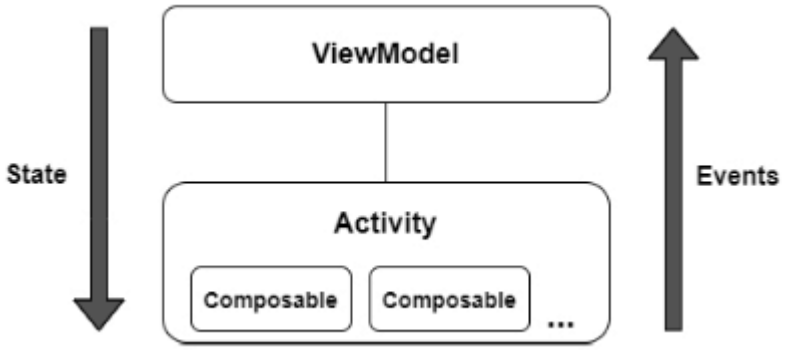
# ViewModel

## Хранение состояния во ViewModel и взаимодействие с интерфейсом

ViewModel позволяет отделить модель данных и логику приложения, связанную с пользовательским интерфейсом, от кода, который отвечает за отображение пользовательского интерфейса и управление им, а также взаимодействие с операционной системой. При таком подходе приложение будет состоять из одного или нескольких объектов Activity, которые представляют пользовательский интерфейс, а также ряда объектов ViewModel, которые будут отвечать за обработку данных. Классы Activity, которые представляют пользовательский интерфейс, отслеживают состояние модели во ViewModel, поэтому любые изменения данных вызывают рекомпозицию. События пользовательского интерфейса, например, нажатие кнопки, настраиваются для вызова соответствующей функции в ViewModel. По сути, это прямая реализация концепции однонаправленного потока данных:




Такое разделение ответственности решает проблемы, связанные с жизненным циклом класса Activity. Независимо от того, сколько раз объект Activity воссоздается в течение жизненного цикла приложения, экземпляры ViewModel остаются в памяти, тем самым обеспечивая согласованность данных. Например, ViewModel, используемая объектом Activity, будет оставаться в памяти до тех пор, пока этот объект Activity не завершит свою работу. А если в приложении только один объект Activity, то ViewModel остается в памяти до конца работы приложения.

ViewModel реализуется как отдельный класс, который содержит данные состояния - данные модели и функции для управления этими данными:

```
import androidx.lifecycle.ViewModel
.....................................

class MyViewModel : ViewModel() {

}
```

Основная цель ViewModel — обеспечить работу с данными с помощью пользовательского интерфейса. При этом сами данные могут браться из внешнего источника, например, из базы данных или веб-сервиса. (В этом случае обычно определяется отдельный модуль репозитория, который не связан с

ViewModel). Пользовательский интерфейс же может реагировать на изменения данных во ViewModel. Существует два способа объявить данные внутри ViewModel, чтобы их можно было наблюдать:

- Определить данные как состояние
- Использовать компонент LiveData

При первом подходе данные определяются с помощью функции mutableStateOf() (или аналогичных функциий, которые создают состояние типа State)

```
class MyViewModel : ViewModel() {
    var count by mutableStateOf(0)  // состояние ViewModel
}
```

Вместе с состоянием могут обычно определяются функции, которые можно вызывать из пользовательского интерфейса для изменения состояния:

```
class MyViewModel : ViewModel() {
    var count by mutableStateOf(0)

    fun increase() {
        count++
    }
}
```

Определив класс ViewModel, его можно использовать в компонентах пользовательского интерфейса. Однако для его создания нам еще понадобится функция viewModel(), которая по умолчанию отсутствует в стандартном проекте Compose. Поэтому нам вручную надо добавить соответствующую зависимость в файл build.gradle. Поэтому вначале откроем в проекте файл libs.version.toml и в секцию [libraries] добавим определение библиотеки:

```
[libraries]

androidx-lifecycle-viewmodel-compose = { module = "androidx.lifecycle:lifecycle-
viewmodel-compose", version.ref = "lifecycleRuntimeKtx" }
```

Далее откроем файл build.gradle.kts (Module :app) и в секцию dependencies добавим соответствующую зависимость:

```
implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.7.0")
```

То есть в итоге получится

```
dependencies {

    implementation(libs.androidx.lifecycle.viewmodel.compose)
    // остальные зависимости
    ....................................

}
```

Затем синхронизируем проект, нажав на кнопку Sync Now

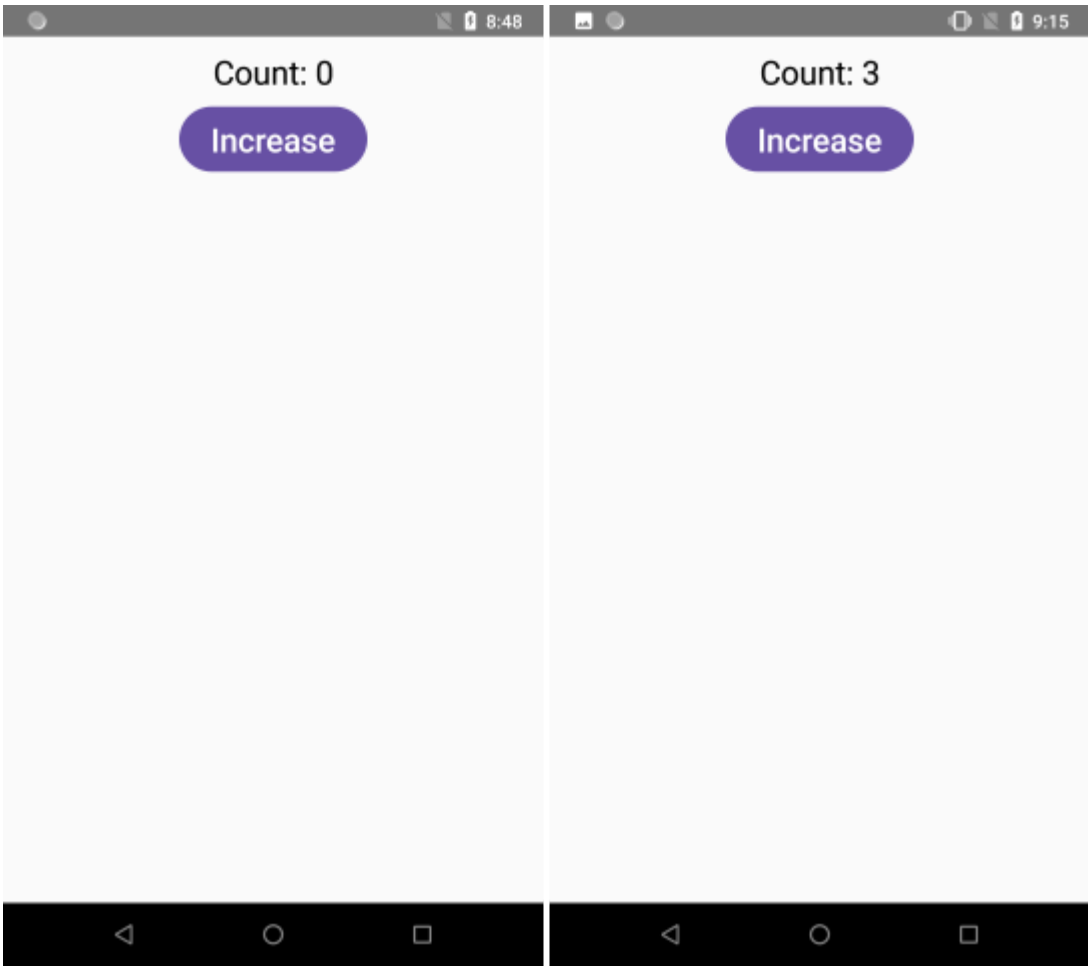
После добавления зависимости мы можем использовать ViewModel. Для этого надо передать экземпляр ViewModel в качестве параметра компоненту, из которого можно получить доступ к значениям состояния и функциям. Причем рекомендуется передавать ViewModel в компонент, который находится на вершине иерархии компонентов. А из этого компонента при необходимости состояние и функции для управления состоянием могут быть переданы дочерним компонентам. Например:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.unit.sp
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewmodel.compose.viewModel
import androidx.compose.runtime.Composable


class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            CounterView()
        }
    }
}

class CounterViewModel : ViewModel() {
    var count by mutableStateOf(0)
```

```
    fun increase() {
        count++
    }
}
@Composable
fun CounterView(vm: CounterViewModel = viewModel()) {

    Column(Modifier.fillMaxWidth(), horizontalAlignment =
Alignment.CenterHorizontally) {
        Text("Count: ${vm.count}", Modifier.padding(10.dp), fontSize = 25.sp)
        Button(onClick = {vm.increase()}) {
            Text(text = "Increase", fontSize = 25.sp)
        }
    }
}
```

Здесь определен компонент верхнего уровня CounterView, который инкапсулирует весь остальной интерфейс. В этот компонент в качестве параметра передается объект CounterViewModel - наш ViewModel, который определяет состояние - переменную count и функцию increase для изменения этого состояния. Причем для создания CounterViewModel применяется не его конструктор, а функция viewModel(). Если экземпляр ViewModel (в нашем случае CounterViewModel) уже создан в текущем контексте, то функция viewModel() вернет ссылку на этот экземпляр. В противном случае будет создан и возвращен новый экземпляр ViewModel.

Внутри CounterView определяем текстовую метку, которая выводит состояние, и кнопку, которая вызывает функцию increase. Таким образом, при нажатии на кнопку произойдет вызов функции increase, которая, в свою очередь, увеличивает состояние count. Это изменение состояния вызывает рекомпозицию пользовательского интерфейса, в результате чего новое значение count отобразится в компоненте Text:




# LiveData

Компонент LiveData представляет еще один способ управления дангными в ViewModel и может использоваться в качестве обертки для данных во ViewModel. После помещения в LiveData данные становятся доступными для компонентов внутри класса Activity. Объекты LiveData можно объявить как изменяемые с помощью класса MutableLiveData, что позволяет функциям ViewModel вносить изменения в базовые значения данных.

Для работы с LiveData в файле libs.versions.toml в секции `[libraries]` укажем саму библиотеку, а в секции `[versions]` укажем версию этой библиотеки:

```
[versions]
runtimeLivedata = "1.6.5"
.....................


[libraries]
androidx-runtime-livedata = { module = "androidx.compose.runtime:runtime-
livedata", version.ref = "runtimeLivedata" }
androidx-lifecycle-viewmodel-compose = { module = "androidx.lifecycle:lifecycle-
viewmodel-compose", version.ref = "lifecycleRuntimeKtx" }
.................
```

Далее в секции [versions] укажем версию этой библиотеки:

```
[versions]
.................
roomRuntime = "2.6.1"
```

Затем в файл build.grade.kts добавим соответствующую зависимость:

То есть в итоге получится

```
dependencies {

    implementation (libs.androidx.runtime.livedata)
    implementation(libs.androidx.lifecycle.viewmodel.compose)

    implementation(libs.androidx.core.ktx)
    // остальные зависимости
    ....................................

}
```

Пример ViewModel, которая использует LiveData:

```
class LiveCounterViewModel : ViewModel() {
    var count: MutableLiveData<Int> = MutableLiveData(0)
    fun increase() {
        count.value = count.value!! + 1
    }
}
```

Здесь у нас есть переменная count, которая представляет тип MutableLiveData - обертку над числом. И есть функция increase, которая увеличивает значение в count. Поскольку теоретически значение в MutableLiveData может представлять null, однако в нашем случае это невозможная ситуация, так как значение в любом случае представляет некоторое число, то при увеличении с помощью оператора !! говорим, что это значение не равно null.

Так, перепишем прошлый пример, использовав LiveData:

```
package com.example.helloapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.padding
```

```
import androidx.compose.material3.Button
import androidx.compose.material3.Text
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.runtime.getValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.unit.sp
import androidx.lifecycle.ViewModel
import androidx.compose.runtime.Composable
import androidx.compose.runtime.livedata.observeAsState
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.viewmodel.compose.viewModel


class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            CounterView()
        }
    }
}
class LiveCounterViewModel : ViewModel() {
    var count: MutableLiveData<Int> = MutableLiveData(0)
    fun increase() {
        count.value = count.value!! + 1
    }
}

@Composable
fun CounterView(vm: LiveCounterViewModel = viewModel()) {
    val countValue by vm.count.observeAsState(0)

    Column(Modifier.fillMaxWidth(), horizontalAlignment =
Alignment.CenterHorizontally) {
        Text("Count: ${countValue}", Modifier.padding(10.dp), fontSize = 25.sp)

        Button(onClick = {vm.increase()}) {
            Text(text = "Increase", fontSize = 25.sp)
        }
    }
}
```

Здесь определен компонент верхнего уровня CounterView, который инкапсулирует весь остальной интерфейс. В этот компонент в качестве параметра передается объект LiveCounterViewModel - наш ViewModel, который определяет состояние - переменную count и функцию increase для изменения этого состояния. Причем для создания CounterViewModel применяется не его конструктор, а функция viewModel(). Если экземпляр ViewModel (в нашем случае LiveCounterViewModel) уже создан в текущем контексте, то функция viewModel() вернет ссылку на этот экземпляр. В противном случае будет создан и возвращен новый экземпляр ViewModel.

Для отслеживания изменения значения в LiveData компонент определяет дополнительную переменную countValue:

```
val countValue by vm.count.observeAsState(0)
```

Вызов функции observeAsState() преобразует значение из LiveData в объект состояния и присваивает его переменной countValue. После преобразования это состояние будет отслеживать, и при его изменении будет запускаться рекомпозиция при компонента.

Внутри CounterView определяем текстовую метку, которая выводит состояние, и кнопку, которая вызывает функцию increase. Таким образом, при нажатии на кнопку произойдет вызов функции increase, которая, в свою очередь, увеличивает состояние count. Это изменение состояния вызывает рекомпозицию пользовательского интерфейса, в результате чего новое значение count отобразится в компоненте Text: