

Сохранение состояния. ViewModel. LiveData

Сохранение состояния

При изменении конфигурации устройства (например, повороте экрана, смене светлой темы оформления на тёмную, изменении общесистемного языка и т. д.) текущая активность будет уничтожена и создана заново. Это связано с тем что к элементам активности могут быть применены альтернативные файлы разметки, цветов, языков и пр., которые соответствуют изменившимся спецификаторам новой конфигурации. И не факт, что, например, все элементы разметки сохранятся – может быть, в новой конфигурации какие-то элементы станут не нужны, а вместо них могут появиться другие. Поэтому системе проще уничтожить активность и пересоздать её.

При этом пользователи могут столкнуться с ситуацией, когда на экране ими были сделаны какие-то изменения, а после поворота экрана они вдруг пропали, всё вернулось в исходное состояние. Это, конечно, может вызвать недоумение или даже возмущение, если музыкальный трек вдруг опять начнёт воспроизводиться с начала, а в заполненных текстовых полях пропадёт введённый текст.

Поэтому разработчик приложения должен позаботиться о своевременном сохранении и восстановлении данных. Для этого Android предлагает несколько вариантов.

Классический подход – это обработка событий `onSaveInstanceState` и `onRestoreInstanceState`. Система вызывает их в тех случаях, когда активность уничтожается, но есть вероятность что она может быть восстановлена, например, при нехватке памяти или изменении конфигурации. Если же активность уничтожается окончательно (например, приложение завершает свою работу), то эти методы вызываться не будут.

Метод `onSaveInstanceState` вызывается для сохранения состояния. В качестве параметра он получает объект `outState`, в котором будут храниться значения. Этот объект принадлежит классу `Bundle`, который является универсальным хранилищем в Android, и содержит методы для записи значений (`putInt`, `putString`, `putFloat`, ...) и для их последующего извлечения (`getInt`, `getString`, `getFloat`, ...). Каждое записываемое значение сохраняется под определённым текстовым ключом, например, в примере сохраняется позиция вертикальной прокрутки элемента `ScrollView`:

```
override fun onSaveInstanceState(outState: Bundle) {  
    super.onSaveInstanceState(outState)  
    outState.putInt("scroll_pos", scrollView.scrollY)  
}
```

Вместе с этим методом должен быть реализован и парный ему метод `onRestoreInstanceState`, который будет вызываться для восстановления состояния после пересоздания активности:

```
override fun onRestoreInstanceState(savedInstanceState: Bundle) {  
    super.onRestoreInstanceState(savedInstanceState)  
    scrollView.scrollY = savedInstanceState.getInt("scroll_pos", 0)  
}
```

На вход этот метод получает объект `savedInstanceState` с данными, сохранёнными ранее в методе `onSaveInstanceState`.

Если эти методы не реализованы, то Android пытается самостоятельно сделать хоть что-то: будут сохранены и восстановлены данные для всех элементов, у которых задан идентификатор `id`. Однако, это будут именно *данные*, а не другие свойства: например, для поля ввода `EditText` будет сохранено и восстановлено свойство `text`, но если в процессе работы изменялся цвет текста, его размер и т. д., то эта информация будет утеряна. О таких вещах лучше позаботиться самостоятельно.

ViewModel

Другой подход, который можно использовать для сохранения данных – это класс `ViewModel`, однако его возможности этим не исчерпываются.

`ViewModel` – это часть паттерна проектирования MVVM – Model-View-ViewModel, которая используется для более удобной организации кода в крупных проектах. Суть этого паттерна заключается в том, чтобы отделить данные, с которыми работает приложение, от механизма их отображения в интерфейсе. Каждая часть паттерна занимается своим делом:

- `Model` – обеспечивает работу с данными приложения: хранит их в памяти, или подгружает из базы данных и т. д.
- `View` – отображает данные, обычно это элементы управления в пользовательском интерфейсе
- `ViewModel` – реализует логику отображения `View` в зависимости от данных, которые предоставляет `Model`: форматирует числа с плавающей точкой в привычный вид, отображает или скрывает элементы управления в зависимости от данных, отображает информацию об ошибках или недоступности данных и пр.

Класс `ViewModel` в Android можно использовать и как часть паттерна MVVM, и просто для сохранения данных. В данной теме рассматривается второй вариант.

Важной особенностью класса `ViewModel` является то, что при создании он связывается с текущим объектом, реализующим интерфейс `ViewModelStoreOwner` – это, например, активности и фрагменты. Если активность будет уничтожена, то объект класса `ViewModel` продолжит своё существование со всеми сохранёнными данными, и при пересоздании активности к ним снова можно будет получить доступ.

Кроме хранения данных в объект класса `ViewModel` можно добавлять и методы, то есть можно реализовать часть бизнес-логики приложения. Конечно, в серьёзных проектах бизнес-логика обычно реализуется в классах с данными, но в часть кода, относящуюся к представлению данных, всё равно можно выносить в объект класса `ViewModel`.

Чтобы воспользоваться классом `ViewModel`, нужно создать дочерний класс и добавить туда нужные поля и методы:

```
class MyViewModel : ViewModel() {  
    var color = 0xFF80FF80  
    var width = 5  
    var height = 10  
}
```

```

    fun area(): Int {
        return width * height
    }
}

```

Как видно из примера, класс-наследник `MyViewModel` представляет собой по сути обычный класс с полями и методами, только ему дан шанс пережить разрушение и восстановление активности или фрагмента.

Обычно доступ к объекту класса `ViewModel` получается в методе `onCreate`, когда производится инициализация активности:

```

val provider = ViewModelProvider(this)
val model = provider[MyViewModel::class.java]

```

Здесь сначала получается ссылка на хранилище объектов `ViewModel`, в качестве параметра передаётся ссылка на объект, к жизненному циклу которого будет сделана привязка. В данном примере привязка идёт к текущей активности, поэтому передаётся ссылка на неё, `this`.

Вторая строка запрашивает у хранилища сам объект класса `ViewModel` – точнее, его дочерний класс с нужными программе полями, методами и т. п. Для этого передаётся ссылка на класс в формате, принятом в языке Kotlin: `MyViewModel::class.java`. Если такой объект уже есть, возвращается ссылка на него, если нет – он создаётся и ссылка на него также будет возвращена.

Синглтоны

В Kotlin есть ещё один вариант сохранения данных при изменении конфигурации устройства – это синглтон (*англ.* singleton).

Синглтон представляет собой уникальный глобальный объект, другой такой создать в приложении не получится. Внешне синглтон похож на класс, в нём также могут быть поля и методы, он может реализовывать интерфейсы, однако вместо ключевого слова `class` при описании используется ключевое слово `object`:

```

object MyData {
    var color = 0xFF80FF80
    var width = 5
    var height = 10

    fun area(): Int {
        return width * height
    }
}

```

Здесь повторён тот же пример, который выше использовался при описании дочернего класса `MyViewModel`. Однако, в отличие от него, синглтон не нужно каким-то особым образом получать у системы, он автоматически создаётся при запуске приложения и существует всё время пока приложение запущено. Например, если в приложении есть активность и

фоновый сервис, и активность была разрушена и восстановлена, но сервис продолжал работать – все данные в синглтоне сохраняются.

Кажется, синглтон удобнее чем ViewModel, и если требуется сохранять данные, которые нужны в разных частях приложения (т. е. общие для нескольких активностей, сервисов и т. д.) то это действительно так. Однако если для каждой активности требуются свои данные, которые не должны пересекаться или быть доступными другим, то рекомендуется использовать ViewModel. Даже если запущено несколько экземпляров одной активности – каждая из них может получить свой объект ViewModel и сохранять там свои свойства, настройки, параметры и т. д., в случае же синглтона эти данные будут общими для всех экземпляров. Поэтому выбирать между ViewModel и синглтоном нужно в зависимости от ситуации и контекста.

LiveData

Поля в классах позволяют сохранять данные, но сторонние объекты не знают о том, что данные в поле изменились. А иногда это бывает очень полезно. Для таких случаев предусмотрен объект LiveData: он позволяет подписаться на уведомление об изменении значения поля. Более того, он отслеживает существование подписчиков: если какой-то подписчик был уничтожен (например, активность, которая отслеживала изменения), то это не приведёт к падению программы при попытке вызова несуществующего метода.

Создаётся переменная следующим образом:

```
val width = MutableLiveData<Int>()
```

Можно указать начальное значение поля, тогда тип может быть определён по нему:

```
val width = MutableLiveData(5)
```

Если начальное значение не указано, то поле будет содержать `null`.

Для доступа к значению используется внутреннее поле `value`:

```
width.value = 7
```

Однако чтение значения чуть сложнее, ведь в поле значения может оказаться `null`, нужно использовать оператор Элвиса:

```
fun area(): Int {  
    return (width.value ?: 0) * height  
}
```

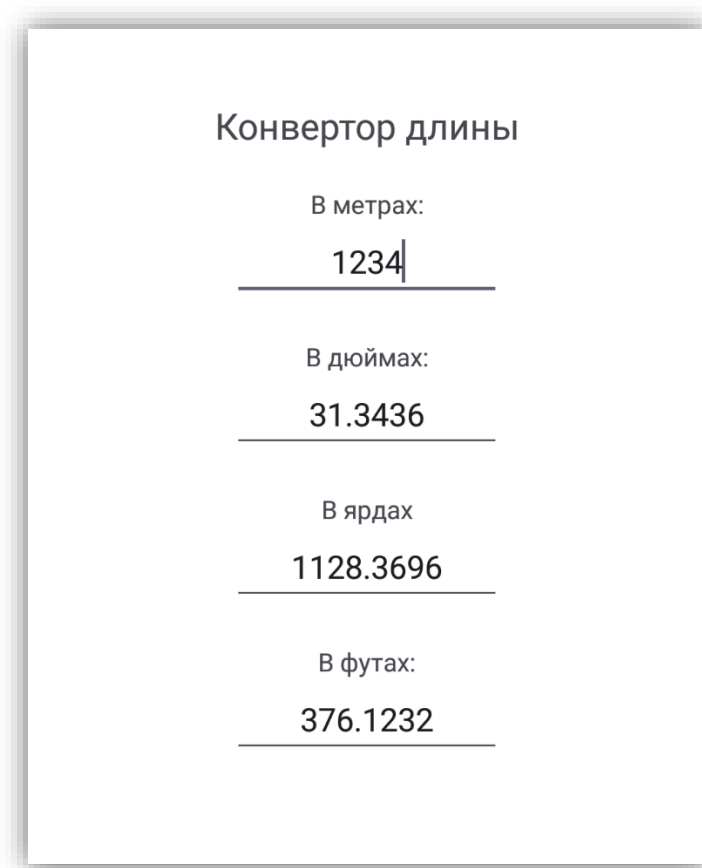
Чтобы подписаться на обновления используется метод `observe`:

```
width.observe(this) {  
    print("Новое значение: ${it ?: 0}")  
}
```

В качестве параметра ему передаётся ссылка на объект, реализующий интерфейс `MyViewModel` – обычно это активность. Это нужно для того чтобы объект `LiveData` мог отслеживать существование подписчика, и не пытался отправить уведомление подписчику, которого уже не существует.

Задание

Разработайте приложение для конвертации чего-нибудь во что-нибудь, из одной величины в несколько других величин. Главная активность приложения должна выглядеть примерно следующим образом:



Конвертор длины

В метрах:
1234

В дюймах:
31.3436

В ярдах:
1128.3696

В футах:
376.1232

При редактировании значения в одном из полей – в других полях должно сразу же появляться новое преобразованное значение. Для этого можно использовать слушатель поля ввода `EditText`:

```
etMeters.doAfterTextChanged {  
    // Действия при изменении текста  
}
```

При изменении конфигурации устройства (смене ориентации, темы оформления со светлой на тёмную и т. д.) значения во всех полях должны сохраняться.

Поля для сохранения данных должны быть либо в объекте `ViewModel`, либо в синглтоне, на выбор. Там же должны быть методы и константы для преобразования значений из одного в другое.