Хранение данных

Сохраняем простые данные

Содержание

1. Сохраняем счет времени после смерти приложения

В предыдущей лекции мы рассмотрели простое приложение которое отображает таймер на экране. Он работал таким образом: пока приложение живое счетчик считал количество секунд, когда юзер смотрел в приложение, то видел актуальное количество секунд. Теперь же мы немного переработаем это.

Новые условия: считать количество секунд просмотренное юзером в приложении. Что это значит для нас? Нужно начинать считать секудны лишь тогда, когда юзер находится в приложении и останавливать когда он не смотрит в наш активити. Т.е. старт в методе onResume и стоп в методе onPause. Также мы хотим сохранять это число и после того как юзер ушел из приложения и вернулся в него. Нам не нужно считать секудны в течение сессии, а все время проведенное в приложении в активном состоянии.

Для начала сделаем самое простое: добавим 2 метода в активити

```
override fun onResume() {
    super.onResume()
    viewModel.resumeCounting()
}

override fun onPause() {
    super.onPause()
    viewModel.pauseCounting()
}
```

Весь остальной код вроде работал и до этого, так что не трогаем. Теперь перейдем в вм и немного скорректируем наш код там

Просто пернесем старт счета у модели из инита в резюм. Также нам нужен метод остановки счета. Просто вызовем его когда юзер уже не смотрит в приложение. Да, самое крутое в MVVM что так легко и просто менять код. Если у вас поменялось что-то в одном месте, то не придется полностью переписывать все и вся, а лишь те классы и те участки где затронуло.

```
fun init(textObservable: TextObservable) {
    this.textObservable = textObservable
}

fun clear() {
    textObservable = null
}

fun resumeCounting() {
    model.start(textCallback)
}

fun pauseCounting() {
    model.stop()
}
```

На самом деле мы бы могли написать всего 2 метода и заниматься очисткой и установкой наблюдателя перед стартом и стопом у модели. Но пусть пока останется так. Если хотите можете переписать. Не так критично.

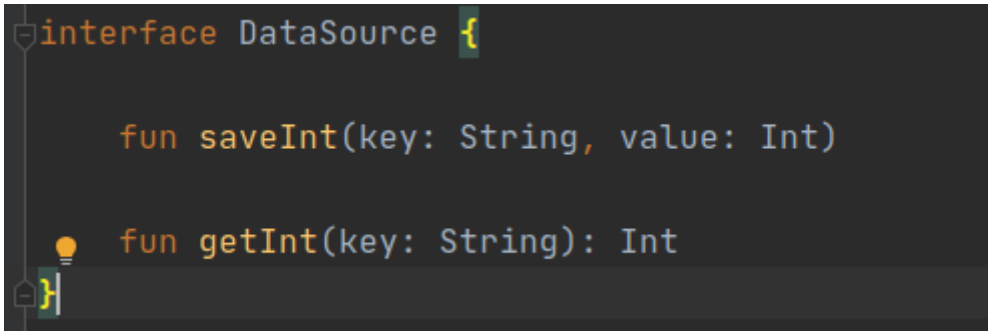
Теперь дойдем до самого интересного – модель.

```
fun start(textCallback: TextCallback) {
    callback = textCallback
    timer = Timer()
    timer?.scheduleAtFixedRate(timerTask, delay: 0, period: 1000)
}

fun stop() {
    timer?.cancel()
    timer = null
}
```

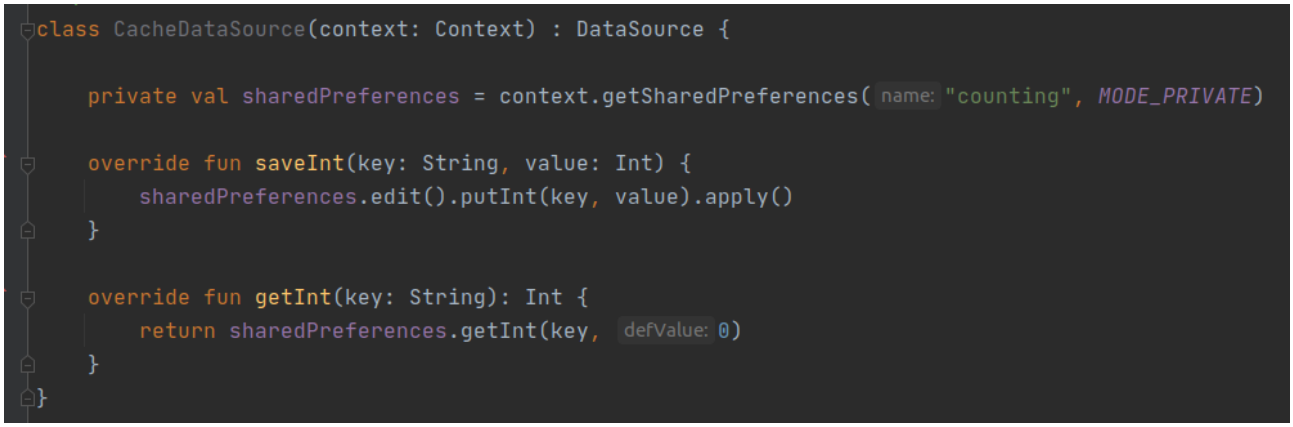
Пока все просто – там где останавливаем счетчик просто отменяем таймер и зануляем его. Почему? А потому что мы не уверены что мы вернемся обратно. Но осталась одна маленькая деталь – мы не сохраняем переменную счета. Она каждый раз у нас будет 0. Значит в метод stop нужно добавить код который бы сохранял нашу переменную и также нужно ее читать из памяти перед тем как начинать счет. И здесь мы прервемся на секунду и поговорим о том, как можно сохранять простые данные в андроид.

Для этого есть простая штука под названием SharedPreferences. Туда можно класть примитивы в виде ключ значение. С его помощью мы можем создать файл в приложении и в него можно писать и читать из него данные. Так же этот файл можно сделать приватным, т.е. не допускать видимости извне. Давайте напишем класс обертку для SharedPreferences чтобы можно было заменять на другую реализацию при тестировании. В отдельной лекции поговорим об этом принципе SOLID.



Назовем интерфейс DataSource – источник данных. Завтра вы сможете поменять реализацию на какую захотите и там например будет создаваться файл в директории с нужным вам именем или сохранение в облако.

Напишем реализацию с SharedPreferences



Так как нам нужен контекст для этого мы передадим его в конструктор и сразу же проинициализируем sharedPreferences. Есть метод который требует имя файла и режим. Пусть мы будем хранить наш счетчик в отдельном файле и назовем его просто “counting”. Режим приватный потому что мы не планируем делиться этой информацией с кем-либо.

Теперь – сохраняем число по ключу. Для этого нам нужно начать редактирование файла и положить число по ключу и в конце вызвать метод apply. Вы скорей всего знаете что есть еще метод commit и можно на офф.сайте прочитать разницу. Так как мы пишем в файл и он хмл вида то это может занимать время. Apply() означает применить изменения сразу и после уже асинхронно записать в файл и не блокировать код. Этот метод рекомендуется использовать вместо commit. Предлагаю вам самостоятельно в этом разобраться. Прочитайте джавадоки над методами исходников и погуглите немного.

Метод для чтения довольно простой – получить значение по ключу или вернуть дефолтное если такового нет. А у нас при первом чтении конечно же такового не будет. Поэтому дефолтным ставим значение 0. И теперь давайте уже применим этот класс в нашей модели.

```
class Model(private val dataSource: DataSource) {

    private var timer: Timer? = null
    private val timerTask
        get() = object : TimerTask() {
            override fun run() {
                count++
                callback?.updateText(count.toString())
            }
        }
    private var callback: TextCallback? = null
    private var count = -1

    fun start(textCallback: TextCallback) {
        callback = textCallback
        Log.d(TAG, msg: "start: count is $count")
        if (count < 0)
            count = dataSource.getInt(COUNTER_KEY)
        Log.d(TAG, msg: "started with count $count")
        timer = Timer()
        timer?.scheduleAtFixedRate(timerTask, delay: 0, period: 1000)
    }

    fun stop() {
        Log.d(TAG, msg: "stop with count: $count")
        dataSource.saveInt(COUNTER_KEY, count)
        timer?.cancel()
        timer = null
    }

    companion object {
        private const val COUNTER_KEY = "counterKey"
        private const val TAG = "uniqueCounterTag"
    }
```

Мы не будем каждый раз при продолжении счета брать значение из файла потому что возможно у нас переменная еще хранит значение. Если же мы первый раз стартуем метод и значение меньше нуля, то тогда проинициализируем его. Можете поставить логи и

посмотреть что и как. Кстати, для хранения констант рекомендуется писать компаньон обьект, потому что в котлине нет слова статик. Private static final String COUNTER_KEY.

И самое главное, нужно поправить инициализацию нашей модели.

```
class MyApplication : Application() {

    lateinit var viewModel: ViewModel

    override fun onCreate() {
        super.onCreate()
        viewModel = ViewModel(Model(CacheDataSource( context: this)))
    }
}
```

Суть в том, что Application класс является наследником Context. Поэтому очень удобно передавать его прямо в инициализации вм.

Итак, я залогирую методы и посмотрим как все работает. Сначала запустим приложение и потом уже свернем развернем с задержкой в пару секунд. После убьем приложение и откроем снова с рабочего стола.

```
2021-06-08 14:59:12.920 23871-23871/com.github.johnnysc.easymvvm D/uniqueCounterTag: start: count is -1
2021-06-08 14:59:12.920 23871-23871/com.github.johnnysc.easymvvm D/uniqueCounterTag: started with count 0
2021-06-08 14:59:17.174 23871-23871/com.github.johnnysc.easymvvm D/uniqueCounterTag: stop with count: 5
2021-06-08 14:59:27.724 23871-23871/com.github.johnnysc.easymvvm D/uniqueCounterTag: start: count is 5
2021-06-08 14:59:27.724 23871-23871/com.github.johnnysc.easymvvm D/uniqueCounterTag: started with count 5
2021-06-08 14:59:37.384 23871-23871/com.github.johnnysc.easymvvm D/uniqueCounterTag: stop with count: 15
2021-06-08 14:59:43.755 23871-23871/com.github.johnnysc.easymvvm D/uniqueCounterTag: start: count is 15
2021-06-08 14:59:43.755 23871-23871/com.github.johnnysc.easymvvm D/uniqueCounterTag: started with count 15
2021-06-08 14:59:50.191 23871-23871/com.github.johnnysc.easymvvm D/uniqueCounterTag: stop with count: 22
```

И убиваем приложение и открываем заново

```
2021-06-08 15:00:19.070 23935-23935/com.github.johnnysc.easymvvm D/uniqueCounterTag: start: count is -1
2021-06-08 15:00:19.070 23935-23935/com.github.johnnysc.easymvvm D/uniqueCounterTag: started with count 22
```

Как видите все правильно работает.

В следующей лекции мы напишем юнит тесты для класса Model и посмотрим как это легко и просто проверять логику не запуская приложение.