## Часть I – ViewModel

Фреймворк Android управляет жизненным циклом контроллеров пользовательского интерфейса, таких как активности и фрагменты. Фреймворк может принять решение об уничтожении или повторном создании контроллера пользовательского интерфейса в ответ на действия пользователя или системные события, которые находятся вне вашего контроля. Если система уничтожает или повторно создаёт контроллер пользовательского интерфейса, все хранящиеся в нём временные данные теряются.

Поскольку эти контроллеры пользовательского интерфейса в первую очередь отвечают за отображение информации пользователю и обработку событий пользовательского ввода, мы не хотим возлагать на них дополнительную ответственность за обработку данных. Поэтому нам нужна отдельная сущность для управления данными — ViewModel.



Для добавления ViewModel в проект необходимо добавить в build.gradle.ktx (Module :app) добавить следующие зависимости:

*dependencies* **{** val lifecycle\_version = "2.6.2" // as example  
 val activity\_version = "1.7.2" // as example  
   
 // other dependencies

*implementation* ("androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle\_version")  
 *implementation* ("androidx.activity:activity-ktx:$activity\_version")  
**}**

ViewModel — это модель, которая подготавливает и управляет данными для пользовательского интерфейса, а Activity или Fragment отображает данные из ViewModel. Однако ViewModel не должна содержать прямых ссылок на представления в иерархии представлений или ссылок на саму Activity или Fragment. Это отделяет владение данными от логики контроллера пользовательского интерфейса.

Объекты ViewModel ограничены жизненным циклом (жизненные циклы есть у Activity и Fragment). Объекты ViewModel автоматически сохраняются при изменении конфигурации, поэтому хранящиеся в них данные немедленно становятся доступны следующему экземпляру Activity или Fragment. ViewModel остаётся в памяти до тех пор, пока жизненный цикл, к которому она относится, не будет окончательно завершён.

В качестве примера приведено приложение для подсчета чаевых, в котором ViewModel ответственен за сохранение и переопределение значений полей ввода / вывода:

val amountInput: MutableState<String> = *mutableStateOf*("")  
var tip: MutableState<String> = *mutableStateOf*("No tips, hooray")  
val percentageInput: MutableState<String> = *mutableStateOf*("")  
val roundUp: MutableState<Boolean> = *mutableStateOf*(false)

Для создания класса ViewModel необходимо имплементировать класс ViewModel:

class TipCalculatorViewModel : ViewModel() {  
 // Код класса  
}

Так же в основном классе необходимо имплементировать созданный класс ViewModel

@Composable  
fun TipCalculatorScreen (  
 modifier: Modifier = Modifier**,** tipViewModel: TipCalculatorViewModel = viewModel()  
){ /\* код для экрана \*/ }

Для получения значения поля из ViewModel необходимо вызвать необходимое значение следующим образом:

val defaultAmount: String = tipViewModel.amountInput.value

Для его переопределения (либо вызова функции):

tipViewModel.onAmountInputChanged(**it)**

Для функции

class TipCalculatorViewModel : ViewModel() {  
 val amountInput: MutableState<String> = *mutableStateOf*("")  
 var tip: MutableState<String> = *mutableStateOf*("No tips, hooray")  
  
 fun onAmountInputChanged(amount: String) {  
 amountInput.value = amount  
 if (amount != "" && amount != "0") {  
 tip.value = calculateTip(billAmount = amount.*toDouble*()**,** tipPercentage = 15.0)  
 } else {  
 tip.value = "No tips, hooray!"  
 }  
 }  
}

Таким образом, при рекомпозиции сохранятся значения полей ввода / вывода информации.

Полный код ViewModel + CalculatorScreen

package com.raaveinm.learning.ui.screen  
  
import android.content.res.Configuration  
import androidx.compose.foundation.layout.Arrangement  
import androidx.compose.ui.text.input.ImeAction  
import androidx.compose.foundation.layout.Column  
import androidx.compose.foundation.layout.Spacer  
import androidx.compose.foundation.layout.fillMaxWidth  
import androidx.compose.foundation.layout.height  
import androidx.compose.foundation.layout.padding  
import androidx.compose.foundation.rememberScrollState  
import androidx.compose.foundation.verticalScroll  
import androidx.compose.material.icons.Icons  
import androidx.compose.material.icons.outlined.*AttachMoney*import androidx.compose.material.icons.outlined.*TipsAndUpdates*import androidx.compose.material3.MaterialTheme  
import androidx.compose.material3.Text  
import androidx.compose.runtime.Composable  
import androidx.compose.ui.Alignment  
import androidx.compose.ui.Modifier  
import androidx.compose.ui.res.stringResource  
import androidx.compose.ui.tooling.preview.Preview  
import androidx.compose.ui.tooling.preview.Wallpapers  
import androidx.compose.ui.unit.dp  
import androidx.lifecycle.viewmodel.compose.viewModel  
import com.raaveinm.learning.R  
import com.raaveinm.learning.ui.layouts.EditNumberField  
import com.raaveinm.learning.ui.layouts.SwitchLayout  
import com.raaveinm.learning.ui.viewmodel.TipCalculatorViewModel  
  
@Composable  
fun TipCalculatorScreen (  
 modifier: Modifier = Modifier**,** tipViewModel: TipCalculatorViewModel = viewModel()  
){  
 val billAmount: String = stringResource(R.string.*bill\_amount*)  
 val tipPercentage: String = stringResource(R.string.*how\_was\_the\_service*)  
  
 Column (modifier = modifier  
 .*padding*(horizontal = 40.*dp*)  
 .*verticalScroll*(rememberScrollState())**,** horizontalAlignment = Alignment.CenterHorizontally**,** verticalArrangement = Arrangement.Center  
 ) **{** Text (  
 text = stringResource(R.string.*calculate\_tip*)**,** modifier = Modifier  
 .*padding*(bottom = 16.*dp***,** top = 40.*dp*)  
 .*align*(alignment = Alignment.Start)  
 )  
  
 EditNumberField(  
 modifier = Modifier.*padding*(bottom = 32.*dp*).*fillMaxWidth*()**,** text = tipViewModel.amountInput.value**,** icon = Icons.Outlined.*AttachMoney***,** onValueChange = **{** tipViewModel.onAmountInputChanged(**it**) **},** label = billAmount**,** action = ImeAction.Next  
 )  
  
 EditNumberField(  
 modifier = Modifier.*padding*(bottom = 16.*dp*).*fillMaxWidth*()**,** text = tipViewModel.percentageInput.value**,** icon = Icons.Outlined.*TipsAndUpdates***,** label = tipPercentage**,** onValueChange = **{** tipViewModel.onPercentageInputChanged(**it**) **},** action = ImeAction.Done  
 )  
  
 SwitchLayout(  
 modifier = Modifier.*padding*(bottom = 32.*dp*)**,** text = R.string.*round\_up\_tip***,** isChecked = tipViewModel.roundUp.value**,** onCheckupChanged = **{** tipViewModel.roundUpChanged(**it**) **}** )  
  
 Text(  
 text = stringResource(R.string.*tip\_amount***,** tipViewModel.tip.value)**,** style = MaterialTheme.typography.displaySmall  
 )  
  
 Spacer(modifier = Modifier.*height*(150.*dp*))  
 **}**}  
  
@Preview(  
 device = "id:pixel\_9\_pro"**,** wallpaper = Wallpapers.NONE**,** showBackground = true**,** showSystemUi = true**,** uiMode = Configuration.*UI\_MODE\_NIGHT\_YES* or Configuration.*UI\_MODE\_TYPE\_UNDEFINED*)  
@Composable  
fun TipCalculatorLayoutPreview () {  
 TipCalculatorScreen()  
}

package com.raaveinm.learning.ui.viewmodel  
  
import androidx.compose.runtime.MutableState  
import androidx.compose.runtime.mutableStateOf  
import androidx.lifecycle.ViewModel  
import java.text.NumberFormat  
  
class TipCalculatorViewModel : ViewModel() {  
 val amountInput: MutableState<String> = *mutableStateOf*("")  
 var tip: MutableState<String> = *mutableStateOf*("No tips, hooray")  
 val percentageInput: MutableState<String> = *mutableStateOf*("")  
 val roundUp: MutableState<Boolean> = *mutableStateOf*(false)  
  
 fun onAmountInputChanged(amount: String) {  
 amountInput.value = amount  
 if (amount != "" && amount != "0") {  
 tip.value = calculateTip(billAmount = amount.*toDouble*()**,** tipPercentage = 15.0)  
 } else {  
 tip.value = "No tips, hooray!"  
 }  
 }  
  
 private fun calculateTip(billAmount: Double**,** tipPercentage: Double) : String {  
 var calculatedTip = tipPercentage / 100 \* billAmount  
 if (roundUp.value) calculatedTip = kotlin.math.*ceil*(calculatedTip)  
 return NumberFormat.getCurrencyInstance().format(calculatedTip)  
 }  
  
 fun onPercentageInputChanged(percentage: String) {  
 percentageInput.value = percentage  
 if (percentage != "" && percentage != "0") {  
 tip.value = calculateTip(billAmount = amountInput.value.*toDouble*()**,** tipPercentage = percentage.*toDouble*())  
 } else {  
 tip.value = "No tips, hooray!"  
 }  
 }  
  
 fun roundUpChanged(roundUp: Boolean) {  
 this.roundUp.value = roundUp  
 if (amountInput.value != "" && amountInput.value != "0") {  
 tip.value = calculateTip(billAmount = amountInput.value.*toDouble*()**,** tipPercentage = percentageInput.value.*toDouble*())  
 }  
 }  
}

# Часть II – Навигация

## Вступление

Многие приложения имеют несколько экранов, между которыми можно перемещаться. Например, приложение «Настройки» содержит множество страниц контента, разбросанных по разным экранам.

|  |  |  |
| --- | --- | --- |
|  |  |  |

Для Android многоэкранные приложения создаются с помощью компонента навигации Jetpack. Компонент Navigation Compose позволяет легко создавать многоэкранные приложения в Compose, используя декларативный подход, подобно тому, как создаются пользовательские интерфейсы.

Компонент Navigation состоит из трёх основных частей:

* NavController: отвечает за навигацию между пунктами назначения, то есть экранами вашего приложения.
* NavGraph: отображает компонуемые пункты назначения для навигации.
* NavHost: компонуемый элемент, выступающий в качестве контейнера для отображения текущего пункта назначения NavGraph.

Для работы NavHostController необходимо предварительно добавить следующие зависимости в build.gradle.ktx (Module :app)

plugins **{** // Kotlin serialization для типобезопасных маршрутов и навигационных аргументов  
 *kotlin*("plugin.serialization") *version* "2.0.21"  
**}***dependencies* **{** val nav\_version = "2.9.1"  
  
 // Интеграция Jetpack Compose  
 *implementation*("androidx.navigation:navigation-compose:$nav\_version")  
  
 // Тестирование  
 *androidTestImplementation*("androidx.navigation:navigation-testing:$nav\_version")  
  
 // Библиотека JSON serialization

*implementation*("org.jetbrains.kotlinx:kotlinx-serialization-json:1.7.3")  
**}**

## Создание NavGraph

Одной из основополагающих концепций навигации в приложении Compose является маршрут. Маршрут — это строка, соответствующая пункту назначения. Эта идея аналогична концепции URL. Подобно тому, как другой URL-адрес соответствует другой странице веб-сайта, маршрут — это строка, которая соответствует пункту назначения и служит его уникальным идентификатором. Пункт назначения обычно представляет собой один или группу объектов Composable, соответствующих тому, что видит пользователь.

В приложении конечное количество экранов, поэтому существует и конечное количество маршрутов. Маршруты определяются с помощью класса enum либо с помощью объектов. Классы enum в Kotlin имеют свойство name, которое возвращает строку с именем свойства.

enum class NavData(/\*передача доп информации между экранами\*/){  
 *MainScreen***,** *RecipeScreen***,** *DetailedImage*}

Либо при использовании объектов. Использование аннотации @Serializable необходимо для автоматического создания необходимых методов сериализации и десериализации для ваших типов маршрутов. Эта аннотация предоставляется плагином сериализации Kotlin.

import kotlinx.serialization.Serializable  
  
@Serializable  
object MainScreen  
  
@Serializable  
object RecipeScreen  
  
@Serializable  
object DetailedImage

Данный способ в последствии предоставляет возможность удобно передавать данные между экранами приложений. Если необходимо передать данные получателю, определите маршрут с классом, имеющим параметры.

import kotlinx.serialization.Serializable  
  
@Serializable  
object MainScreen  
  
@Serializable  
data class RecipeScreen(val coffeeNameId: Int) // передача значения Int  
  
@Serializable  
data class DetailedImage(val coffeeImage: Int) // передача значения Int

## Определение NavController

В Compose используется сериализуемый объект или класс для определения маршрута. Маршрут описывает, как добраться до пункта назначения, и содержит всю необходимую для него информацию.

После определения маршрутов используется составной объект NavHost для создания навигационного графа.

@Composable  
fun MainScreen (  
 navController: NavHostController = rememberNavController()  
) {  
 Scaffold(  
 modifier = Modifier  
 ) **{** innerPadding **->** NavHost(navController**,** startDestination = MainScreen) **{** *composable*<MainScreen> **{** /\*\*/ **}** *composable*<RecipeScreen> **{** /\*\*/ **}** *composable*<DetailedImage> **{** /\*\*/ **}**  
 **}  
 }  
 }**}

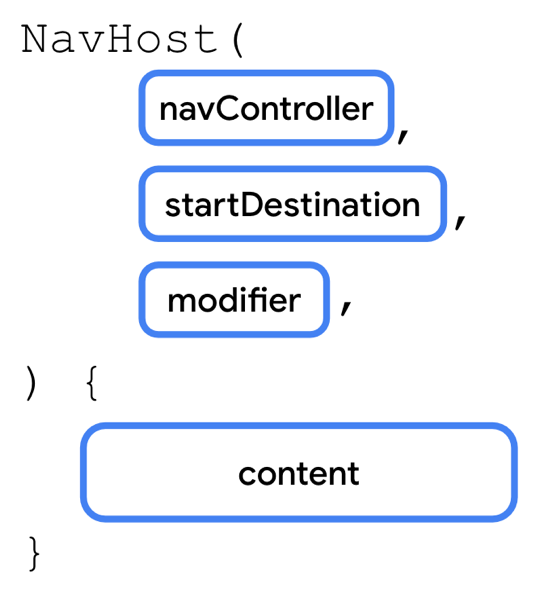
Вызов компонуемого объекта NavHost передаёт NavController и маршрут к начальной точке назначения.

Лямбда-функция, переданная в NavHost, в конечном итоге вызывает NavController.createGraph() и возвращает NavGraph.

Каждый маршрут предоставляется как аргумент типа для NavGraphBuilder.composable<T>(), который добавляет точку назначения к результирующему NavGraph.

## Создание NavHost

NavHost — это компонуемый объект, отображающий другие компонуемые пункты назначения на основе заданного маршрута. Синтаксис NavHost такой же, как у любого другого компонуемого объекта.



navController: Экземпляр класса NavHostController. Этот объект можно использовать для навигации между экранами, например, вызвав метод navigate() для перехода к другому пункту назначения. NavHostController можно получить, вызвав rememberNavController() из компонуемой функции.

startDestination: Строковый маршрут, определяющий пункт назначения, отображаемый по умолчанию при первом отображении NavHost.

В функции content объекта NavHost вызывается функция composable(). Функция composable() имеет два обязательных параметра.

route: Строка, соответствующая имени маршрута. Это может быть любая уникальная строка. Будет использоваться свойство name констант перечисления.

content: Здесь можно вызвать компонуемый объект, который нужно отобразить для заданного маршрута.

A screenshot of a computer

AI-generated content may be incorrect.

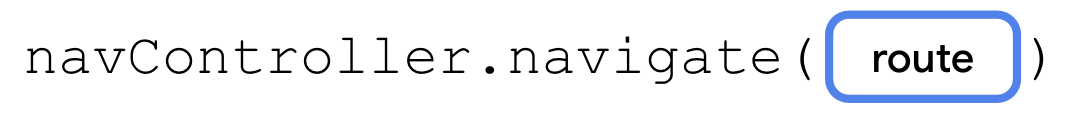
NavHost(navController**,** startDestination = MainScreen) **{** *composable*<MainScreen> **{** AllRecipes(navController**,** modifier = Modifier.*padding*(innerPadding))  
 **}** *composable*<RecipeScreen> **{** backStackEntry **->** val recipeScreen: RecipeScreen = backStackEntry.*toRoute*()  
 val selectedCoffee = *coffees*.*find* **{ it**.name == recipeScreen.coffeeNameId **}** if (selectedCoffee != null) {  
 RecipeExtended(  
 coffee = selectedCoffee**,** navController = navController  
 )  
 }  
 **}** *composable*<DetailedImage> **{** backStackEntry **->** val detailedImage: DetailedImage = backStackEntry.*toRoute*()  
 val selectedCoffee = *coffees*.*find* **{ it**.imageResourceId == detailedImage.coffeeImage **}** if (selectedCoffee != null) { FullScreenImage(image = selectedCoffee.imageResourceId) }  
 **}  
}**

## Навигация между экранами

NavHostController — свойство navController, полученное в результате вызова rememberNavController(), — отвечает за навигацию между маршрутами. Необходимо обеспечить доступ к нему с разных экранов вашего приложения.

Преимущество использования NavHost для управления навигацией в приложении заключается в том, что логика навигации отделена от пользовательского интерфейса. Этот вариант позволяет избежать некоторых основных недостатков передачи navController в качестве параметра.

Чтобы перейти на другой экран, достаточно вызвать метод navigate() в экземпляре NavHostController.



Метод принимает один параметр: строку, соответствующую маршруту, определённому в NavHost. Если маршрут соответствует одному из вызовов composable() в NavHost, приложение переходит на этот экран.

RecipeCard(  
 coffee = coffee**,** modifier = Modifier.*padding*(dimensionResource(R.dimen.*padding\_small*))**,** toExpandedScreen = **{** navController.navigate(RecipeScreen(/\*дополнительные параметры\*/))  
 **}**)

Для передачи дополнительной информации между экранами необходимо определить, что будет передано в NavGraph, после чего можно передать соответствующий объект. В данном примере – это ссылка на ресурс, определенный в strings.xml, соответственно это будет значение типа Int:

navController.navigate(RecipeScreen(coffeeNameId = coffee.name))

## Переход между приложениями

Этот элемент UI не является частью приложения. Фактически, он предоставляется операционной системой Android. Системный UI, такой как экран обмена, не вызывается navController. Вместо этого, используется Intent.

Intent — это запрос к системе на выполнение какого-либо действия, чаще всего — запуск новой activity. Существует множество различных intents, и рекомендуется обращаться к документации для получения полного списка. В данном случае будет рассмотрен ACTION\_SEND. Этому intent'у передаются некоторые данные, например строка, и передаются соответствующие действия для обмена этими данными.

Базовый процесс настройки intent выглядит следующим образом:

Создается объект intent указывается намерение, например, ACTION\_SEND.

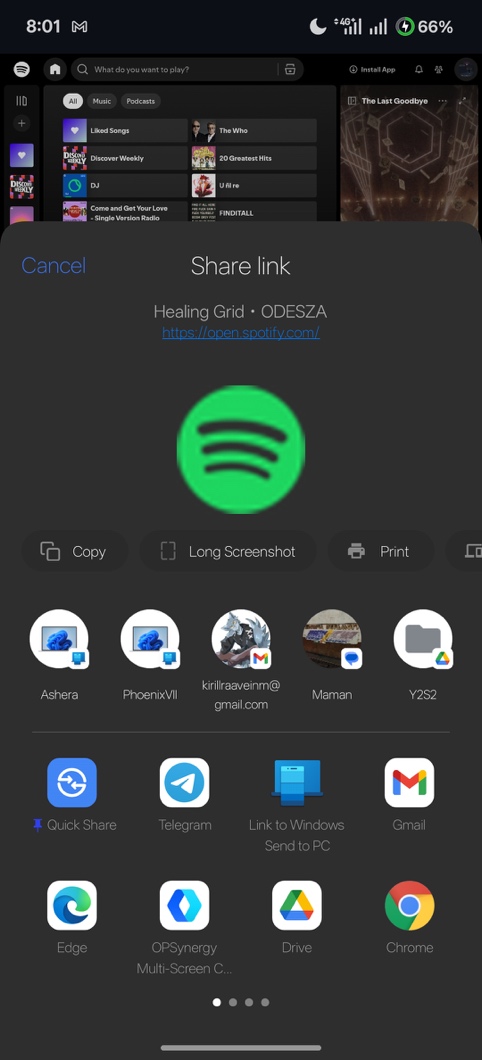
Указывается тип дополнительных данных, отправляемых с intent'ом. Для простого текста можно использовать "text/plain", хотя доступны и другие типы, такие как "image/\*" или "video/\*".

Передаются любые дополнительные данные в intent, например, текст или изображение для обмена, вызвав метод putExtra(). Этот intent примет два extras: EXTRA\_SUBJECT и EXTRA\_TEXT. Так же есть возможность добавить дополнительные данные, например, получателей электронных писем (EXTRA\_EMAIL, EXTRA\_CC, EXTRA\_BCC), тему электронного письма (EXTRA\_SUBJECT) и т. д.

Вызывается метод startActivity(), передав ему activity, созданную из intent'а.

val *sendIntent*: Intent = Intent().*apply* **{** *action* = Intent.*ACTION\_SEND* putExtra(Intent.*EXTRA\_TEXT***,** "This is my text to send.")  
 *type* = "text/plain"  
**}**val *shareIntent* = Intent.createChooser(*sendIntent***,** null)  
startActivity(shareIntent)

Таким образом будет вызван ранее описанный системный элемент UI:



Передача двоичных данных с помощью действия ACTION\_SEND. Задается соответствующий тип MIME и помещается URI к данным в дополнительный EXTRA\_STREAM, как показано в следующем примере. Этот метод обычно используется для обмена изображениями, но может применяться для обмена любым типом двоичного контента.

val *shareIntent*: Intent = Intent().*apply* **{** *action* = Intent.*ACTION\_SEND* // Example: content://com.google.android.apps.photos.contentprovider/...  
 putExtra(Intent.*EXTRA\_STREAM***,** uriToImage)  
 *type* = "image/jpeg"  
**}**startActivity(Intent.createChooser(shareIntent**,** null))

Чтобы поделиться несколькими фрагментами контента, можно использовать ACTION\_SEND\_MULTIPLE вместе со списком URI, указывающих на контент.

Тип MIME зависит от состава контента, которым вы делитесь. Например, если вы делитесь тремя изображениями JPEG, используйте тип «image/jpg». Для смешанных типов изображений используется «image/\*», чтобы сопоставить действие, обрабатывающее изображения любого типа.

Хотя обмен смешанными типами возможен, настоятельно не рекомендуется, поскольку получателю неясно, что именно должно быть отправлено. Если необходимо отправить несколько типов, используйте «\*/\*».

val *imageUris*: ArrayList<Uri> = *arrayListOf*(  
 // Add your image URIs here  
 imageUri1**,** imageUri2  
)  
  
val *shareIntent* = Intent().*apply* **{** *action* = Intent.*ACTION\_SEND\_MULTIPLE* putParcelableArrayListExtra(Intent.*EXTRA\_STREAM***,** *imageUris*)  
 *type* = "image/\*"  
**}**startActivity(Intent.createChooser(shareIntent**,** null))

Подробнее про передачу данных: <https://developer.android.com/training/sharing/send> .

# Часть III - Тестирование NavHost

## Создание теста исходного маршрута

При тестировании Compose Navigation не будет доступа к тому же NavHostController, который есть в коде приложения. Однако можно использовать TestNavHostController и настроить тестовое правило с этим навигационным контроллером.

Перед созданием теста необходимо создать тестовое правило и передать *ComponentActivity* в качестве параметра:

import androidx.activity.ComponentActivity  
import androidx.compose.ui.test.junit4.createAndroidComposeRule  
import org.junit.Rule  
  
@get:Rule  
val *composableTestRule* = *createAndroidComposeRule*<ComponentActivity>()

Чтобы убедиться, что приложение переходит в правильное место, необходимо ссылаться на экземпляр TestNavHostController для проверки маршрута хоста, когда приложение выполняет действия по навигации.

Далее необходимо создать экземпляр TestNavHostController как переменную lateinit. В Kotlin ключевое слово lateinit используется для объявления свойства, которое можно инициализировать после объявления объекта.

private lateinit var *navController*: TestNavHostController

Затем необходимо указать составной элемент, который будет использоваться для UI тестов, необходимо создать объект TestNavHostContoller в тестовом классе. Этот объект будет использоваться позже для определения состояния навигации, поскольку приложение использует контроллер для навигации по различным экранам.

fun setupMainScreen(){  
 *composableTestRule*.setContent **{** *navController* = TestNavHostController(*LocalContext*.current).*apply* **{** navigatorProvider.addNavigator(ComposeNavigator())  
 **}** MainScreen(navController = *navController*)  
 **}**}

Каждый тест в классе включает в себя тестирование какого-либо аспекта навигации. Поэтому каждый тест зависит от созданного объекта TestNavHostController. Вместо того, чтобы вручную вызывать функцию setupMainScreenNavHost() для каждого теста для настройки навигационного контроллера, сделать это можно автоматически с помощью аннотации @Before из библиотеки junit. Если метод аннотирован @Before, он выполняется перед каждым методом с аннотацией @Test.

@Before  
fun setupMainScreenNavHost(){  
 composableTestRule.setContent **{** navController = TestNavHostController(*LocalContext*.current).*apply* **{** navigatorProvider.addNavigator(ComposeNavigator())  
 **}** MainScreen(navController = navController)  
 **}**}

Для тестирования маршрута, заданного следующим образом

package com.raaveinm.learning.ui.navigation  
  
import kotlinx.serialization.Serializable  
  
@Serializable  
object MainScreen  
  
@Serializable  
data class RecipeScreen(val coffeeNameId: Int)  
  
@Serializable  
data class DetailedImage(val coffeeImage: Int)

Необходимо следующая проверка

@Test  
fun appNavHost\_verifyStartDestination(){  
 assertEquals(  
 MainScreen::class.qualifiedName**,** navController.currentDestination?.route  
 )  
}

При указывании маршрутов в приложении при помощи перечисления – необходимо вызвать путь, указанный в enum

assertEquals(  
 AppRoutes.MainScreen.name**,** navController.currentBackStackEntry?.destination?.route  
)

## Вспомогательные методы

Тесты пользовательского интерфейса часто требуют повторения шагов для приведения пользовательского интерфейса в состояние, в котором можно протестировать определённый его элемент. Пользовательский интерфейс также может требовать сложных утверждений, требующих нескольких строк кода. Метод assertion требует большого объёма кода, и используется много раз при тестировании навигации в приложениях. В таких ситуациях написание вспомогательных методов в тестах избавляет от необходимости дублировать код.

Для каждого написанного теста навигации используется свойство .*qualifiedName* (.*name* для элементов *enum*) для проверки корректности текущего маршрута назначения контроллера навигации. Для удобства можно написать вспомогательную функцию, которую можно вызывать в любой момент для проверки. Для этого нужно создать новый файл (ScreenAssertions.kt), в котором будут прописаны маршруты и базовые утверждения

import androidx.navigation.NavController  
import org.junit.Assert.assertEquals  
  
fun NavController.assertionCurrentName(expectedName: String?){  
 assertEquals(expectedName**,** currentBackStackEntry?.destination?.route))  
}

Тогда основной тестовый фал будет выглядеть следующим образом:

@Test  
fun appNavHost\_verifyStartDestination() = navController.*assertionCurrentName*(MainScreen::class.qualifiedName)

# Часть IV – Адаптивность

## Запуск на различных устройствах

Чтобы создавать удобные приложения, разработчикам необходимо понимать, как пользователи взаимодействуют с устройствами различных форм-факторов. Поэтому необходимо тестировать приложения на различных форм-факторах с самого начала разработки.

Для достижения этой цели можно использовать множество эмуляторов с разными размерами экрана. Однако это может быть затруднительно, особенно если вы разрабатываете приложение для нескольких размеров экрана одновременно. Вам также может потребоваться протестировать реакцию работающего приложения на изменения размера экрана, такие как изменение ориентации, изменение размера окна на рабочем столе и изменение состояния сворачивания в складном режиме.

Android Studio помогает вам тестировать эти сценарии благодаря эмулятору изменяемого размера.

В Android Studio выберите «Инструменты» > «Диспетчер устройств».

A screenshot of a computer

AI-generated content may be incorrect.

Затем необходимо создать виртуальное устройство

A screenshot of a device

AI-generated content may be incorrect.

И выбрать устройство изменяемого размера:

A screenshot of a computer

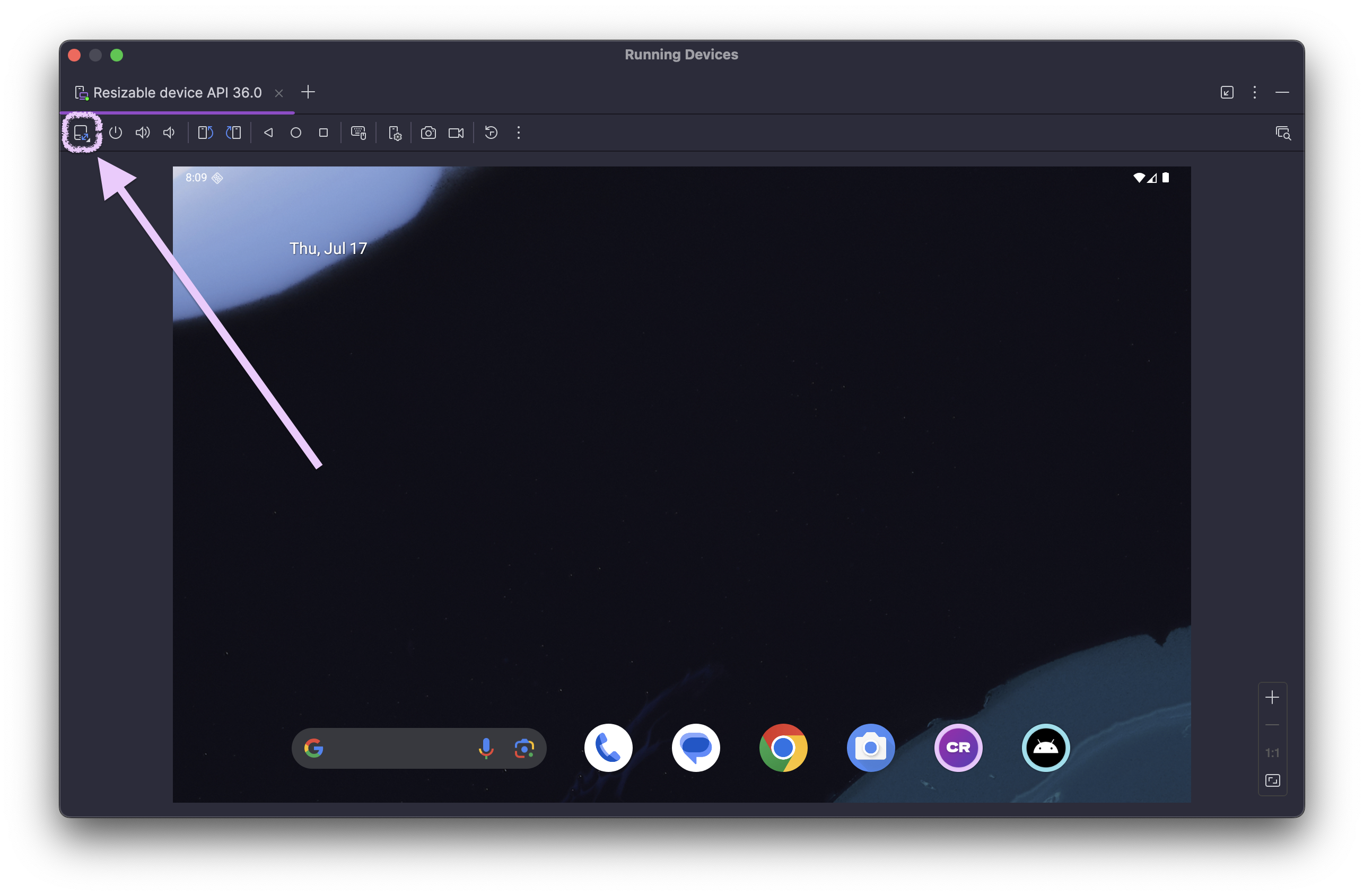
AI-generated content may be incorrect.

Далее необходимо задать базовые настройки, выбрать версию Android:

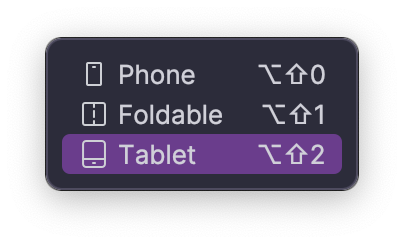
A screenshot of a device

AI-generated content may be incorrect.

Для выбора форм-фактор созданного устройства необходимо открыть соответствующее меню:



Затем, во всплывшем окне выбрать необходимый тип устройства:

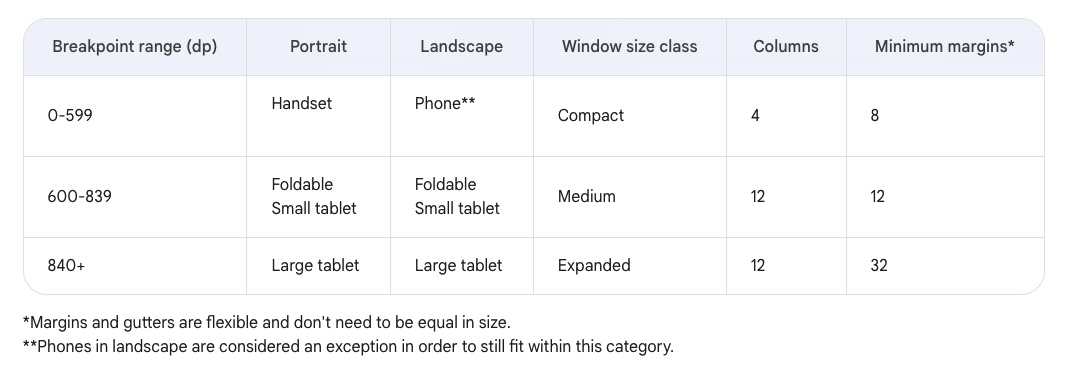


## Создание UI для различных форм-факторов

При проектировании для больших экранов, таких как планшеты и складные устройства, необходимо учитывать эргономику пользователя и близость его пальцев к экрану. На мобильных устройствах пальцы пользователя легко дотягиваются до большей части экрана; расположение интерактивных элементов, таких как кнопки и элементы навигации, не так критично. Однако на больших экранах размещение важных интерактивных элементов в центре экрана может затруднять доступ к ним.

Портретный режим является основной ориентацией для телефонов, а альбомный — для планшетов. При разработке с учётом адаптивности соответствующий размер экрана определяется текущим размером окна и ориентацией устройства, а не только его размером.

Для создания адаптивного приложения необходимо, чтобы макет менялся в зависимости от размера экрана. Точка измерения, в которой меняется макет, называется breakpoint (контрольной точкой). В Material Design учтен диапазон breakpoint, охватывающий большинство экранов Android.



API WindowSizeClass, представленный в Compose, упрощает реализацию контрольных точек Material Design.

Классы размеров окон вводят три категории размеров: компактный, средний и расширенный, как для ширины, так и для высоты.

Для использования данного API необходимо для начала добавить следующую зависимость в gradle:

*dependencies* **{**  
 *implementation*("androidx.compose.material3:material3-window-size-class")  
**}**

После чего получить размер экрана можно следующим образом:

val windowSize = calculateWindowSizeClass(this)

Пример обработки UI для различных устройств:

@Composable  
fun AdaptiveApp(  
 windowSize: WindowWidthSizeClass**,** modifier: Modifier = Modifier**,**) {  
 when (windowSize){  
 WindowWidthSizeClass.Compact -> {

/\* для телефонов\*/

}  
 WindowWidthSizeClass.Medium -> {

/\* для маленьких планшетов/разворачиваемых телефонов\*/

}  
 WindowWidthSizeClass.Expanded -> {

/\* для больших планшетов\*/

}  
 else -> {}  
 }

Для лучшей практики можно ознакомиться с основами паттернов качества приложения <https://developer.android.com/docs/quality-guidelines/core-app-quality> .

**Часть V – Практическое задание**

**Задание 1: реализовать логирование жизненного цикла Activity**

Добавить логирование в MainActivity для отслеживания состояний её жизненного цикла.

Переопределить методы onCreate(), onStart(), onResume(), onPause(), onStop() и onDestroy().

В каждом из переопределенных методов добавить вызов Log.d(), который будет записывать в лог имя вызванного метода. Использовать единый TAG для всех сообщений.

Запустить приложение и изучить вывод в окне **Logcat** при сворачивании и разворачивании приложения, а также при изменении ориентации устройства.

**Задание 2: сохранить состояние UI с помощью ViewModel**

Разработать экран с простым счетчиком, который сохраняет свое состояние при изменениях конфигурации (например, при повороте экрана).

Создать класс, наследуемый от ViewModel, который будет содержать изменяемую переменную состояния для хранения значения счетчика.

На экране UI отобразить значение счетчика, получая его из ViewModel.

Добавить кнопку, которая при нажатии будет вызывать функцию в ViewModel для увеличения значения счетчика. ViewModel должен обеспечивать сохранение состояния счетчика, чтобы его значение не сбрасывалось при пересоздании Activity.

**Задание 3: реализовать навигацию между экранами**

Создать приложение с двумя экранами и реализовать навигацию между ними.

Добавить в проект зависимость navigation-compose.

Создать NavHost в основной Composable-функции. Определить два экрана ("Home" и "Details") с помощью функции composable().

На экране "Home" разместить кнопку, которая по нажатию будет осуществлять переход на экран "Details" с помощью navController.navigate().

На экране "Details" отобразить любой информационный текст.

**Задание 4: адаптировать пользовательский интерфейс под разные размеры экрана**

Добавить в проект зависимость material3-window-size-class.

В основной Activity использовать функцию calculateWindowSizeClass для определения класса размера окна устройства.

Создать Composable-функцию, которая будет принимать WindowWidthSizeClass в качестве параметра.

Внутри этой функции использовать оператор when для изменения макета в зависимости от класса размера окна (Compact, Medium, Expanded).

Для Compact ширины отображать макет в одну колонку.

Для Medium или Expanded ширины отображать макет в две колонки (например, список слева и детали справа).