Практическая работа 1. Построение умных макетов

Возможности макетов не ограничиваются управлением внешним видом вашего приложения. У всех макетов, написанных нами ранее, поведение должно было определяться кодом активности или фрагмента. Но только представьте, что макеты могли бы думать самостоятельно и принимать собственные решения. В этой главе представлен механизм связывания данных: способ наращивания интеллекта ваших макетов. Вы узнаете, как заставить представления получать данные непосредственно от модели представления. Мы воспользуемся связыванием слушателей, чтобы кнопки вызывали свои методы. Вы даже увидите, как одна простая строка кода позволяет реагировать на обновления Live Data. Скоро ваши макеты станут более мощными, чем когда-либо.

Снова к приложению Guessing Game

В двух предыдущих главах мы построили приложение `Guessing Game`, в котором пользователь отгадывает буквы, входящие в секретное слово. Когда пользователь успешно отгадывает все буквы или у него кончаются жизни, игра завершается.

В приложении используются две модели представления (`GameViewModel` и `ResultViewModel`) для хранения игровой логики и данных приложения; они сохраняют свое состояние при повороте экрана устройства. `GameViewModel` используется `GameFragment`, а `ResultViewModel` используется `ResultFragment`:

Первый экран, который видит пользователь, — `GameFragment`. Его взаимодействия с этим экраном составляют суть игры. На этом экране отображается информация (например, количество оставшихся жизней и все ошибочные предположения пользователя), а пользователь получает возможность вводить предположения. После завершения игры приложение переходит к `ResultFragment`. На этом экране выводится информация о том, выиграл или проиграл пользователь и какое слово было загадано.

Фрагменты обновляют представления в своих макетах

Каждый фрагмент приложения отвечает за обновление представлений в своих макетах. Например, `ResultFragment` заполняет свое текстовое представление `won_lost` значением свойства `result` объекта `ResultViewModel`:

Такой подход работает, но у него есть недостаток. Когда фрагменты отвечают за поддержание актуальности представлений, код Kotlin становится сложнее и создает больше проблем с чтением и сопровождением.

Существует ли другой способ?

Представления могут обновлять себя сами

В альтернативном решении используется прием, называемый связыванием данных. При связывании данных представления получают свои значения напрямую от модели представления, так что коду фрагмента уже не приходится заниматься их обновлением. Например, вместо того чтобы использовать код `ResultFragment` для обновления текста в представлении `won_lost`, можно воспользоваться

связыванием данных для того, чтобы представление получало свой текст напрямую из свойства `result` объекта `ResultViewModel`.

В этой главе вы научитесь применять связывание данных на примере приложения `Guessing Game`. Рассмотрим последовательность действий для его реализации.

Что мы собираемся сделать

Ниже описаны основные действия, которые необходимо выполнить для того, чтобы в приложении `Guessing Game` использовалось связывание данных:

1. Реализация связывания данных для `ResultFragment`. Мы обновим фрагмент `ResultFragment`, чтобы в его макете использовалось связывание данных для отображения свойства `result` непосредственно из `ResultViewModel`.

2. Реализация связывания данных для `GameFragment`. Затем мы изменим фрагмент `GameFragment` так, чтобы его макет использовал связывание данных для отображения значений свойств из `GameViewModel`. Заодно мы сделаем так, чтобы макет динамически реагировал на любые обновления данных `Live Data`.

3. Добавление кнопки `Finish Game` в `GameFragment`. Наконец, мы добавим в макет `GameFragment` новую кнопку, щелчок на которой немедленно завершает игру.

Включение связывания данных в файле build.gradle приложения

Как и в случае со связыванием представлений, чтобы вы могли использовать связывание данных в своем представлении, его необходимо явно включить в разделе android файла `build.gradle` приложения. Код включения связывания данных выглядит примерно так:

```
android {
 ...
 buildFeatures {
 dataBinding true
 }
}
```

Мы уже включили связывание представлений в приложении `Guessing Game`, давайте добавим к нему связывание данных. Откройте файл *GuessingGame/app/build.gradle*, обновите раздел `buildFeatures` и приведите его к следующему виду:

```
android {
 ...
 buildFeatures {
 viewBinding true
 dataBinding true
```

```
    }
}
```

Затем выберите вариант `Sync Now`, чтобы синхронизировать это изменение с остальными частями проекта. После того как связывание данных будет включено, реализуем его в `ResultFragment`.

ResultFragment обновляет текст в своем макете

Как вы, возможно, помните, макет `ResultFragment` содержит текстовое представление с идентификатором `won_lost`, которое определяется следующим образом:

```
<LinearLayout ...>
 <TextView
 android:id="@+id/won_lost"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:gravity="center"
 android:textSize="28sp" />
 ...
</LinearLayout>
```

Когда приложение переходит к `ResultFragment`, фрагмент получает значение свойства `result` от своей модели представления и отображает его в текстовом представлении, для чего используется следующий код:

```
binding.wonLost.text = viewModel.result
```

Мы изменим фрагмент и его макет, чтобы в нем использовалось связывание данных. Вместо того чтобы обновлять текстовое представление из кода `ResultFragment`, мы сделаем так, чтобы текстовое представление получало свое значение прямо от модели представления.

Как реализуется связывание данных

Чтобы текстовое представление получало свой текст от `ResultViewModel`, необходимо обновить код фрагмента `ResultFragment` и его макет. Для этого необходимо выполнить следующие действия:

1. Добавление элементов `<layout>` и `<data>` в макет. Элемент `<layout>` необходим для связывания данных, а элемент `<data>` используется для определения переменной связывания данных, которая соединяет макет с моделью представления.

2. Переменной связывания данных в макете присваивается экземпляр модели представления. Это будет сделано в коде фрагмента.

3. Переменная связывания данных используется для обращения к свойствам модели представления. Мы обновим текстовое представление `won_lost`, чтобы оно получало свой текст прямо из модели представления.

Начнем с обновления кода макета.

1. Добавление элементов и

Каждый макет, в котором используется связывание данных, записывается по следующей схеме:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 tools:context=".ResultFragment">
 <data>
 <variable
 name="resultViewModel"
 type="com.hfad.guessinggame.ResultViewModel" />
 </data>
 ...
</layout>
```

В приведенном макете корневым элементом является элемент `<layout>`. Он включает связывание данных в макете и поэтому должен присутствовать во всех макетах, которые должны использовать связывание данных.

В элементе `<layout>` находится элемент `<data>`. В нем задаются любые переменные (определяемые элементом `<variable>`), необходимые для связывания данных, например модель представления, от которой представления макета должны получать свои данные. Так, в приведенном выше примере используется код:

```
<data>
 <variable
 name="resultViewModel"
 type="com.hfad.guessinggame.ResultViewModel" />
</data>
```

для определения переменной связывания данных с именем `resultViewModel`, относящейся к типу `ResultViewModel`. Элемент `<layout>` также включает представление или группу представлений для иерархии представлений макета. Например, если вы хотите отобразить представления в линейном макете, в элемент `<layout>` следует поместить элемент `<LinearLayout>` вместе со всеми представлениями. Через несколько страниц вы увидите, как это делается. После того как вы определите переменную связывания данных, необходимо присвоить ей значение.

2. Присваивание значения переменной связывания данных в макете

Значение переменной связывания данных макета задается в коде Kotlin. На предыдущей странице был приведен код добавления переменной связывания данных (с именем `resultViewModel`) в файл макета `ResultFragment fragment_result.xml`:

```
<data>
 <variable
 name="resultViewModel"
 type="com.hfad.guessinggame.ResultViewModel" />
</data>
```

Переменная имеет тип `ResultViewModel`, и ей должен быть присвоен объект соответствующего типа. Напомним, что метод `onCreateView()` класса `ResultFragment` включает код получения объекта `ResultViewModel`:

```
viewModelFactory = ResultViewModelFactory(result)
viewModel = ViewModelProvider(this,
viewModelFactory).get(ResultViewModel::class.java)
```

Чтобы присвоить эту модель представления переменной связывания данных макета `resultViewModel`, добавим следующую строку в код `onCreateView()`:

```
binding.resultViewModel = viewModel
```

Эта команда использует свойство `binding` фрагмента для того, чтобы присвоить свойству `resultViewModel` макета объект модели представления фрагмента:

После того как вы зададите переменную связывания данных, макет сможет использовать ее для обращения к свойствам и методам модели представления. Давайте посмотрим, как это делается.

3. Использование переменной связывания данных макета для обращения к модели представления

Макет `ResultFragment` определяет текстовое представление `won_lost`, в котором должно выводиться свойство `result` объекта `ResultViewModel`. Его текст задается в коде фрагмента следующим образом:

```
binding.wonLost.text = viewModel.result
```

После определения переменной связывания данных для модели представления в коде макета мы сможем использовать связывание данных для получения текста. Код решения этой задачи выглядит так:

```
<TextView
 android:id="@+id/won_lost"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:gravity="center"
 android:textSize="28sp"
 android:text="@{resultViewModel.result}" />
```

В этом коде переменная `resultViewModel` используется для получения значения свойства `result` модели представления. Затем содержимому текстового представления присваивается значение этого свойства. И это все, что необходимо знать для реализации связывания данных в `ResultFragment`. Полный код `result_fragment.xml` и `ResultFragment.kt` будет приведен через пару страниц.

Полный код fragment_result.xml

Ниже приведен полный код макета `ResultFragment`; обновите файл `fragment_result.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 tools:context=".ResultFragment">
 <data>
 <variable
 name="resultViewModel"
 type="com.hfad.guessinggame.ResultViewModel" />
 </data>
 <LinearLayout

 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical"
 <TextView
 android:id="@+id/won_lost"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:gravity="center"
 android:textSize="28sp"
 android:text="@{resultViewModel.result}" />
 <Button
 android:id="@+id/new_game_button"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="center"
 android:text="Start new game"/>
 </LinearLayout>
</layout>
```

Полный код ResultFragment.kt

Код фрагмента тоже необходимо обновить. Измените содержимое файла `ResultFragment.kt` так, как показано ниже:

```
package com.hfad.guessinggame

class ResultFragment : Fragment() {
 private var _binding: FragmentResultBinding? = null
 private val binding get() = _binding!!
 lateinit var viewModel: ResultViewModel
 lateinit var viewModelFactory: ResultViewModelFactory
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 _binding = FragmentResultBinding.inflate(inflater, container, false)
 val view = binding.root
 val result = ResultFragmentArgs.fromBundle(requireArguments()).result
 viewModelFactory = ResultViewModelFactory(result)
 viewModel = ViewModelProvider(this, viewModelFactory)
 .get(ResultViewModel::class.java)
 binding.resultViewModel = viewModel
 binding.newGameButton.setOnClickListener {
 view.findNavController()
 .navigate(R.id.action_resultFragment_to_gameFragment)
 }
 return view
 }
 override fun onDestroyView() {
 super.onDestroyView()
 _binding = null
 }
}
```

Посмотрим, что происходит во время выполнения приложения.

Что происходит во время выполнения приложения

При выполнении приложения происходят следующие события:

1. Когда пользователь завершает игру, приложение переходит к `ResultFragment`.
2. `ResultFragment` получает ссылку на свой объект `ResultViewModel`.
3. `ResultFragment` присваивает объект `ResultViewModel` переменной связывания данных `resultViewModel` макета. Представления макета теперь могут использовать переменную для обращения к свойствам и методам модели представления.
4. Текстом представления `won_lost` становится значение свойства `result` объекта `ResultViewModel`. Текст отображается на экране устройства.

Приложение работает точно так же, как прежде. Однако в новой версии макет `ResultFragment` получает результат игры непосредственно от `ResultViewModel`.

Мы реализуем связывание данных в `ResultFragment`. Прежде чем применять его в `GameFragment`, стоит поближе присмотреться к элементу `<layout>` и понять, почему он необходим для связывания данных.

Как говорилось ранее, каждый макет, использующий связывание данных, должен содержать корневой элемент `<layout>`:

```
<layout ... >
 ...
</layout>
```

Когда вы включаете связывание данных в файле `build.gradle` приложения, для каждого макета с корневым элементом `<layout>` генерируется класс связывания. Это тот же класс связывания, который генерируется при включении связывания представлений.

В приложении `Guessing Game` файл макета `fragment_result.xml` содержит корневой элемент `<layout>`. Это означает, что его класс связывания `FragmentResultBinding` все равно будет генерироваться связыванием данных, даже если связывание представлений отключено:

Для включения в макет одной или нескольких переменных связывания данных используются элементы `<data>` и `<variable>`:

```
<layout ... >
 <data>
 <variable
 name="variableName"
 type="com.hfad.myapp.ClassName" />
 </data>
 ...
</layout>
```

Каждой переменной в коде Kotlin присваивается объект соответствующего типа. После присваивания представления макета могут использовать переменную связывания данных для обращения к свойствам и методам своего объекта.

GameFragment тоже может использовать связывание данных

На данный момент мы реализовали связывание данных для фрагмента `ResultFragment`, чтобы текстовое представление из этого макета получало свой текст прямо из свойства `result` объекта `ResultViewModel`.

На следующем шаге то же будет сделано с `GameFragment`: мы реализуем связывание данных, чтобы текстовое представление в макете получало свои значения непосредственно из `GameViewModel`:

Для этого необходимо обновить код `GameFragment` и его макет. Начнем с кода макета.

Добавление элементов и в fragment_game.xml

Как и в случае с `ResultFragment`, начнем с добавления элементов `<layout>` и `<data>` в макет `GameFragment`. Эти элементы указывают, что макет использует связывание данных, и определяют переменную связывания данных.

Код выглядит так, как показано ниже; обновите файл `fragment_game.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 tools:context=".GameFragment">
 <data>
 <variable
 name="gameViewModel"
 type="com.hfad.guessinggame.GameViewModel" />
 </data>
 <LinearLayout

 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical"
 android:padding="16dp"
 >
 ...
 </LinearLayout>
</layout>
```

Как видите, в коде определяется переменная связывания данных с именем `gameViewModel` и типом `GameViewModel`. Прежде чем задавать значение этой переменной в коде Kotlin фрагмента, обновим представления макета, чтобы они получали свои значения из модели представления.

Использование переменной связывания данных для задания текста в макете

Как вы узнали ранее, представления могут получать данные напрямую от модели представления с использованием переменной связывания данных макета. Например, текстовое представление word из `fragment_game.xml` может отображать значение свойства `secretWordDisplay` объекта `GameViewModel` следующим кодом:

```
<TextView
 android:id="@+id/word"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="center"
 android:textSize="36sp"
```

```
 android:letterSpacing="0.1"
 android:text="@{gameViewModel.secretWordDisplay}" />
```

В двух представлениях должен выводиться дополнительный текст

Однако текстовые представления `lives` и `incorrect_guesses` не ограничиваются выводом значений свойств: они включают дополнительный текст. Например, в коде `GameFragment` в настоящее время текст представления `incorrectGuesses` задается следующим кодом:

```
binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
```

Таким образом, мы не можем просто использовать свойство `incorrectGuesses` модели представления с использованием следующего кода, так как дополнительный текст в этом случае отображаться не будет:

```
<TextView
 android:id="@+id/incorrect_guesses"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:text="@{gameViewModel.incorrectGuesses}" />
```

Чтобы решить эту проблему, мы воспользуемся строковым форматированием для передачи значения свойства строковому ресурсу. Давайте посмотрим, как это делается.

Снова о строковых ресурсах

Как вы узнали ранее в этой книге, включение строковых ресурсов в файл строковых ресурсов приложения позволяет избежать жестко фиксированного текста. Например, следующий код определяет строковый ресурс с именем `my_string`, который выводит текст «This is a String resource»:

```
<resources>
 ...
 <string name="my_string">This is a String resource</string>
</resources>
```

Строковые ресурсы могут получать аргументы

Также можно определить строковые ресурсы, получающие один или несколько аргументов. Это может быть удобно для выбора более сложного текста. Например, следующий ресурс использует обозначение `%s` для определения позиции, в которой должен быть вставлен переданный строковый аргумент:

```
<string name="hello">Hello, %s.</string>
```

А в этом примере %d сообщает, где должен располагаться строковый аргумент:

```
<string name="messages">You have %d messages.</string>
```

Строковому ресурсу можно передать несколько аргументов, для чего их следует пронумеровать. Например, следующий ресурс получает строку в первом аргументе и число во втором:

```
<string name="welcome">Hello, %1$s. You have %2$d messages.</string>
```

Добавление двух новых строковых ресурсов в файл strings.xml

В приложении Guessing Game будут определены два новых строковых ресурса: для количества оставшихся жизней и для количества ошибочных предположений, сделанных пользователем. Откройте файл strings.xml в папке app/src/main/res/values и добавьте в него следующие два ресурса:

```
<resources>
 ...
 <string name="lives_left">You have %d lives left</string>
 <string name="incorrect_guesses">Incorrect guesses: %s</string>
</resources>
```

Итак, мы определили два строковых ресурса; используем их в макете GameFragment.

Макет может передавать параметры строковым ресурсам

Чтобы использовать два строковых ресурса, которые мы только что создали, необходимо передать каждому ресурсу параметр.

Передача параметров строковым ресурсам из кода макета происходит так:

```
android:text="@{@string/string_name(arg1, arg2)}"
```

где string_name— имя строкового ресурса, а arg1 и arg2– два параметра. arg1 определяет значение первого аргумента строки, а arg2 — второго. В приложении Guessing Game строковый ресурс lives_left определяется следующим образом:

```
<string name="lives_left">You have %d lives left</string>
```

Чтобы использовать его для отображения свойства livesLeft объекта GameViewModel, можно написать следующий код:

```
<TextView
 android:id="@+id/lives"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:text="@{@string/lives_left(gameViewModel.livesLeft)}" />
```

Аналогичным образом определение строкового ресурса `incorrect_guesses` выглядит так:

```
<string name="incorrect_guesses">Incorrect guesses: %s</string>
```

Следовательно, значение свойства `incorrectGuesses` может отображаться следующим кодом:

```
<TextView
 android:id="@+id/incorrect_guesses"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />
```

И это все, что необходимо знать для изменения макета `GameFragment`, чтобы в нем использовалось связывание данных. Давайте посмотрим, как выглядит полный код.

Полный код fragment_game.xml

Ниже приведен полный код макета `GameFragment`; обновите файл `fragment_game.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 tools:context=".GameFragment">
 <data>
 <variable
 name="gameViewModel"
 type="com.hfad.guessinggame.GameViewModel" />
 </data>
<LinearLayout
android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical"
 android:padding="16dp"
 <TextView
 android:id="@+id/word"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:textSize="36sp"
    android:letterSpacing="0.1"
    android:text="@{gameViewModel.secretWordDisplay}" />
    <TextView
    android:id="@+id/lives"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:text="@{@string/lives_left(gameViewModel.livesLeft)}" />
    <TextView
    android:id="@+id/incorrect_guesses"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />
    <EditText
    android:id="@+id/guess"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16sp"
    android:hint="Guess a letter"
    android:inputType="text"
    android:maxLength="1" />
    <Button
    android:id="@+id/guess_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:text="Guess!"/>
    </LinearLayout>
  </layout>
```

И это весь код, который необходимо изменить в файле макета. Но прежде чем проводить тест-драйв приложения, сначала необходимо обновить соответствующий фрагмент кода в `GameFragment.kt`.

Необходимо задать значение переменной gameViewModel

Первое, что необходимо сделать с кодом `GameFragment`, присвоить переменной связывания данных `gameViewModel` макета экземпляр модели представления фрагмента. В настоящее время метод `onCreateView()` фрагмента `GameFragment` включает следующую строку, которая использует провайдер модели представления для получения объекта `GameViewModel`:

```
viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
```

Эту модель представления можно присвоить переменной связывания данных, для чего следует добавить следующую строку:

```
binding.gameViewModel = viewModel
```

После того как переменная `gameViewModel` будет связана с моделью представления подобным образом, представления макета смогут использовать его для получения своих данных.

Также необходимо включить использование Live Data механизмом связывания данных

Помимо присваивания переменной `gameViewModel`, необходимо включить использование `Live Data` в связывании данных макета.

Но в отличие от `ResultViewModel`, код `GameViewModel` использует `Live Data`, чтобы при обновлении значений свойств код `GameFragment` реагировал соответствующим образом. Например, следующий код из файла `GameFragment.kt` включает наблюдение за свойством `incorrectGuesses` и обновляет представление `incorrect_guesses` при обновлении свойства:

```
viewModel.incorrectGuesses.observe(viewLifecycleOwner, Observer { newValue ->
 binding.incorrectGuesses.text = "Incorrect guesses: $newValue"
})
```

Так как теперь в коде используется связывание данных, можно заставить каждое представление реагировать на изменения данных, включив следующую строку в код фрагмента:

```
binding.lifecycleOwner = viewLifecycleOwner
```

Каждый раз, когда в модели представления обновляется значение свойства `Live Data`, макету уже не нужно полагаться на то, что код фрагмента обновит его представления. Макет может отреагировать на любые изменения без каких-либо вмешательств со стороны фрагмента. И это все, что необходимо знать для обновления кода `GameFragment`. Давайте посмотрим, как он выглядит.

Полный код GameFragment.kt

Ниже приведен полный 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)
 binding.gameViewModel = viewModel
 binding.lifecycleOwner = viewLifecycleOwner
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
 }
}
```

И это все изменения, которые необходимо внести в `GameFragment`. Давайте разберемся, что происходит при выполнении этого кода, и проведем тест-драйв приложения.

Что происходит при выполнении приложения

При выполнении приложения происходят следующие события:

1. `GameFragment` получает ссылку на свой объект `GameViewModel`.
2. `GameFragment` присваивает объект `GameViewModel` переменной связывания данных `gameViewModel` макета. Представления макета теперь могут использовать эту переменную для обращения к свойствам и методам модели представления.
3. `GameFragment` назначает владельца жизненного цикла макета. Макет теперь может наблюдать за свойствами `Live Data` модели представления и реагировать на любые изменения.
4. Когда пользователь вводит правильное предположение, обновляется свойство `secretWordDisplay` объекта `GameViewModel`. Макет наблюдает за этим изменением, а в его представлении `word` отображается новый текст.
5. Если пользователь вводит ошибочное предположение, обновляются свойства `livesLeft` и `incorrectGuesses` объекта `GameViewModel`. Макет наблюдает за изменениями, а в соответствующих представлениях выводится новый текст.

6. Когда игра завершится, `GameFragment` переходит к `ResultFragment`.

Приложение работает так же, как прежде. Однако в этой версии `GameFragment` использует связывание данных для получения значений свойств напрямую из `GameViewModel`.

Свызывание данных может использоваться для вызова методов

К настоящему моменту мы реализовали связывание данных, чтобы представления получали свои значения прямо из модели представления, вместо того чтобы зависеть от кода фрагмента. Связывание данных можно использовать и другим способом: кнопка может вызвать метод из модели представления без написания дополнительного кода фрагмента. Чтобы показать, как это делается, мы добавим в макет `GameFragment` новую кнопку `Finish Game`; щелчок на этой кнопке будет завершать игру. Новая версия экрана будет выглядеть так:

Чтобы новая кнопка завершала игру, необходимо сделать следующее:

1. Добавление нового метода `finishGame()` в `GameViewModel`. Этот метод присваивает свойству `_gameOver` модели представления значение `true`. Когда это происходит, существующий код `GameFragment` реагирует на изменение переходом к `ResultFragment`.
2. Добавление новой кнопки в макет, по щелчку на которой будет вызываться метод `finishGame()`. Кнопка будет вызывать метод с помощью связывания данных.

Начнем с обновления кода `GameViewModel`.

Добавление метода finishGame() в GameViewModel.kt

В код `GameViewModel` необходимо добавить новый метод `finishGame()`, который присваивает `_gameOver` значение `true`. Вы уже знаете, как это делается; ниже приведен новый код `GameViewModel.kt`. Обновите код и включите новый метод:

```
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
    }

    fun finishGame() {
    _gameOver.value = true
    }
  }
```

И это весь код, который необходимо добавить. На следующем шаге в макет `GameFragment` будет добавлена новая кнопка `Finish Game`, по щелчку на которой будет вызываться метод `finishGame()`.

Использование связывания данных для вызова метода по щелчку на кнопке

Как было сказано выше, вы можете воспользоваться связыванием данных для того, чтобы по щелчку на кнопке вызывался метод его модели представления без написания дополнительного кода фрагмента. Для этого следует добавить к кнопке атрибут `android:onClick` и присвоить ему выражение для вызова метода.

В приложении `Guessing Game` в макет `GameFragment` нужно добавить новую кнопку, по щелчку на которой будет вызываться метод `finishGame()` объекта `GameViewModel`. Код кнопки выглядит так:

```
<Button
 android:id="@+id/finish_game_button"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="center"
 android:text="Finish Game"
 android:onClick="@{() -> gameViewModel.finishGame()}" />
```

Строка:

```
android:onClick="@{() -> gameViewModel.finishGame()}"
```

эквивалентна включению следущего кода во фрагмент:

```
binding.finishGameButton.setOnClickListener() {
 viewModel.finishGame()
}
```

кроме того, что на этот раз код фрагмента не потребуется. Во внутренней реализации связывание данных использует следующую строку:

```
android:onClick="@{() -> gameViewModel.finishGame()}"
```

для создания слушателя `onClickListener` для кнопки. Когда пользователь щелкает на кнопке, выполняется код, который вызывает метод `finishGame()` модели представления.

Когда вы используете выражение связывания для вызова метода:

```
"@{() -> gameViewModel.finishGame()}"
```

выражение связывания иногда называют связыванием слушателя.

И это все, что необходимо знать для включения кнопки в макет `GameFragment` и ее реагирования на щелчки. Посмотрим, как выглядит полный код.

Полный код fragment_game.xml

Ниже приведен полный код макета `GameFragment`; обновите код `fragment_game.xml` и включите в него новую кнопку:

```
<?xml version="1.0" encoding="utf-8"?>
<layout
 xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
 tools:context=".GameFragment">
 <data>
 <variable
 name="gameViewModel"
 type="com.hfad.guessinggame.GameViewModel" />
 </data>
 <LinearLayout
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical"
 android:padding="16dp">
 <TextView
 android:id="@+id/word"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="center"
 android:textSize="36sp"
 android:letterSpacing="0.1"
 android:text="@{gameViewModel.secretWordDisplay}" />
 <TextView
 android:id="@+id/lives"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:text="@{@string/lives_left(gameViewModel.livesLeft)}" />
<TextView
 android:id="@+id/incorrect_guesses"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:text="@{@string/incorrect_guesses(gameViewModel.incorrectGuesses)}" />
 <EditText
 android:id="@+id/guess"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:textSize="16sp"
 android:hint="Guess a letter"
 android:inputType="text"
 android:maxLength="1" />
 <Button
 android:id="@+id/guess_button"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:layout_gravity="center"
 android:text="Guess!"/>
 <Button
 android:id="@+id/finish_game_button"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
```

```
    android:layout_gravity="center"
    android:text="Finish Game"
    android:onClick="@{() -> gameViewModel.finishGame()}" />
    </LinearLayout>
  </layout>
```

Посмотрим, что происходит во время выполнения кода.

Что происходит при выполнении приложения

При выполнении приложения происходит следующее:

1. `GameFragment` присваивает объект `GameViewModel` переменной связывания данных `gameViewModel` макета. Представления макета теперь могут использовать переменную для обращения к свойствам и методам модели представления.

2. Пользователь щелкает на кнопке `Finish Game`. Кнопка вызывает метод `finishGame()` объекта `GameViewModel`, который присваивает свойству `_gameOver` значение `true`.

3. `GameFragment` обнаруживает, что значение `_gameOver` изменилось. Фрагмент реагирует переходом к `ResultFragment`.

При запуске приложения в макете `GameFragment` появляется новая кнопка `Finish Game`. По щелчку на кнопке приложение переходит к фрагменту `ResultFragment`, в котором выводится загаданное слово.

Приложение `Guessing Game` почти завершено, осталось внести одно последнее изменение.

Связывание представлений можно отключить

Обновление приложения `Guessing Game` почти завершено, но осталось внести еще одно изменение, чтобы в нем генерировались только те классы связывания, которые используются для связывания данных.

Как вы знаете, связывание данных и связывание представлений генерируют одни и те же классы связывания, но в разных обстоятельствах: связывание представлений генерирует класс связывания для каждого макета, тогда как связывание данных генерирует класс для каждого макета с корневым элементом `<layout>`.

В приложении `Guessing Game` классы связывания используются только для файлов `fragment_game.xml` и `fragment_result.xml`. Так как оба файла включают элемент `<layout>`, связывание представлений можно спокойно отключить, а его классы связывания попрежнему будут генерироваться механизмом связывания данных:

Отключение связывания представлений в файле build.gradle приложения

Чтобы отключить связывание представлений, откройте файл `GuessingGame/app/build.gradle` и удалите соответствующую строку из раздела `buildFeatures`:

```
android {
 ...
 buildFeatures {
 dataBinding true
 }
}
```

Затем щелкните на ссылке `Sync Now`, чтобы синхронизировать изменения с остальными частями проекта. Давайте проведем быстрый тест-драйв приложения и убедимся в том, что оно продолжает успешно работать.

Игра работает точно так же, как и прежде. Отключение связывания представлений ни на что не повлияло, потому что связывание данных сгенерировало все классы связывания, необходимые для работы приложения.

Поздравляем! Вы освоили связывание данных и научились использовать его в сочетании с моделями представлений и `Live Data`. Этот подход способствует упрощению кода фрагментов и помогает строить динамичные приложения, быстро реагирующие на действия пользователя.

Резюме

- Связывание данных предоставляет прямой доступ к свойствам и методам.
- Связывание данных включается в файле `build.gradle` приложения.
- Чтобы использовать связывание данных, включите корневой элемент `<layout>` в каждый макет, который должен использовать связывание данных.
- Связывание данных генерирует класс связывания для каждого макета с корневым элементом `<layout>`. Этот класс связывания не отличается от того, который генерируется для связывания представлений.
- Используйте элементы `<data>` и `<variable>` для определения переменных.
- Строковые ресурсы могут получать аргументы.
- Связывание данных позволяет реагировать на обновление значений свойств `Live Data`.