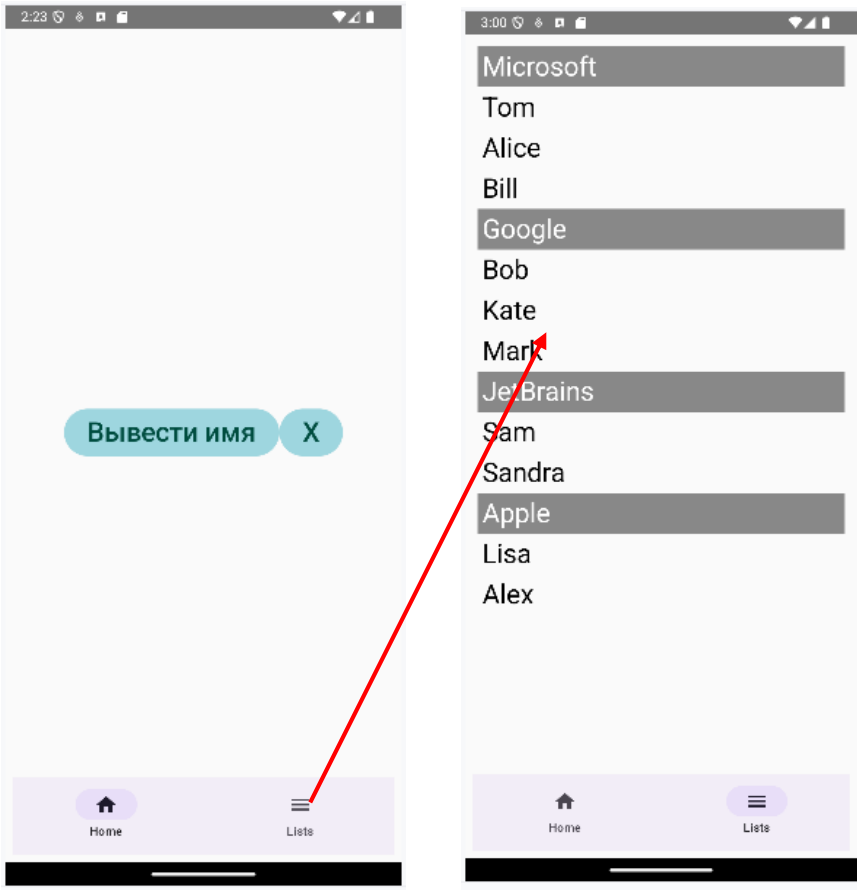
```
Button({rotated = !rotated},
```

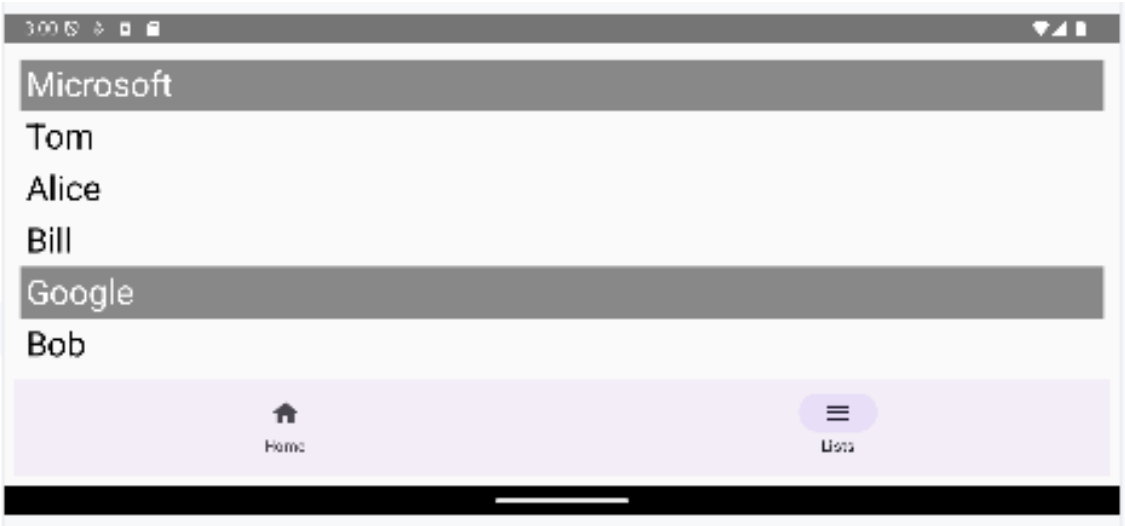
Задание на лабораторную работу

1. В приложении, которое было создано во второй лабораторной работе, на экране № 2 (т.е. на экране, на который осуществляется переход с главного экрана) вместо текста Lists создать списки с прикрепленными заголовками (по вариантам). Должна работать программная прокрутка, изменение ориентации устройства.

Варианты заданий

1. Student: Группа, ФИО (заголовок – Группа)
2. Customer: Банк, ФИО (заголовок – Банк)
3. Patient: Диагноз, ФИО (заголовок – Диагноз)
4. Abiturient: Факультет, ФИО (заголовок – Факультет)
5. Book: Автор, название (заголовок – Автор)
6. House: Адрес, № квартиры (заголовок – Адрес)
7. Phone: Страна, провайдер (заголовок – Провайдер)
8. Car: Марка, Модель (заголовок – Марка)
10. Train: Пункт назначения, номер поезда (заголовок – Пункт назначения)
11. Bus: Номер автобуса, ФИО пассажира (заголовок – Номер автобуса)
12. Airlines: Пункт назначения, дата и время вылета (заголовок – Пункт назначения)
13. Country: Страна, город (заголовок – Страна)
14. Animals: Класс, наименование (заголовок – Класс)
15. Plants: Класс, наименование (заголовок – Класс)

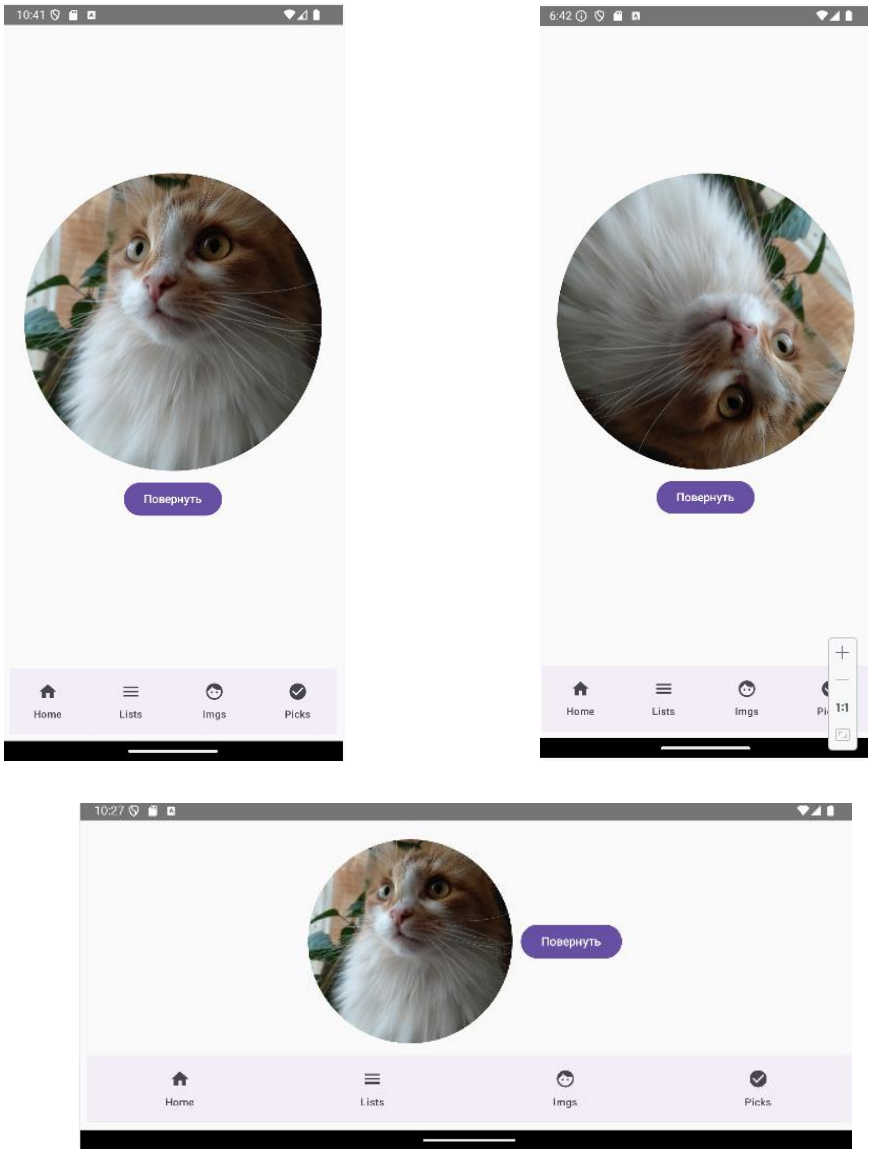




2. Дополните навигацию для перехода на экран № 3, на котором разместите изображение из ресурсов drawable.

3. Создайте какую-нибудь анимацию с изображением. Обработайте изменение ориентации экрана.

Например



Обработать изменение ориентации экрана можно, например, так

```
val configuration = LocalConfiguration.current
if (configuration.orientation == Configuration.ORIENTATION_LANDSCAPE) {
 …     //альбомная ориентация
} else {
…     //портретная ориентация
}
```