Практическая работа 12.1. Live Data

Вашему коду часто приходится реагировать на изменения значений свойств. Например, если свойство модели представления меняет значение, фрагмент может отреагировать обновлением своих представлений или переходом. Но как фрагмент узнает о том, что свойство было обновлено? В этой практической работе вы узнаете о `Live Data`: механизме оповещения заинтересованных сторон о каких-либо изменениях. Вы узнаете о `MutableLiveData` и о том, как заставить ваш фрагмент наблюдать за изменениями свойств этого типа. Мы покажем, как тип `LiveData` помогает обеспечить целостность данных вашего приложения. Вскоре вы будете писать приложения, которые реагируют на происходящее быстрее, чем когда-либо.

Снова о приложении `Guessing Game`

Мы построили приложение `Guessing Game`, в котором пользователь пытается отгадать буквы, входящие в загаданное слово. Когда пользователь отгадывает все буквы или у него кончаются жизни, игра завершается. Чтобы код фрагмента не разрастался, а состояние приложения не терялось при повороте экрана устройства, мы использовали `модели представлений` для игровой логики и данных приложения. `GameFragment` использует `GameViewModel` для своей логики и данных, а `ResultViewModel` хранит результат игры, необходимый для `ResultFragment`:

Когда каждый фрагмент отображается на экране или пользователь вводит предположение, фрагменты получают новейшие значения от модели представления и отображают их на экране. Такое решение работает, но у него есть ряд недостатков.

Фрагменты решают, когда обновлять представления

Недостаток такого решения состоит в том, что каждый фрагмент решает, когда следует получить новейшие значения свойств из модели представления и обновить ее представления. Однако эти значения изменяются не всегда. Например, если пользователь вводит правильное предположение, `GameFragment` обновляет выводимый текст с количеством оставшихся жизней и ошибочных предположений, несмотря на то что эти значения не изменялись.

Пусть модель представления сама сообщает об изменении значений

Возможно альтернативное решение: пусть `GameViewModel` сообщает `GameFragment` об изменении каждого из своих свойств. Если фрагмент будет оповещаться об этих изменениях, он уже не сможет самостоятельно решать, когда следует получить новейшие значения свойств от модели представления и обновить ее представления. Вместо этого представления будут обновляться только тогда, когда фрагмент получит оповещения об изменении используемых свойств.

Мы реализуем это изменение в приложении `Guessing Game` при помощи библиотеки `Android Live Data`, которая является частью Android Jetpack. `Live Data` позволяет модели представления оповещать заинтересованные стороны (например, фрагменты и активности) об обновлении значений свойств. Они могут отреагировать на изменения обновлением представлений или вызовом других методов. В оставшейся части этой главы вы научитесь использовать `Live Data`. Но сначала рассмотрим основные этапы обновления приложения.

Что нам предстоит сделать

Основные этапы построения приложения:

1. Перевод приложения `Guessing Game` на использование `Live Data`. Мы обновим `GameViewModel`, чтобы для свойств `livesLeft`, `incorrectGuesses` и `secretWordDisplay` использовался механизм `Live Data`. Затем мы позаботимся о том, чтобы фрагмент `GameFragment` обновлял свои представления при изменении значений этих свойств.

2. Защита свойств и методов `GameViewModel`. Мы ограничим доступ к свойствам `GameViewModel`, чтобы они могли обновляться только из `GameViewModel`. Также проследим за тем, чтобы фрагменту `GameFragment` были доступны только методы, необходимые для выполнения его работы.

3. Добавление свойства `gameOver`. Для принятия решения о том, завершена ли игра, класс `GameViewModel` будет использовать новое свойство `gameOver`. `GameFragment` будет переходить к `ResultFragment` при изменении значения этого свойства.

Добавление зависимости для Live Data в файл build.gradle приложения

Так как мы собираемся использовать библиотеку `Live Data`, начнем с добавления зависимости `Live Data` в файл `build.gradle` приложения. Откройте проект приложения `Guessing Game` (если это не было сделано ранее), откройте файл *GuessingGame/app/build.gradle* и добавьте следующую строку в раздел `dependencies`:

```
dependencies {
 ...
 implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.3.1'
 ...
}
```

Синхронизируйте изменения, когда среда предложит вам это сделать.

GameViewModel и GameFragment должны использовать Live Data

Мы хотим использовать `Live Data` в приложении `Guessing Game`, чтобы объект `GameViewModel` оповещал `GameFragment` об изменении значений свойств. Тогда `GameFragment` сможет реагировать на эти изменения. Эта задача будет решаться в два этапа:

1. Определение изменений свойств `GameViewModel`, о которых должен знать `GameFragment`.
2. Определение того, как фрагмент `GameFragment` должен реагировать на каждое изменение.

Пока сосредоточимся на изменениях кода `GameViewModel`.

Какие свойства модели представления должны использовать `Live Data`?

`GameViewModel` включает три свойства: `secretWordDisplay`, `incorrectGuesses` и `livesLeft`, которые `GameFragment` использует для обновления своих представлений. Мы укажем, что эти три свойства используют `Live Data`, чтобы `GameFragment` оповещался об изменении значений их свойств. Чтобы указать, что свойство использует `Live Data`, следует изменить его тип на `MutableLiveData<Type>`, где `Type` — тип данных, которые должны храниться в свойстве. Например, свойство `livesLeft` в настоящее время определяется в коде с типом Int:

```
var livesLeft = 8
```

Чтобы свойство использовало механизм `Live Data`, замените его тип на `MutableLiveData<Int>`, чтобы оно выглядело так:

```
var livesLeft = MutableLiveData<Int>(8)
```

Эта запись означает, что `livesLeft` теперь относится к типу `MutableLiveData<Int>` и имеет исходное значение 8. Аналогичным образом можно определить свойства `incorrectGuesses` и `secretWordDisplay`, для чего используется код следующего вида:

```
var incorrectGuesses = MutableLiveData<String>("")
var secretWordDisplay = MutableLiveData<String>()
```

Здесь каждому свойству назначается тип `MutableLiveData<String>`. Свойству `incorrectGuesses` присваивается значение "", тогда как значение `secretWordDisplay` будет задаваться в init-блоке `GameViewModel`. Так определяются свойства `Live Data`. На следующем шаге мы займемся обновлением их значений.

Объекты Live Data используют свойство value

При использовании свойств `MutableLiveData` вы обновляете их значения при помощи свойства с именем `value`. Например, для обновления свойства `secretWordDisplay` возвращаемым значением метода `deriveSecretWordDisplay()` не используется код следующего вида:

```
secretWordDisplay = deriveSecretWordDisplay()
```

как это делалось ранее. Вместо этого используется следующий код:

```
secretWordDisplay.value = deriveSecretWordDisplay()
```

Изменение значения свойства именно таким способом важно, потому что при этом заинтересованные стороны — в данном случае `GameFragment` — оповещаются о любых изменениях. Каждый раз, когда

значение свойства `value` у `secretWordDisplay` обновляется, `GameFragment` оповещается об этих изменениях и может отреагировать обновлением своих представлений.

Свойство value может содержать null

При использовании `Live Data` необходимо учитывать еще одно обстоятельство: тип значения допускает null. Это означает, что при использовании значений `Live Data` в коде необходимо выполнять проверки null-безопасности, в противном случае ваш код компилироваться не будет. Например, свойство `livesLeft` определяется следующим кодом:

```
var livesLeft = MutableLiveData<Int>(8)
```

Свойство имеет тип `MutableLiveData<Int>`, поэтому его свойство `value` может принимать значение Int или содержать null. Так как свойство value может содержать null, для уменьшения его на 1 не удастся использовать следующий код:

```
livesLeft.value--
```

Вместо этого приходится использовать команду:

```
livesLeft.value = livesLeft.value?.minus(1)
```

которая вычитает 1 из value при условии, что значение отлично от null. Аналогичным образом следующий метод `isLost()` не компилируется, потому что liveLeft.value может содержать null:

```
fun isLost() = livesLeft.value <= 0
```

Однако его можно изменить с использованием «элвис-оператора» языка Kotlin:

```
fun isLost() = livesLeft.value ?: 0 <= 0
```

Свойства Live Data могут определяться с val.

Как вы уже знаете, ключевые слова `val` и `var` используются в Kotlin для определения того, возможно ли для свойства присваивание ссылки на новый объект. Когда мы изначально определяли каждое свойство, мы использовали `var`, чтобы его можно было обновить. Например, для свойства `livesLeft` используется следующее определение:

```
var livesLeft = 8
```

которое инициализирует `livesLeft` объектом Int со значением 8. Каждый раз, когда пользователь делает ошибочное предположение, мы уменьшаем `livesLeft` на 1, что дает ссылку на новый объект Int:

```
livesLeft--
```

С `Live Data` вы обновляете свойство `value` существующего объекта, вместо того чтобы заменять его другим объектом, и заинтересованные стороны оповещаются об этом изменении. Так как объект более не заменяется, свойство можно определить с `val` вместо `var`, как в следующем примере:

```
val livesLeft = MutableLiveData<Int>(8)
```

Полный код GameViewModel.kt

Теперь вы знаете, как работает `Live Data`. Давайте обновим класс `GameViewModel`, чтобы все свойства `livesLeft`, `incorrectGuesses` и `secretWordDisplay` использовали `Live Data`.

Ниже приведен полный код `GameViewModel.kt`; обновите свою версию кода:

```
package com.hfad.guessinggame
import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
class GameViewModel : ViewModel() {
 val words = listOf("Android", "Activity", "Fragment")
 val secretWord = words.random().uppercase()
 val secretWordDisplay = "" MutableLiveData<String>()
 var correctGuesses = ""
 val incorrectGuesses = MutableLiveData<String>("")
 val livesLeft = MutableLiveData<Int>(8)
 init {
 secretWordDisplay.value = deriveSecretWordDisplay()
 }
 fun deriveSecretWordDisplay() : String {
 var display = ""
 secretWord.forEach {
 display += checkLetter(it.toString())
 }
 return display
 }
 fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
 true -> str
 false -> "_"
 }
 fun makeGuess(guess: String) {
 if (guess.length == 1) {
```

```
    if (secretWord.contains(guess)) {
    correctGuesses += guess
    secretWordDisplay.value = deriveSecretWordDisplay()
    } else {
    incorrectGuesses.value += "$guess "

    livesLeft.value = livesLeft.value?.minus(1)
    }
    }
    }
    fun isWon() = secretWord.equals(secretWordDisplay.value, true)
    fun isLost() = livesLeft.value ?: 0 <= 0
    fun wonLostMessage() : String {
    var message = ""
    if (isWon()) message = "You won!"
    else if (isLost()) message = "You lost!"
    message += " The word was $secretWord."
    return message
    }
   }
```

И это все, что было необходимо сделать для `GameViewModel`. На следующем шаге мы заставим `GameFragment` реагировать на обновление свойств `livesLeft`, `incorrectGuesses` и `secretWordDisplay`.

Фрагмент наблюдает за свойствами модели предтавления и реагирует на изменения

Чтобы фрагмент реагировал на изменения `value` в свойстве `MutableLiveData` модели представления, следует вызвать метод `observe()` свойства. Например, если добавить в `GameFragment` следующий код, он будет наблюдать за свойством `livesLeft` модели представления и выполнять заданное действие при его изменении:

```
viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
 //Код, использующий новое значение
})
```

Как видите, в этом коде методу `observe()` передаются аргументы `viewLifecycleOwner` и `Observer`.

`viewLifecycleOwner` относится к жизненному циклу представлений фрагмента. Он привязан к промежутку времени, в котором фрагмент имеет доступ к своему пользовательскому интерфейсу: от момента его создания в методе `onCreateView()` фрагмента до его уничтожения и вызова метода `onDestroyView()`.

`Observer` — класс, который может получать данные `Live Data`. Он связан с `viewLifecycleOwner`, и поэтому он активен (и может получать оповещения `Live Data`), только когда фрагмент имеет доступ к своим представлениям. Если значение свойства `Live Data` зменяется, когда фрагмент не имеет доступа к своему пользовательскому интерфейсу наблюдатель не получит оповещения, а фрагмент не

среагирует. Тем самым предотвращаются попытки фрагмента обновить представления в то время, когда они недоступны, что может привести к фатальному сбою приложения.

Класс `Observer` получает параметр с лямбда-выражением, которое определяет, как должно использоваться новое значение свойства. Например, в приложении `Guessing Game` фрагмент `GameFragment` должен обновлять текст `lives` при каждом обновлении свойства `livesLeft` модели представления. Для этого можно воспользоваться следующим кодом:

```
viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
 binding.lives.text = "You have $newValue lives left"
})
```

Вот и все, что необходимо знать для того, чтобы фрагмент `GameFragment` обновлял свои представления при изменении значений свойств модели представления. Давайте внесем изменения в код.

Полный код GameFragment.kt

Ниже приведен обновленный код `GameFragment`; включите в файл `GameFragment.kt` изменения, выделенные жирным шрифтом:

```
package com.hfad.guessinggame
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.Observer
class GameFragment : Fragment() {
 private var _binding: FragmentGameBinding? = null
 private val binding get() = _binding!!
 lateinit var viewModel: GameViewModel
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 _binding = FragmentGameBinding.inflate(inflater, container, false)
 val view = binding.root
 viewModel = ViewModelProvider(this).get(GameViewModel::class.java)

 viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
 binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
 })
 viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
 binding.lives.text = "You have $newValue lives left"
 })
 viewModel.secretWordDisplay.observe(viewLifecycleOwner, Observer { newValue ->
```

```
    binding.word.text = newValue
    })
    binding.guessButton.setOnClickListener() {
    viewModel.makeGuess(binding.guess.text.toString().uppercase())
    binding.guess.text = null

    if (viewModel.isWon() || viewModel.isLost()) {
    val action = GameFragmentDirections
    .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
    view.findNavController().navigate(action)
    }
    }
    return view
    }
    override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
    }
    }
```

И это все изменения, которые необходимо внести в `GameFragment`, чтобы представления обновлялись каждый раз, когда будут обнаружены изменения в свойствах `incorrectGuesses`, `livesLeft` и `secretWordDisplay`. Мы реализовали поддержку `Live Data` в приложении `Guessing Game`. Давайте посмотрим, что происходит во время выполнения приложения.

Модели ResultViewModel не нужно использовать Live Data, поэтому и обновлять ее не придется

Как вы помните, `ResultViewModel` содержит одно свойство (с именем `result`), значение которого задается при создании модели представления. Код выглядит примерно так:

```
class ResultViewModel(finalResult: String) : ViewModel(){
 val result = finalResult
}
```

Как видите, `result` определяется с ключевым словом `val`, поэтому после инициализации обновить ее другим значением не удастся. `ResultFragment` не нуждается в оповещениях об изменениях `result`, потому что после присваивания `result` не может измениться. Заставлять `ResultFragment` реагировать на изменения не нужно, потому что изменений быть не может.

Разберемся, что происходит при выполнении кода, затем проведем тест-драйв приложения.

Что происходит при выполнении приложения

При выполнении приложения происходят следующие события:

1. `GameFragment` запрашивает у класса `ViewModelProvider` экземпляр `GameViewModel`. Объект `GameViewModel` инициализируется, и задаются значения трех свойств `MutableLiveData` — `livesLeft`, `incorrectGuesses` и `secretWordDisplay`.

2. `GameFragment` наблюдает за свойствами `livesLeft`, `incorrectGuesses` и `secretWordDisplay` объекта `GameViewModel`.

3. `GameFragment` обновляет свои представления значениями свойств, за которыми он наблюдает.

4. Когда пользователь вводит правильную букву, значение `secretWordDisplay` обновляется и новое значение передается `GameFragment`. `GameFragment` реагирует обновлением экранного представления загаданного слова.

5. Когда пользователь вводит ошибочное предположение, значения `incorrectGuesses` и `livesLeft` обновляются и передаются `GameFragment`. `GameFragment` реагирует обновлением своих представлений.

6. Когда `isWon()` или `isLost()` возвращает true, `GameFragment` переходит к `ResultFragment` и передает `result`. `ResultFragment` выводит результат.

При запуске приложения, как и прежде, отображается фрагмент `GameFragment`. При вводе правильного предположения экранное представление загаданного слова обновляется. При вводе ошибочного предположения обновляется количество оставшихся жизней, а предположение включается в выводимое количество неправильных предположений. Если пользователь правильно угадал все буквы или потерял все жизни, приложение переходит к фрагменту `ResultFragment`, который отображает результат.

Игра работает так же, как прежде, но в ее внутренней реализации используется `Live Data`.

Фрагменты могут обновлять свойства GameViewModel

До настоящего момента мы обновили коды `GameViewModel` и `GameFragment`, чтобы в них использовался механизм `Live Data`. Кажый раз, когда значение свойства `MutableLiveData` из `GameViewModel` обновляется, `GameFragment` реагирует обновлением своих представлений. Тем не менее в этом коде кроется небольшая проблема. `GameFragment` обладает полным доступом к свойствам и методам `GameViewModel`, поэтому при желании фрагмент может использовать их некорректно. Ничто не помешает фрагменту, скажем, присвоить свойству `livesLeft` значение `100`, чтобы у пользователя было намного больше предположений и он легко выигрывал каждую партию. Для предотвращения этой проблемы мы ограничим прямой доступ к свойствам `GameViewModel`, чтобы они могли обновляться только методами модели представления.

Приватные свойства

Чтобы защитить свойства `GameViewModel`, мы пометим их ключевым словом `private`, чтобы их значения могли обновляться только кодом `GameViewModel`. После этого мы предоставим доступ к версии, доступной только для чтения, каждого свойства `MutableLiveData`, за которым должен наблюдать `GameFragment`. Вместо того чтобы определять свойство `livesLeft` кодом следующего вида:

```
val livesLeft = MutableLiveData<Int>(8)
```

мы будем использовать следующую запись:

```
private val _livesLeft = MutableLiveData<Int>(8)
val livesLeft: LiveData<Int>
 get() = _livesLeft
```

Свойство `_livesLeft` содержит ссылку на объект `MutableLiveData`. Для `GameFragment` это свойство недоступно, потому что оно помечено ключевым словом `private`. Однако `GameFragment` может обратиться к значению этого свойства через метод чтения `livesLeft`. Свойство `livesLeft` относится к типу `LiveData`, который имеет много общего с `MutableLiveData`, за исключением того, что оно не может использоваться для обновления свойства `value` используемого объекта: `GamesFragment` может читать значение, но не может обновить его. При такой структуре кода private-свойство иногда называется резервным (backing). Оно содержит ссылку на объект, к которому другие классы могут обращаться только через другое свойство.

Обновим код `GameViewModel`.

Полный код GameViewModel.kt

Обновим код `GameViewModel`, чтобы в нем использовались резервные свойства для ограничения прямого доступа к свойствам `Live Data`. Также мы пометим ключевым словом `private` все свойства и методы, которые не должны использоваться `GameFragment`.

Ниже приведен полный код `GameViewModel.kt`; обновите свою версию кода:

```
package com.hfad.guessinggame
import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.LiveData
class GameViewModel : ViewModel() {
 private val words = listOf("Android", "Activity", "Fragment")
 private val secretWord = words.random().uppercase()
 private val _secretWordDisplay = MutableLiveData<String>()
 val secretWordDisplay: LiveData<String>
 get() = _secretWordDisplay
 private var correctGuesses = ""
 private val _incorrectGuesses = MutableLiveData<String>("")
 val incorrectGuesses: LiveData<String>
 get() = _incorrectGuesses
 private val _livesLeft = MutableLiveData<Int>(8)
 val livesLeft: LiveData<Int>
 get() = _livesLeft
 init {
 _secretWordDisplay.value = deriveSecretWordDisplay()
 }
 private fun deriveSecretWordDisplay() : String {
 var display = ""
 secretWord.forEach {
```

```
    display += checkLetter(it.toString())
    }
    return display
    }
    private fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
    true -> str
    false -> "_"
    }
    fun makeGuess(guess: String) {
    if (guess.length == 1) {
    if (secretWord.contains(guess)) {
    correctGuesses += guess
    _secretWordDisplay.value = deriveSecretWordDisplay()
    } else {
    _incorrectGuesses.value += "$guess "
    _livesLeft.value = _livesLeft.value?.minus(1)
    }
    }
    }
    fun isWon() = secretWord.equals(secretWordDisplay.value, true)
    fun isLost() = livesLeft.value ?: 0 <= 0
    fun wonLostMessage() : String {
    var message = ""
    if (isWon()) message = "You won!"
    else if (isLost()) message = "You lost!"
    message += " The word was $secretWord."
    return message
    }
  }
```

Посмотрим, что происходит при выполнении кода.

Что происходит при выполнении приложения

Во время выполнения приложения происходят следующие события:

1. `GameFragment` запрашивает у класса `ViewModelProvider` экземпляр `GameViewModel`.
2. Инициализируются свойства `GameViewModel`. `livesLeft`, `incorrectGuesses` и `secretWordDisplay` — свойства `Live Data`, ссылающиеся на те же объекты, что и их резервные свойства `MutableLiveData`.
3. `GameFragment` наблюдает за свойствами `livesLeft`, `incorrectGuesses` и `secretWordDisplay`. `GameFragment` не может обновлять эти свойства, но реагирует, когда `GameViewModel` обновляет какие-либо из резервных свойств, потому что они ссылаются на один и тот же используемый объект.
4. `GameFragment` продолжает реагировать на изменения значений до того момента, когда `isWon()` или `isLost()` вернет true. `GameFragment` переходит к фрагменту `ResultFragment` и передает ему результат. `ResultFragment` отображает результат на экране.

Когда вы запускаете приложение, оно работает так же, как прежде. Но в новой версии свойства `MutableLiveData` класса `GameViewModel` защищены — доступ к ним со стороны `GameFragment` ограничен.

GameFragment содержит игровую логику

В текущей версии приложения `GameFragment` решает, завершена ли игра, вызывая методы `isWon()` и `isLost()` класса `GameViewModel` после каждого предположения пользователя. Если один из этих методов возвращает true, `GameFragment` переходит к `ResultFragment` и передает результат. Текущая версия кода выглядит так:

```
binding.guessButton.setOnClickListener() {
 viewModel.makeGuess(binding.guess.text.toString().uppercase())
 binding.guess.text = null
 if (viewModel.isWon() || viewModel.isLost()) {
 val action = GameFragmentDirections
 .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
 view.findNavController().navigate(action)
 }
 }
```

Недостаток такого подхода заключается в том, что решение о завершении игры принимает `GameFragment` вместо `GameViewModel`. Так как определение завершения игры является игровым решением, за него должен отвечать класс `GameViewModel`, а не `GameFragment`.

Принятие решения о завершении игры в GameViewModel

Для устранения этого недостатка мы добавим в `GameViewModel` свойство `MutableLiveData<Boolean>` с именем `_gameOver`, для обращения к значению которого будет использоваться свойство `LiveData` с именем `gameOver`. Этому свойству присваивается значение true, когда пользователь выигрывает или проигрывает игру. Когда такое происходит, `GameFragment` реагирует на это переходом к `ResultFragment`.

Полный код GameViewModel.kt

Необходимо добавить в `GameViewModel` свойство `gameOver`, а также резервное свойство `_gameOver`. Метод `makeGuess()` присваивает ему true, если пользователь угадал все буквы загаданного слова либо у него кончились жизни.

Ниже приведен полный код `GameViewModel`; обновите файл `GameViewModel.kt`:

```
package com.hfad.guessinggame
import androidx.lifecycle.ViewModel
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.LiveData
class GameViewModel : ViewModel() {
 private val words = listOf("Android", "Activity", "Fragment")
 private val secretWord = words.random().uppercase()
 private val _secretWordDisplay = MutableLiveData<String>()
 val secretWordDisplay: LiveData<String>
```

```
    get() = _secretWordDisplay
    private var correctGuesses = ""
    private val _incorrectGuesses = MutableLiveData<String>("")
    val incorrectGuesses: LiveData<String>
    get() = _incorrectGuesses
    private val _livesLeft = MutableLiveData<Int>(8)
    val livesLeft: LiveData<Int>
    get() = _livesLeft
    private val _gameOver = MutableLiveData<Boolean>(false)
    val gameOver: LiveData<Boolean>
    get() = _gameOver
    init {
    _secretWordDisplay.value = deriveSecretWordDisplay()
    }
    private fun deriveSecretWordDisplay() : String {
    var display = ""
    secretWord.forEach {
    display += checkLetter(it.toString())
    }
    return display
    }
    private fun checkLetter(str: String) = when (correctGuesses.contains(str)) {
    true -> str
    false -> "_"
    }
    fun makeGuess(guess: String) {
    if (guess.length == 1) {
    if (secretWord.contains(guess)) {
    correctGuesses += guess
    _secretWordDisplay.value = deriveSecretWordDisplay()
    } else {
    _incorrectGuesses.value += "$guess "
    _livesLeft.value = _livesLeft.value?.minus(1)
    }
    if (isWon() || isLost()) _gameOver.value = true
    }
    }
    private fun isWon() = secretWord.equals(secretWordDisplay.value, true)
    private fun isLost() = livesLeft.value ?: 0 <= 0
    fun wonLostMessage() : String {
    var message = ""
    if (isWon()) message = "You won!"
    else if (isLost()) message = "You lost!"
    message += " The word was $secretWord."
    return message
    }
   }
```

Отслеживание нового свойства в GameFragment

После того как свойство `gameOver` будет добавлено в `GameViewModel`, необходимо заставить `GameFragment` реагировать на его обновления. Для этого фрагмент будет наблюдать за свойством,

чтобы когда оно примет значение true, фрагмент переходил к `ResultFragment`.

Ниже приведен код GameFragment; обновите файл `GameFragment.kt`:

```
package com.hfad.guessinggame
import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.hfad.guessinggame.databinding.FragmentGameBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.Observer
class GameFragment : Fragment() {
 private var _binding: FragmentGameBinding? = null
 private val binding get() = _binding!!
 lateinit var viewModel: GameViewModel
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 _binding = FragmentGameBinding.inflate(inflater, container, false)
 val view = binding.root
 viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
 viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
 binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
 })
 viewModel.livesLeft.observe(viewLifecycleOwner, Observer { newValue ->
 binding.lives.text = "You have $newValue lives left"
 })
 viewModel.secretWordDisplay.observe(viewLifecycleOwner, Observer { newValue ->
 binding.word.text = newValue
 })
 viewModel.gameOver.observe(viewLifecycleOwner, Observer { newValue ->
 if (newValue) {
 val action = GameFragmentDirections
 .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
 view.findNavController().navigate(action)
 }
 })
 binding.guessButton.setOnClickListener() {
 viewModel.makeGuess(binding.guess.text.toString().uppercase())
 binding.guess.text = null
 }
 return view
 }
 override fun onDestroyView() {
 super.onDestroyView()
 _binding = null
 }
}
```

Вот и все! Давайте разберемся, что происходит во время выполнения приложения.

Что происходит при выполнении приложения

Во время выполнения приложения происходят следующие события:

1. GameFragment запрашивает у класса `ViewModelProvider` экземпляр `GameViewModel`.
2. Инициализируются свойства `GameViewModel`. Свойства `_gameOver` и `gameOver` ссылаются на объект `MutableLiveData<Boolean>`, у которого свойству `value` присваивается значение false.
3. `GameFragment` наблюдает за свойством `gameOver` класса `GameViewModel`. `GameFragment` не может обновить объект `MutableLiveData`, ссылка на который хранится в свойстве `gameOver`, но может отреагировать на изменение его значения.
4. Каждый раз, когда вызывается метод `makeGuess()` класса `GameViewModel`, он проверяет, был ли получен результат true при вызове `isWon()` или `isLost()`. Если один из двух результатов равен true, то свойству `_gameOver` присваивается значение true.
5. `GameFragment` замечает, что значение стало равно true, через свойство `gameOver` класса `GameViewModel`. Новое значение передается `GameFragment`.
6. `GameFragment` реагирует переходом к `ResultFragment` с передачей `result`.

Приложение работает так же, как прежде. Однако в новой версии решение о завершении игры принимает `GameViewModel`, а не `GameFragment`. Фрагмент просто наблюдает за свойством `gameOver` модели представления и переходит к `ResultFragment`, когда оно принимает значение true.

Поздравляем! Вы построили приложение, которое использует `Live Data` для реагирования на изменения в момент их возникновения.

Резюме

- Механизм `Live Data` позволяет свойствам модели представления сообщать заинтересованным сторонам (например, фрагментам или активностям) об изменении их значений.
- Чтобы свойство использовало `Live Data`, определите его с ключевым словом `val` и измените его тип на `MutableLiveData<Тип>`, где Тип — тип данных, которые должны храниться в свойстве.
- Свойство `value` объекта `MutableLiveData` используется для хранения его значения. Это свойство допускает null.
- Чтобы фрагмент или активность реагировали на изменения модели представления, вызовите метод observe свойства и укажите, как должно использоваться новое значение.
- Прямой доступ к свойствам модели представления можно ограничить при помощи резервных свойств.
- Тип `Live Data` похож на `MutableLiveData`, но он является неизменяемым. Он используется для предоставления доступа к значению объекта `MutableLiveData` с запретом на изменение его значения.