Практическая работа 11.2 Сохранение состояния с помощью модели представления

При повороте экрана состояние теряется

Однако в игре есть проблема. Если повернуть экран во время игры, приложение теряет свое состояние и игра начинается заново.

Игра теряет состояние, потому что поворот экрана изменяет конфигурацию приложения; Android уничтожает активность (и отображаемый ею фрагмент) и немедленно создает ее заново. При этом представления и свойства игры возвращаются к исходным значениям. Проблему можно решить сохранением состояния всех свойств в методе `onSaveInstanceState` фрагмента. Однако на этот раз мы воспользуемся другим решением и реализуем **модель представления**.

Как было сказано выше, модель представления — это отдельный класс, который существует параллельно с кодом активности или фрагмента. Он отвечает за данные, которые должны отображаться на экране, а также может включать любую бизнес-логику. Так, в приложении `Guessing Game` это означает, что модель представления должна хранить свойства игры (например, слово, которое пользователь пытается отгадать, и количество оставшихся жизней) и любые методы, управляющие ходом игры.

Когда вы реализуете модель представления, весь код, относящийся к данным приложения или бизнес-логике, перемещается из активности или фрагмента в модель представления. Весь код, управляющий пользовательским интерфейсом (например, вывод текста или получение данных от пользователя), остается в коде активности или фрагмента. Этот способ формирования структуры приложений Android следует принципу проектирования, известному как принцип разделения обязанностей. Приложение разбивается на классы, при этом каждый класс предназначен для выполнения некоторой отдельной обязанности. Контроллер пользовательского интерфейса — код активности или фрагмента — обеспечивает работу пользовательского интерфейса, а модель представления отвечает за бизнес-логику и данные:

Модель представления используется для упрощения кода активностей и фрагментов, а также для сохранения состояния свойств, чтобы они нормально пережили изменения конфигурации.

Такая архитектура приложения имеет два ключевых преимущества.

- Она упрощает код активностей и фрагментов. Выделение данных и бизнес-логики приложения в модель представления означает уменьшение объема кода активностей и фрагментов, который вам придется сопровождать.
- Ваше приложение справляется с изменениями конфигурации. Модель представления — это отдельный класс, который существует наряду с кодом активностей или фрагментов. Она не уничтожается при повороте экрана устройства, так что состояние свойств, хранящихся в модели представления, переживет изменения конфигурации, и вам не придется добавлять их значения в `Bundle`.

Теперь вы знаете, что вам даст использование модели представления. Посмотрим, как добавить модель представления в приложение `Guessing Game`.

Включение зависимости для модели представления в файл build.gradle приложения

Библиотека моделей представлений является частью Android Jetpack, поэтому вам придется обновить файл `build.gradle` приложения и включить ее как зависимость. Откройте файл *GuessingGame/app/build.gradle* и добавьте следующую строку (выделенную жирным шрифтом) в раздел `dependencies`:

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

Синхронизируйте изменения, когда вам будет предложено это сделать. Теперь все готово к созданию модели представления.

Мы создадим модель представления с именем `GameViewModel`, которая будет использоваться фрагментом `GameFragment` для хранения его логики и данных. Выделите пакет `com.hfad.guessinggame` в папке *app/src/main/java folder* и выберите команду `File→New→Kotlin Class/File`. Введите имя файла «GameViewModel» и выберите вариант создания класса. Когда файл `GameViewModel.kt` будет создан, обновите его код, чтобы он соответствовал приведенному ниже (изменения выделены жирным шрифтом):

```
package com.hfad.guessinggame
import androidx.lifecycle.ViewModel
class GameViewModel : ViewModel() {

}
```

Как видите, класс `GameViewModel` расширяет `androidx.lifecycle.ViewModel.ViewModel` — абстрактный класс, который используется для преобразования обычного традиционного класса в полноценную модель представления. После того как мы добавили класс `GameViewModel` в проект `Guessing` и преобразовали его в модель представления, можно переходить к написанию оставшегося кода.

Полный код GameViewModel.kt

Как вы уже узнали, модель представления отвечает за бизнес-логику и данные активности или фрагмента. Для игры `Guessing Game` это означает, что мы должны определить все свойства и методы

`GameFragment`, относящиеся к игровому процессу, и переместить их в `GameViewModel`. Весь код, относящийся к навигации или пользовательскому интерфейсу, останется в `GameFragment`. Ниже приведен полный код `GameViewModel.kt`; обновите код и включите в него изменения (выделенные жирным шрифтом):

```
package com.hfad.guessinggame
import androidx.lifecycle.ViewModel
class GameViewModel : ViewModel() {
 val words = listOf("Android", "Activity", "Fragment")
 val secretWord = words.random().uppercase()
 var secretWordDisplay = ""
 var correctGuesses = ""
 var incorrectGuesses = ""
 var livesLeft = 8
 init {
 secretWordDisplay = 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 = deriveSecretWordDisplay()
 } else {
 incorrectGuesses += "$guess "
 livesLeft--
 }
 }
 }
 fun isWon() = secretWord.equals(secretWordDisplay, true)
 fun isLost() = livesLeft <= 0
 fun wonLostMessage() : String {
 var message = ""
 if (isWon()) message = "You won!"
 else if (isLost()) message = "You lost!"
 message += " The word was $secretWord."
 return message
 }
}
```

Вот и все, что необходимо сделать для `GameViewModel`. На следующем шаге мы свяжем класс с `GameFragment` и обновим код этого фрагмента.

Создание объекта GameViewModel

Чтобы связать модель представления с активностью или фрагментом, следует добавить в код свойство `ViewModel` и инициализировать его объектом `ViewModel`, который необходимо создать. Код выглядит так:

```
class GameFragment : Fragment() {
 ...
 lateinit var viewModel: GameViewModel
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 ...
 viewModel = ViewModelProvider(this).get(GameViewModel::class.java)
 ...
 }
}
```

В этом коде для создания объекта `ViewModel` используется провайдер модели представления. Что же происходит при использовании этого класса?

Использование класса ViewModelProvider для создания моделей представлений

Как подсказывает имя, `ViewModelProvider` — специальный класс, который должен предоставлять модели представлений активностям и фрагментам. Он гарантирует, что новый объект модели представления создается только в том случае, если он не существует. Как вы уже знаете, при повороте экрана любые фрагменты, отображаемые на экране, уничтожаются и создаются заново. Когда это происходит, провайдер модели представления следит за тем, чтобы использовался тот же объект модели представления. Модель представления сохраняет свое состояние, так что любые свойства, используемые фрагментом, не сбрасываются. Провайдер модели представления хранит модель представления, пока активность или фрагмент продолжают существовать. Когда фрагмент отсоединяется или удаляется из своей активности, провайдер разрывает связь с моделью представления фрагмента. Когда провайдер модели представления получает следующий запрос на объект модели представления, он создает новый объект. Итак, теперь вы знаете, как связать модель представления с фрагментом, и мы можем перейти к обновлению кода `GameFragment`.

Обновленный код GameFragment.kt

Из `GameFragment` следует удалить свойства и методы, перемещенные в `GameViewModel`, и заставить фрагмент использовать модель представления. Ниже приведен обновленный код `GameFragment`; убедитесь в том, что файл `GameFragment.kt` включает эти изменения:

```
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
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)

updateScreen()
 binding.guessButton.setOnClickListener() {
 viewModel.makeGuess(binding.guess.text.toString().uppercase())
 binding.guess.text = null
 updateScreen()
 if (viewModel.isWon() || viewModel.isLost()) {
 val action = GameFragmentDirections
 .actionGameFragmentToResultFragment(viewModel.wonLostMessage())
 view.findNavController().navigate(action)
 }
 }
 return view
 }
 override fun onDestroyView() {
 super.onDestroyView()
 _binding = null
 }
 fun updateScreen() {
 binding.word.text = viewModel.secretWordDisplay
 binding.lives.text = "You have ${viewModel.livesLeft} lives left."
 binding.incorrectGuesses.text = "Incorrect guesses:
${viewModel.incorrectGuesses}"
 }
 }
```

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

Что происходит при выполнении приложения

Во время выполнения приложения происходят следующие события:

1. `GameFragment` запрашивает у класса `ViewModelProvider` экземпляр `GameViewModel`. Провайдер модели представления видит, что в настоящий момент объект `GameViewModel`, связанный с фрагментом, не существует, поэтому он создает новый экземпляр.

2. Объект `GameViewModel` инициализируется. Свойству `livesLeft` присваивается 8, `correctGuesses` и `incorrectGuesses` присваивается "", `secretWord` — случайно выбранное слово, а `secretWordDisplay` — результат `deriveSecretWordDisplay()`.

3. `GameFragment` вызывает метод `updateScreen()`. Метод обращается к свойствам `secretWordDisplay`, `livesLeft` и `incorrectGueses` объекта `GameViewModel` и выводит их на экран.

4. Когда пользователь вводит предположение, `GameFragment` вызывает метод `makeGuess()` объекта `GameViewModel`. Метод проверяет, содержит ли `secretWord` введенную букву. Если буква встречается в слове, то она добавляется в `correctGuesses`, а значение `secretWordDisplay` обновляется. Если буква отсутствует в слове, она добавляется в `incorrectGuesses`, а значение `livesLeft` уменьшается на 1.

5. `GameFragment` снова вызывает свой метод `updateScreen()`. Метод получает обновленные значения свойств из объекта `GameViewModel` и обновляет экран.

6. После каждого предположения `GameFragment` проверяет, возвращает ли значение true один из методов `isWon()` или `isLost()` модели представления. Если хотя бы один из методов возвращает true, `GameFragment` передает результат фрагменту `ResultFragment`, который выводит результат.

При запуске приложения отображается фрагмент `GameFragment`, как и прежде. Если начать игру и повернуть экран, игра сохраняет свое состояние.

Вы узнали, как добавить модель представления в ваши приложения и как использовать ее для предотвращения проблем, которые могут возникнуть при повороте экрана устройства. Но прежде чем двигаться дальше, нужно поближе познакомиться с моделями представлений.

К настоящему моменту мы обновили приложение `Guessing Game`, чтобы фрагмент `GameFragment` использовал модель представления с именем `GameViewModel`, которая отвечает за всю бизнес-логику и данные фрагмента. Такое использование модели представления упрощает код `GameFragment.kt` и означает, что приложение не будет терять состояние при повороте экрана.

Каждая модель представления, которую вы создаете, связывается с одним UI-контроллером — активностью или фрагментом. Таким образом, если вы хотите, чтобы фрагмент `ResultFragment` использовал модель представления, необходимо создать новую модель для этого фрагмента. Давайте сделаем это. Выделите пакет `com.hfad.guessinggame` в папке *app/src/main/java* и выберите команду `File→New→Kotlin Class/File`. Введите имя файла «ResultViewModel» и выберите вариант создания класса. Когда файл `ResultViewModel.kt` будет создан, обновите код и приведите его к следующему виду:

```
package com.hfad.guessinggame
import androidx.lifecycle.ViewModel
```

```
class ResultViewModel : ViewModel(){
}
```

Это базовый код, необходимый для определения модели представления. Какой еще код следует добавить?

Как говорилось ранее, `ResultFragment` выводит в своем макете сообщение о том, выиграл он или проиграл текущую игру. Это сообщение передается `ResultFragment` фрагментом `GameFragment` при завершении игры. В новой версии приложения `ResultViewModel` отвечает за игровую логику и данные `ResultFragment`, поэтому в `ResultViewModel` необходимо включить свойство для хранения результата. Мы также используем конструктор с параметром `String`, чтобы значение свойства задавалось сразу же после создания `ResultViewModel`. Ниже приведен полный код `ResultViewModel.kt`; обновите свою версию файла (изменения выделены жирным шрифтом):

```
package com.hfad.guessinggame
import androidx.lifecycle.ViewModel
class ResultViewModel(finalResult: String) : ViewModel(){
 val result = finalResult
}
```

Затем обновим фрагмент `ResultFragment`, чтобы в нем использовалась новая модель представления.

Связываем ResultViewModel с ResultFragment

Ранее для добавления ссылки `GameViewModel` в `GameFragment` использовался следующий код:

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

Он отдает команду провайдеру модели представления получить объект `GameViewModel`, связанный с фрагментом, или создать новый объект, если он еще не существует. Однако этот способ не может использоваться для включения ссылки на `ResultViewModel` в `ResultFragment`. Дело в том, что он подходит только для моделей представлений с конструктором без аргументов. Такое решение работало для `GameViewModel`, потому что объект можно было сконструировать для передачи аргументов. Но конструктор класса `ResultViewModel` должен получать строку, поэтому приведенный выше код работать не будет.

Фабрика модели представления создает модели представлений

Альтернативный способ создания модели представления основан на передаче провайдеру модели представления фабрики модели представлений: отдельного класса, единственным назначением которого является создание и инициализация моделей представлений. Такой подход означает, что провайдеру модели представления не придется беспокоиться о создании модели представления своими силами. Вместо этого он использует фабрику модели представления. Хотя фабрики моделей представлений могут использоваться для любых разновидностей моделей представлений, чаще всего они используются для моделей, конструкторы которых должны получать аргументы. Дело в том, что

провайдер модели представления не может передавать аргументы конструктору сам по себе: для этого он должен использовать фабрику модели представления. Код использования фабрики модели представления в приложении `Guessing Game` выглядит так:

1. Мы определяем класс `ResultViewModelFactory`, который будет использоваться `ResultFragment` для создания объекта фабрики.
2. `ResultFragment` отдает команду провайдеру модели представления использовать объект фабрики.
3. Когда провайдеру модели представления потребуется новый объект `ResultViewModel`, он использует `ResultViewModelFactory`.

Сделаем следующий шаг и добавим класс фабрики в приложение `Guessing Game`.

Создание класса ResultViewModelFactory

Мы добавим в приложение `Guessing Game` класс фабрики модели представления с именем `ResultViewModelFactory`. Этот класс будет использоваться провайдером модели представления для создания объекта `ResultViewModel`. Чтобы создать класс, выделите пакет `com.hfad.guessinggame` в папке */src/main/java* и выберите команду `File→New→Kotlin Class/File`. Введите имя файла «ResultViewModelFactory» и выберите вариант создания класса. После того как файл `ResultViewModelFactory.kt` будет создан, приведите его к следующему виду:

```
package com.hfad.guessinggame
import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import java.lang.IllegalArgumentException
class ResultViewModelFactory(private val finalResult: String)
 : ViewModelProvider.Factory {
 override fun <T : ViewModel?> create(modelClass: Class<T>): T {
 if (modelClass.isAssignableFrom(ResultViewModel::class.java))
 return ResultViewModel(finalResult) as T
 throw IllegalArgumentException("Unknown ViewModel")
 }
}
```

Как видно из кода, класс `ResultViewModelFactory` реализует интерфейс с именем `ViewModelProvider.Factory` и переопределяет его метод `create()`. Тем самым класс преобразуется в фабрику модели представления, которая может использоваться провайдером модели представления для создания объекта `ResultViewModel`. Приведенный выше код — все, что необходимо для работы `ResultViewModelFactory`. Посмотрим, как использовать его в коде `ResultFragment`.

Использование фабрики для создания модели представления

Как говорилось ранее, фабрика модели представления используется для создания модели представления, для чего провайдеру модели представления передается фабрика. Провайдер модели представления решает, нужно ли создать новый объект модели представления, и при необходимости использует фабрику для его создания. Код, обеспечивающий использование фабрики провайдером

модели представления, будет практически идентичным для всех создаваемых вами моделей представлений. Он выглядит примерно так:

```
class ResultFragment : Fragment() {
 ...
 lateinit var viewModel: ResultViewModel
 lateinit var viewModelFactory: ResultViewModelFactory
 override fun onCreateView(
 inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
 ): View? {
 ...
 val result = ResultFragmentArgs.fromBundle(requireArguments()).result
 viewModelFactory = ResultViewModelFactory(result)
 viewModel = ViewModelProvider(this, viewModelFactory)
 .get(ResultViewModel::class.java)
 ...
 }
}
```

Как видите, в этом коде определяются два свойства: `viewModel` и `viewModelFactory`. Они задаются в методе `onCreateView()` фрагмента. В `onCreateView()` код использует строку `result`, переданную ему из `GameFragment`, для создания нового объекта `ResultViewModelFactory`. Он передает фабрику провайдеру модели представления, который использует ее для получения объекта `ResultViewModel`. Итак, теперь вы знаете, как использовать фабрику для связывания модели представления с фрагментом, и мы можем перейти к обновлению кода `ResultFragment`.

Обновленный код ResultFragment.kt

Ниже приведен полный код `ResultFragment`. Обновите код в файле `ResultFragment.kt`, чтобы он включал изменения, приведенные ниже:

```
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.FragmentResultBinding
import androidx.navigation.findNavController
import androidx.lifecycle.ViewModelProvider
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.wonLost.text = viewModel.result
    binding.newGameButton.setOnClickListener {
    view.findNavController()
    .navigate(R.id.action_resultFragment_to_gameFragment)
    }
    return view
    }
    override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
    }
   }
```

И это все изменения, которые необходимо внести в `ResultFragment`. После ответов на некоторые вопросы мы разберемся в том, что же происходит во время выполнения приложения.

Что происходит при выполнении приложения

При выполнении приложения происходят следующие события:

1. `GameFragment` запрашивает у класса `ViewModelProvider` экземпляр `GameViewModel`. Объект `GameViewModel` инициализируется и выбирает случайное слово.
2. `GameFragment` взаимодействует с объектом `GameViewModel`. Объект `GameViewModel` сохраняет все предположения, сделанные пользователем, и отслеживает количество оставшихся жизней.
3. После каждого предположения `GameFragment` проверяет, возвращает ли true один из методов `isWon()` или `isLost()` модели представления. Если один из методов возвращает true, `GameFragment` переходит к `ResultFragment` с передачей `result`.
4. `ResultFragment` создает объект `ResultViewModelFactory` и передает ему строку `result`.
5. `ResultFragment` запрашивает у класса `ViewModelProvider` экземпляр `ResultViewModel`. Класс `ViewModelProvider` видит, что существующего объекта `ResultViewModel` нет, поэтому он создает его при помощи `ResultViewModelFactory`. Свойство result объекта `ResultViewModel` инициализируется строкой `result`.
6. `ResultFragment` получает строку `result` из объекта `ResultViewModel`, и выводит ее на экран.

При запуске приложения, как и прежде, отображается `GameFragment`. Если пользователь угадал все буквы или потратил все жизни, приложение переходит к `ResultFragment`. Выводится сообщение с информацией о том, выиграл или проиграл пользователь и какое слово было загадано.

На первый взгляд игра работает точно так же, как и прежде, но теперь для игровой логики и данных в ее новой версии используются модели представлений.

Резюме

- Модель представления упрощает код активности или фрагмента за счет отделения кода, относящегося к бизнес-логике и данным.
- Модели представлений могут нормально переносить изменения конфигурации.
- Модель представления обычно связывается с отдельным UI-контроллером.
- Модель представления расширяет класс `androidx.lifecycle.ViewModel`.
- Модель представления создается вызовом метода `get()` класса `ViewModelProvider`. Это гарантирует, что модель представления будет создаваться только в том случае, если она еще не существует.
- Модель представления продолжает существовать до тех пор, пока UI-контроллер не будет уничтожен навсегда. Это происходит при завершении активности или удаления фрагмента из его активности.
- Если конструктору модели представления необходимы аргументы, вам придется написать для него дополнительный класс — фабрику. Класс `ViewModelProvider` использует фабрику каждый раз, когда ему потребуется создать экземпляр модели представления.
- Класс фабрики модели представления должен реализовать интерфейс с именем `ViewModelProvider.Factory`.