1. Wprowadzenie do ViewModel i ViewModelFactory

W tym kodzie źródłowym poznasz jeden z komponentów architektury systemu Android, ViewModel: Android Architecture Components, `ViewModel`:

- Używasz klasy ViewModel do przechowywania danych związanych z interfejsem użytkownika i zarządzania nimi w sposób uwzględniający cykl życia. Klasa ViewModel pozwala przetrwać zmianom w konfiguracji urządzenia, takim jak obracanie ekranu i zmiany dostępności klawiatury.
- a pomocą klasy ViewModelFactory tworzy się instancję i zwraca obiekt ViewModel, który przetrwa zmiany konfiguracji.

Czego się nauczysz

- Jak korzystać z zalecanej architektury aplikacji na Androida. app architecture.
- Jak korzystać z klas Lifecycle, ViewModel i ViewModelFactory w aplikacji.
- Jak zachować dane interfejsu użytkownika poprzez zmiany konfiguracji urządzenia.
- Co to jest wzorzec projektowania metody fabrycznej i jak go używać. factory method.
- Jak utworzyć obiekt ViewModel za pomocą interfejsu. `ViewModelProvider.Factory`.

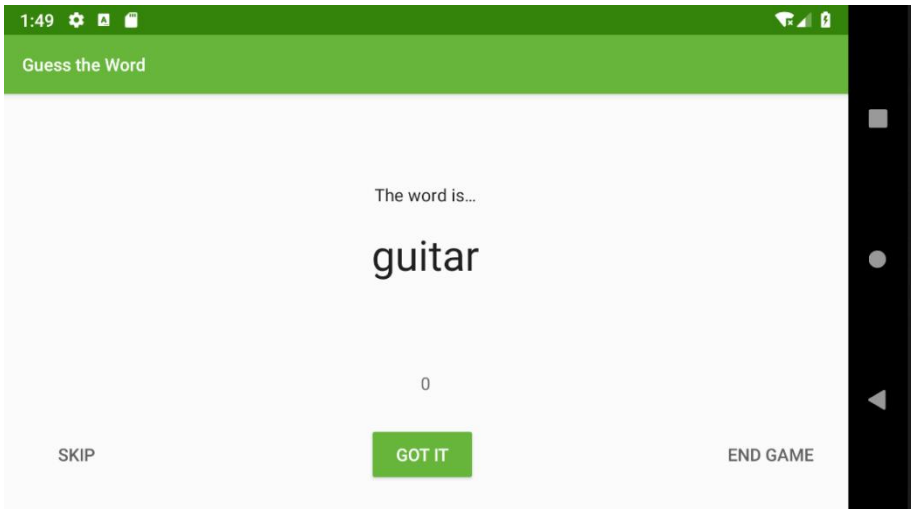
Co będziesz robić

- Dodaj ViewModel do aplikacji, aby zapisać dane aplikacji, aby dane przetrwały zmiany konfiguracji.
- Użyj `ViewModelFactory` i fabrycznego wzorca projektowego, aby utworzyć instancję obiektu ViewModel z parametrami konstruktora.

2. Przegląd aplikacji

GuessTheWord to gra dla dwóch graczy, w której gracze współpracują ze sobą, aby osiągnąć jak najwyższy wynik.

Aby zagrać w tę grę, pierwszy gracz otwiera aplikację na urządzeniu i widzi słowo, na przykład „gitara”, jak pokazano na zrzucie ekranu poniżej.



Pierwszy gracz opisuje słowo, uważając, aby nie wypowiedzieć samego słowa.

- Gdy drugi gracz poprawnie odgadnie słowo, pierwszy gracz naciska przycisk Got It, który zwiększa liczbę o jeden i pokazuje następne słowo.
- Jeśli drugi gracz nie może odgadnąć słowa, pierwszy gracz naciska przycisk Pomiń, co zmniejsza liczbę o jeden i przeskakuje do następnego słowa.
- Aby zakończyć grę, naciśnij przycisk End Game. (Ta funkcja nie znajduje się w kodzie startowym).

3. Zadanie: poznaj kod startowy

W tym zadaniu pobierasz i uruchamiasz aplikację startową oraz sprawdzasz kod.

Step 1: Rozpocznij

1. Pobierz kod startowy GuessTheWord i otwórz projekt w Android Studio.
2. Uruchom aplikację na urządzeniu z Androidem lub emulatorze.
3. Stuknij przyciski. Zauważ, że przycisk Pomiń wyświetla następne słowo i zmniejsza wynik o jeden, a przycisk Got pokazuje kolejne słowo i zwiększa wynik o jeden. Przycisk Zakończ grę nie jest zaimplementowany, więc nic się nie dzieje po jego naciśnięciu..

Step 2: Wykonaj instrukcje dotyczące kodu

1. W Android Studio zapoznaj się z kodem, aby poznać działanie aplikacji.
2. Sprawdź pliki opisane poniżej, które są szczególnie ważne.

MainActivity.kt

Ten plik zawiera tylko domyślny kod wygenerowany na podstawie szablonu.

res/layout/main_activity.xml

Ten plik zawiera główny układ aplikacji. `NavHostFragment` obsługuje inne fragmenty, gdy użytkownik porusza się po aplikacji.

UI fragments

Kod startowy zawiera trzy fragmenty w trzech różnych pakietach w pakiecie `com.example.android.guesstheword.screens`:

- `title/TitleFragment` dla ekranu tytułowego
- `game/GameFragment` na ekranie gry
- `score/ScoreFragment` dla ekranu wyników

screens/title/TitleFragment.kt

Fragment tytułu to pierwszy ekran wyświetlany po uruchomieniu aplikacji. Moduł obsługi kliknięć jest ustawiony na przycisk Odtwórz, aby przejść do ekranu gry.

screens/game/GameFragment.kt

To jest główny fragment, w którym odbywa się większość akcji gry:

- Zmienne są zdefiniowane dla bieżącego słowa i bieżącego wyniku.
- `wordList` zdefiniowany w metodzie`resetList()` jest przykładową listą słów do użycia w grze.
- Metoda `onSkip()` to moduł obsługi kliknięć przycisku Pomiń. Zmniejsza wynik o 1, a następnie wyświetla następne słowo za pomocą metody `nextWord()`.
- Metoda `onCorrect()` to moduł obsługi kliknięć przycisku Got It. Ta metoda jest implementowana podobnie do metody `onSkip()` Jedyną różnicą jest to, że ta metoda dodaje 1 do wyniku zamiast odejmować.

screens/score/ScoreFragment.kt

`ScoreFragment` to ostatni ekran w grze, który wyświetla ostateczny wynik gracza. W tym ćwiczeniu dodajesz implementację, aby wyświetlić ten ekran i pokazać końcowy wynik..

res/navigation/main_navigation.xml

Wykres nawigacyjny pokazuje sposób łączenia fragmentów za pomocą nawigacji:

- Z fragmentu tytułowego użytkownik może przejść do fragmentu gry.
- Z fragmentu gry użytkownik może przejść do score fragment.
- Ze score fragment, użytkownik może wrócić do fragmentu gry.

4. Zadanie: znajdowanie problemów w aplikacji startowej

W tym zadaniu wynajdujemy problemy z aplikacją startową GuessTheWord.

1. Uruchom kod startowy i zagraj w grę kilkoma słowami, dotykając Pomiń lub Mam go po każdym słowie.

2. Ekran gry pokazuje teraz słowo i aktualny wynik. Zmień orientację ekranu, obracając urządzenie lub emulator. Zauważ, że aktualny wynik został utracony.
3. Uruchom grę przez kilka dodatkowych słów. Gdy wyświetlony zostanie ekran gry z pewnym wynikiem, zamknij i ponownie otwórz aplikację. Zauważ, że gra uruchamia się ponownie od początku, ponieważ stan aplikacji nie jest zapisywany.
4. Zagraj w grę za pomocą kilku słów, a następnie dotknij przycisku Zakończ grę. Zauważ, że nic się nie dzieje..

Problemy w aplikacji:

- Aplikacja startowa nie zapisuje i nie przywraca stanu aplikacji podczas zmian konfiguracji, takich jak zmiana orientacji urządzenia lub gdy aplikacja wyłącza się i uruchamia ponownie.
  Możesz rozwiązać ten problem za pomocą wywołania zwrotnego `onSaveInstanceState()` Jednak użycie metody `onSaveInstanceState()` wymaga napisania dodatkowego kodu w celu zapisania stanu w pakiecie oraz wdrożenia logiki w celu odzyskania tego stanu. Ponadto ilość danych, które można przechowywać, jest minimalna.
- Ekran gry nie przechodzi do ekranu wyników, gdy użytkownik stuknie przycisk Zakończ grę.

Możesz rozwiązać te problemy za pomocą komponentów architektury aplikacji, o których dowiesz się w tym ćwiczeniu.

Architektura aplikacji

Architektura aplikacji to sposób projektowania klas aplikacji, oraz relacji między nimi, taki, że kod jest zorganizowany, działa dobrze w określonych scenariuszach i jest łatwy w obsłudze. W tym zestawie czterech ćwiczeń ulepszenia wprowadzone w aplikacji GuessTheWord są zgodne z wytycznymi dotyczącymi architektury aplikacji na Androida oraz używania komponentów architektury Android. Architektura aplikacji na Androida jest podobna do wzorca projektowego MVVM (model-view-viewmodel).

Aplikacja GuessTheWord jest zgodna z zasadą rozdzielania zagadnień separation of concerns i jest podzielona na klasy, z których każda dotyczy osobnego zagadnienia (concern). W pierwszej części skupimy się na kontrolerze interfejsu użytkownika, (UI controller, `ViewModel`, `ViewModelFactory`).

Kontroler interfejsu użytkownika (UI controller)

Kontroler interfejsu użytkownika to klasa oparta na interfejsach użytkownika, takich jak `Activity` lub `Fragment`. Kontroler interfejsu użytkownika powinien zawierać tylko logikę, która obsługuje interakcje interfejsu użytkownika i systemu operacyjnego, takie jak wyświetlanie widoków i przechwytywanie danych wejściowych użytkownika. Nie umieszczaj logiki decyzyjnej, takiej jak logika, która określa tekst do wyświetlenia, w kontrolerze interfejsu użytkownika.

W kodzie startowym GuessTheWord kontrolery interfejsu użytkownika to trzy fragmenty: `GameFragment`, `ScoreFragment`, i `TitleFragment`. Zgodnie z zasadą projektowania

„rozdzielenia obaw” `GameFragment` jest odpowiedzialny tylko za rysowanie elementów gry na ekranie i przechwytywaniu, kiedy użytkownik naciska przyciski, i nic więcej. Gdy użytkownik stuknie przycisk, informacje te są przekazywane do GameViewModel. `GameViewModel`.

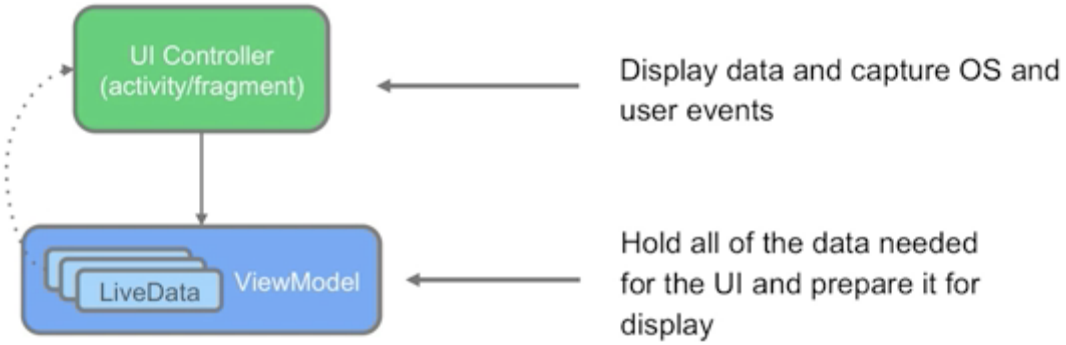
ViewModel

`ViewModel` przechowuje dane do wyświetlenia we fragmencie lub activity związanym z `ViewModel`. `ViewModel` może wykonywać proste obliczenia i transformacje danych, aby przygotować dane do wyświetlenia przez kontroler interfejsu użytkownika. W tej architekturze ViewModel podejmuje decyzje.

`GameViewModel` przechowuje dane, takie jak wartość punktacji, lista słów i bieżące słowo, ponieważ są to dane, które mają być wyświetlane na ekranie. GameViewModel zawiera również logikę biznesową do wykonywania prostych obliczeń w celu ustalenia aktualnego stanu danych.

ViewModelFactory

`ViewModelFactory` tworzy obiekty `ViewModel` z parametrami konstruktora lub bez nich.



W późniejszych ćwiczeniach dowiesz się o innych komponentach architektury Androida, które są powiązane z kontrolerami interfejsu użytkownika i `ViewModel`.

5. Zadanie: Utwórz GameViewModel

Klasa ViewModel została zaprojektowana do przechowywania danych związanych z interfejsem użytkownika i zarządzania nimi. W tej aplikacji każdy ViewModel jest powiązany z jednym fragmentem.

W tym zadaniu dodajesz swój pierwszy ViewModel do aplikacji, GameViewModel dla GameFragment. Dowiesz się również, co to znaczy, że ViewModel ma świadomość cyklu życia.

Step 1: Dodaj klasę GameViewModel class

1. Otwórz plik `build.gradle(module:app)` Wewnątrz bloku zależności dodaj zależność Gradle dla `ViewModel`.

   Jeśli używasz najnowszej wersji biblioteki, aplikacja powinna się skompilować zgodnie z oczekiwaniami. Jeśli nie, spróbuj rozwiązać problem lub przywróć wersję pokazaną poniżej.

```
//ViewModel
implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0'
```

2. W folderze `screens/game/` utwórz nową klasę Kotlin o nazwie `GameViewModel`.
3. Spraw, aby klasa `GameViewModel` rozszerzyła klasę abstrakcyjną (abstract class) `ViewModel`.
4. Aby lepiej zrozumieć, w jaki sposób ViewModel jest świadomy cyklu życia, dodaj blok init z instrukcją dziennika.

```
class GameViewModel : ViewModel() {
    init {
        Log.i("GameViewModel", "GameViewModel created!")
    }
}
```

Step 2: Zastąp onCleared () i dodaj rejestrowanie

`ViewModel` jest niszczony po odłączeniu powiązanego fragmentu lub po zakończeniu. Tuż przed zniszczeniem ViewModel wywoływane jest wywołanie zwrotne `onCleared()` w celu oczyszczenia zasobów.

1. W klasie `GameViewModel` zastąp metodę `onCleared()` method.
2. Dodaj instrukcję dziennika wewnątrz `onCleared()` aby śledzić cykl życia `GameViewModel`.

```
override fun onCleared() {
    super.onCleared()
    Log.i("GameViewModel", "GameViewModel destroyed!")
}
```

Step 3: Powiąż GameViewModel z fragmentem gry

ViewModel musi być powiązany z kontrolerem interfejsu użytkownika. Aby powiązać te dwa elementy, należy utworzyć odwołanie do ViewModel wewnątrz kontrolera interfejsu użytkownika.

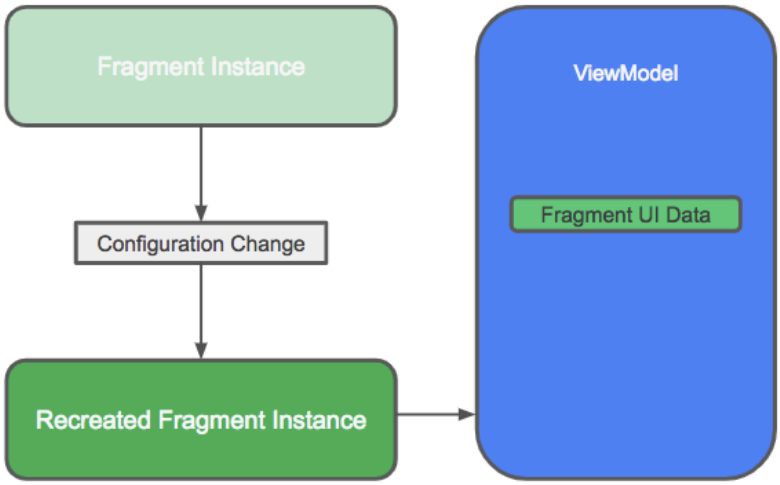
W tym kroku utworzysz odwołanie do `GameViewModel` wewnątrz odpowiedniego kontrolera interfejsu użytkownika, którym jest `GameFragment`.

1. W klasie `GameFragment` class, dodaj pole typu `GameViewModel` na najwyższym poziomie jako zmienną klasową.

```
private lateinit var viewModel: GameViewModel
```

Step 4: Zainicjuj ViewModel

Podczas zmian konfiguracji, takich jak obracanie ekranu, odtwarzane są kontrolery interfejsu użytkownika, takie jak fragmenty. Jednak instancje ViewModel przetrwają. Jeśli utworzysz instancję ViewModel za pomocą klasy ViewModel, nowy obiekt jest tworzony za każdym razem, gdy fragment jest odtwarzany ponownie. Zamiast tego utwórz instancję ViewModel za pomocą `ViewModelProvider`.



Ważne: Ważne: Zawsze używaj ViewModelProvider do tworzenia obiektów ViewModel, zamiast bezpośrednio tworzyć instancję `ViewModel`.

Jak działa `ViewModelProvider`:

- `ViewModelProvider` zwraca istniejący `ViewModel` eśli taki istnieje, lub tworzy nowy, jeśli jeszcze nie istnieje.
- `ViewModelProvider` tworzy instancję `ViewModel` w powiązaniu z danym zakresem (activity lub fragment).
- Utworzony `ViewModel` jest zachowany tak długo, jak zakres jest żywy. Na przykład, jeśli zakres jest fragmentem, ViewModel jest zachowywany do momentu odłączenia fragmentu.

Zainicjuj `ViewModel`, używając metody `ViewModelProviders.of()` aby utworzyć `ViewModelProvider`:

1. W klasie `GameFragment` zainicjuj zmienną `viewModel` Umieść ten kod wewnątrz `onCreateView()`,po definicji zmiennej wiążącej. Użyj metody `ViewModelProviders.of()` przekaż powiązany kontekst `GameFragment` oraz `GameViewModel`.
2. Nad inicjalizacją obiektu `ViewModel` dodaj instrukcję dziennika, aby zarejestrować wywołanie metody `ViewModelProviders.of()`.

```
Log.i("GameFragment", "Called ViewModelProviders.of")
viewModel = ViewModelProviders.of(this).get(GameViewModel::class.java)
```

3. Uruchom aplikację. W Android Studio otwórz okienko Logcat i filtruj w Game. Naciśnij przycisk Odtwórz na swoim urządzeniu lub emulatorze. Otwiera się ekran gry.

   Jak pokazano w Logcat, metoda `onCreateView()GameFragment` wywołuje metodę `ViewModelProviders.of()` w celu utworzenia `GameViewModel`.

4. Włącz automatyczne obracanie w urządzeniu lub emulatorze i kilkakrotnie zmień orientację ekranu. `GameFragment` jest niszczony i ponownie tworzony za każdym razem, więc `ViewModelProviders.of()` jest wywoływany za każdym razem. Ale `GameViewModel` est tworzony tylko raz i nie jest odtwarzany ani niszczony dla każdego połączenia.

```
I/GameFragment: Called ViewModelProviders.of
I/GameViewModel: GameViewModel created!
I/GameFragment: Called ViewModelProviders.of
I/GameFragment: Called ViewModelProviders.of
I/GameFragment: Called ViewModelProviders.of
```

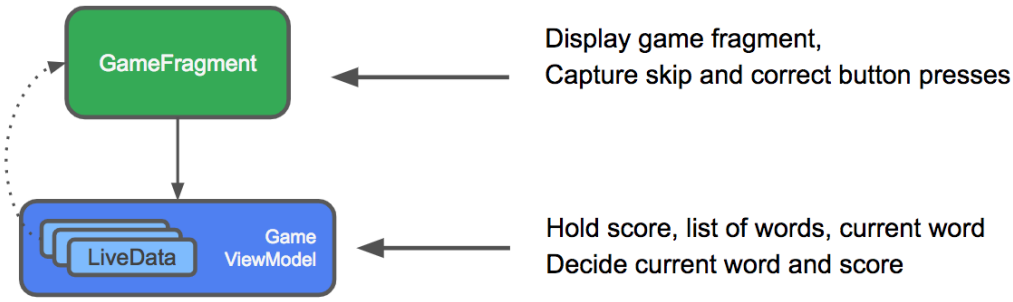
5. Wyjdź z gry lub przejdź do innego fragmentu gry. `GameFragment` jest zniszczony. Powiązany `GameViewModel` jest również niszczony i wywoływane jest wywołanie zwrotne `onCleared`.

```
I/GameFragment: Called ViewModelProviders.of
I/GameViewModel: GameViewModel created!
I/GameFragment: Called ViewModelProviders.of
I/GameFragment: Called ViewModelProviders.of
I/GameFragment: Called ViewModelProviders.of
I/GameViewModel: GameViewModel destroyed!
```

6. Zadanie: wypełnij GameViewModel

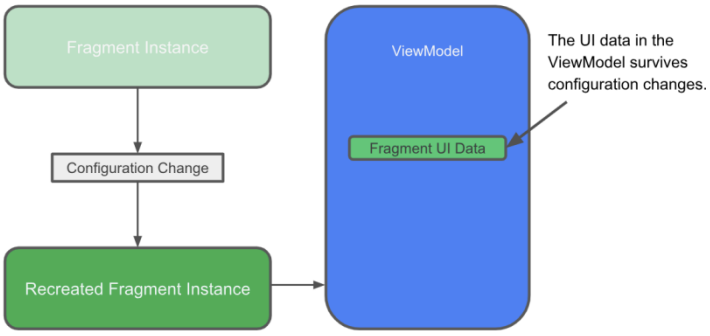
ViewModel przetrwa zmiany konfiguracji, więc jest to dobre miejsce dla danych, które muszą przetrwać zmiany konfiguracji:

- Umieść dane do wyświetlenia na ekranie i kod do ich przetworzenia w `ViewModel`.
- ViewModel nigdy nie powinien zawierać odniesień do fragmentów, działań lub widoków, ponieważ działania, fragmenty i widoki nie przetrwają zmian konfiguracji.



Dla porównania, oto sposób obsługi danych interfejsu `GameFragment` w aplikacji startowej przed dodaniem `ViewModel`, i po dodaniu `ViewModel`:

- Przed dodaniem `ViewModel`:
  Gdy aplikacja przechodzi zmianę konfiguracji, na przykład obracanie ekranu, fragment gry zostaje zniszczony i ponownie utworzony. Dane zostały utracone.
- Po dodaniu ViewModel i przeniesieniu danych interfejsu użytkownika fragmentu gry do ViewModel:
  Wszystkie dane, które fragment musi wyświetlić, to teraz ViewModel. Gdy aplikacja przechodzi zmianę konfiguracji, ViewModel przetrwa, a dane zostaną zachowane.



W tym zadaniu przenosisz dane interfejsu użytkownika aplikacji do klasy `GameViewModel` wraz z metodami przetwarzania danych. Robisz to, aby dane były zachowywane podczas zmian konfiguracji.

Step 1: Przenieś pola danych i przetwarzanie danych do ViewModel

Przenieś następujące pola danych i metody z `GameFragment` do `GameViewModel`:

1. Przenieś pola danych `word`, `score`, i `wordList` data fields. Upewnij się, że `word` i `score` nie są `private`.

   Nie należy przenosić zmiennej powiązania, `GameFragmentBinding`, ponieważ zawiera ona odwołania do widoków. Ta zmienna służy do nadmuchiwania układu, konfigurowania detektorów kliknięć i wyświetlania danych na ekranie.

2. Przenieś metody `resetList()` i `nextWord()` Te metody decydują o tym, jakie słowo ma być wyświetlane na ekranie.
3. Wewnątrz metody `onCreateView()` przenieś wywołania metod do `resetList()` i `nextWord()` do blok `init` `GameViewModel`.

   Te metody muszą znajdować się w bloku `init` ponieważ należy zresetować listę słów podczas tworzenia `ViewModel` , nie za każdym razem, gdy fragment jest tworzony. Możesz usunąć instrukcję dziennika w bloku `init` w `GameFragment`.

Programy obsługi kliknięć `onSkip()` i `onCorrect()` w `GameFragment` zawierają kod do przetwarzania danych i aktualizacji interfejsu użytkownika. Kod do aktualizacji interfejsu powinien pozostać w tym fragmencie, ale kod do przetwarzania danych musi zostać przeniesiony do `ViewModel`.

Na razie umieść identyczne metody w obu miejscach:

1. Skopiuj metody `onSkip()` i `onCorrect()` z `GameFragment` do `GameViewModel`.
2. W `GameViewModel`, upewnij się, że metody `onSkip()` i `onCorrect()` nie są `private`, , ponieważ będziesz odwoływał się do tych metod z fragmentu.

Oto kod klasy `GameViewModel` po refaktoryzacji:

```
class GameViewModel : ViewModel() {
   // The current word
   var word = ""
   // The current score
   var score = 0
   // The list of words - the front of the list is the next word to guess
   private lateinit var wordList: MutableList<String>

   /**
    * Resets the list of words and randomizes the order
    */
   private fun resetList() {
       wordList = mutableListOf(
               "queen",
               "hospital",
               "basketball",
               "cat",
               "change",
               "snail",
               "soup",
               "calendar",
               "sad",
               "desk",
               "guitar",
               "home",
               "railway",
               "zebra",
               "jelly",
               "car",
               "crow",
               "trade",
               "bag",
               "roll",
               "bubble"
```

```
        )
        wordList.shuffle()
    }

    init {
        resetList()
        nextWord()
        Log.i("GameViewModel", "GameViewModel created!")
    }
    /**
     * Moves to the next word in the list
     */
    private fun nextWord() {
        if (!wordList.isEmpty()) {
            //Select and remove a word from the list
            word = wordList.removeAt(0)
        }
        updateWordText()
        updateScoreText()
    }
 /** Methods for buttons presses **/
    fun onSkip() {
        if (!wordList.isEmpty()) {
            score--
        }
        nextWord()
    }

    fun onCorrect() {
        if (!wordList.isEmpty()) {
            score++
        }
        nextWord()
    }

    override fun onCleared() {
        super.onCleared()
        Log.i("GameViewModel", "GameViewModel destroyed!")
    }
}
```

Here is the code for the `GameFragment` class, after refactoring:

```
/**
* Fragment where the game is played
*/
class GameFragment : Fragment() {


    private lateinit var binding: GameFragmentBinding


    private lateinit var viewModel: GameViewModel


    override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?,
                              savedInstanceState: Bundle?): View? {

        // Inflate view and obtain an instance of the binding class
        binding = DataBindingUtil.inflate(
```

```
                inflater,
                R.layout.game_fragment,
                container,
                false
        )

        Log.i("GameFragment", "Called ViewModelProviders.of")
        viewModel =
ViewModelProviders.of(this).get(GameViewModel::class.java)

        binding.correctButton.setOnClickListener { onCorrect() }
        binding.skipButton.setOnClickListener { onSkip() }
        updateScoreText()
        updateWordText()
        return binding.root

    }


    /** Methods for button click handlers **/

    private fun onSkip() {
        if (!wordList.isEmpty()) {
            score--
        }
        nextWord()
    }

    private fun onCorrect() {
        if (!wordList.isEmpty()) {
            score++
        }
        nextWord()
    }


    /** Methods for updating the UI **/

    private fun updateWordText() {
        binding.wordText.text = word
    }

    private fun updateScoreText() {
        binding.scoreText.text = score.toString()
    }
}
```

Step 2: Zaktualizuj odniesienia do programów obsługi kliknięć i pól danych w GameFragment

1. W `GameFragment`, zaktualizuj metody `onSkip()` i `onCorrect()` Usuń kod, aby zaktualizować wynik, i zamiast tego wywołaj odpowiednie metody `onSkip()` i `onCorrect()` w **`viewModel`**.
2. Ponieważ przenieśliśmy metodę `nextWord()` do `ViewModel`, fragment gry nie może już uzyskać do niej dostępu.

   W `GameFragment`, w metodach `onSkip()` i `onCorrect()` zamień wywołanie

`nextWord()` na `updateScoreText()` i `updateWordText()`.Te metody wyświetlają dane na ekranie.

```
private fun onSkip() {
   viewModel.onSkip()
   updateWordText()
   updateScoreText()
}
private fun onCorrect() {
   viewModel.onCorrect()
   updateScoreText()
   updateWordText()
}
```

3. W `GameFragment`, zaktualizuj zmienne `score` i `word` , aby używały zmiennych `GameViewModel` ponieważ zmienne te są teraz w `GameViewModel`.

```
private fun updateWordText() {
   binding.wordText.text = viewModel.word
}

private fun updateScoreText() {
   binding.scoreText.text = viewModel.score.toString()
}
```

Reminder: Ponieważ działania, fragmenty i widoki aplikacji nie przetrwają zmian konfiguracji, model ViewModel nie powinien zawierać odniesień do activity, fragmentów lub widoków aplikacji.

4. W `GameViewModel`, w metodzie `nextWord()` usuń wywołania metod `updateWordText()` i `updateScoreText()` methods. Te metody są teraz wywoływane z `GameFragment`.
5. Zbuduj aplikację i upewnij się, że nie ma błędów. Jeśli masz błędy, wyczyść i odbuduj projekt. (clean and rebuild the project).
6. Uruchom aplikację i zagraj w kilka słów. Podczas wyświetlania ekranu gry obróć urządzenie. Zauważ, że bieżący wynik i bieżące słowo są zachowywane po zmianie orientacji.

Teraz wszystkie dane aplikacji są przechowywane w ViewModel, więc są zachowywane podczas zmian konfiguracji.

7. Zadanie: Zaimplementuj detektor kliknięć dla przycisku End Game

W tym zadaniu implementujesz detektor kliknięć dla przycisku Zakończ grę.

1. W `GameFragment`, dodaj metodę o nazwie `onEndGame()`.Metoda `onEndGame()` zostanie wywołana, gdy użytkownik dotknie przycisku Zakończ grę. **End Game** button.

```
private fun onEndGame() {
   }
```

2. W `GameFragment`, w metodzie`onCreateView()` zlokalizuj kod, który ustawia detektory kliknięć dla przycisków **Got It** i **Skip** buttons. Tuż pod tymi dwoma wierszami ustaw detektor kliknięć dla przycisku **End Game** button. Użyj zmiennej powiązania,, `binding`. Wewnątrz detektora kliknięć wywołaj metodę `onEndGame()`.

```
binding.endGameButton.setOnClickListener { onEndGame() }
```

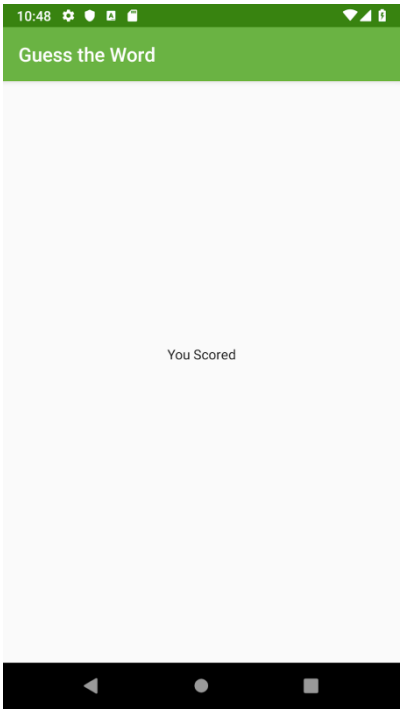
3. W `GameFragment`, dodaj metodę o nazwie `gameFinished()` aby przejść do aplikacji na score screen. Podaj wynik jako argument, używając Bezpiecznych argumentów Safe Args.

```
/**
* Called when the game is finished
*/
private fun gameFinished() {
   Toast.makeText(activity, "Game has just finished",
Toast.LENGTH_SHORT).show()
   val action = GameFragmentDirections.actionGameToScore()
   action.score = viewModel.score
   NavHostFragment.findNavController(this).navigate(action)
}
```

4. W metodzie `onEndGame()` wywołaj metodę `gameFinished()`.

```
private fun onEndGame() {
   gameFinished()
}
```

5. Uruchom aplikację, zagraj w grę i przewiń kilka słów. Naciśnij przycisk Zakończ grę. Zauważ, że aplikacja przechodzi do ekranu wyników, ale wynik końcowy nie jest wyświetlany. Naprawisz to w następnym zadaniu.



8. Zadanie: użyj ViewModelFactory

Gdy użytkownik zakończy grę, ScoreFragment nie wyświetla wyniku. Chcesz, aby ViewModel przechowywał wynik wyświetlany przez `ScoreFragment`. You'll pass in the score value during the `ViewModel` initialization using the factory method pattern.

Wzorzec metody fabrycznej (*factory method pattern)* to kreatywny wzorzec projektowy (creational design pattern), który wykorzystuje metody fabryczne do tworzenia obiektów. Metoda fabryczna (*factory metod)* to metoda zwracająca instancję tej samej klasy.

W tym zadaniu tworzysz `ViewModel` ze sparametryzowanym konstruktorem dla score fragment i metodą fabryczną, aby utworzyć instancję `ViewModel`.

1. W ramach pakietu `score` utwórz nową klasę Kotlin o nazwie `ScoreViewModel`. Ta klasa będzie `ViewModel` dla score fragment.
2. Rozszerz klasę `ScoreViewModel` z `ViewModel`. Dodaj parametr konstruktora do końcowego wyniku. Dodaj blok `init` z instrukcją dziennika.
3. W klasie `ScoreViewModel` dodaj zmienną o nazwie `score` , aby zapisać wynik końcowy.

```
class ScoreViewModel(finalScore: Int) : ViewModel() {
   // The final score
   var score = finalScore
   init {
       Log.i("ScoreViewModel", "Final score is $finalScore")
   }
}
```

4. W ramach pakietu `score` utwórz kolejną klasę Kotlin o nazwie `ScoreViewModelFactory`. Ta klasa będzie odpowiedzialna za tworzenie wystąpienia obiektu `ScoreViewModel`.
5. Rozszerz klasę `ScoreViewModelFactory` z `ViewModelProvider.Factory`. Dodaj parametr konstruktora do końcowego wyniku.

```
class ScoreViewModelFactory(private val finalScore: Int) :
ViewModelProvider.Factory {
}
```

6. W `ScoreViewModelFactory`, Android Studio wyświetla błąd dotyczący niezaimplementowanego elementu abstrakcyjnego. Aby rozwiązać problem, zastąp metodę „override” `create()` W metodzie `create()` zwróć nowo zbudowany obiekt `ScoreViewModel`.

```
override fun <T : ViewModel?> create(modelClass: Class<T>): T {
   if (modelClass.isAssignableFrom(ScoreViewModel::class.java)) {
       return ScoreViewModel(finalScore) as T
   }
   throw IllegalArgumentException("Unknown ViewModel class")
}
```

7. W `ScoreFragment`, utwórz zmienne klas dla `ScoreViewModel` i `ScoreViewModelFactory`.

```
private lateinit var viewModel: ScoreViewModel
private lateinit var viewModelFactory: ScoreViewModelFactory
```

8. W `ScoreFragment`, wewnątrz `onCreateView()`,po zainicjowaniu zmiennej powiązania `binding`, zainicjuj **`viewModelFactory`**. Użyj `ScoreViewModelFactory`. Przekaż końcowy wynik z pakietu argumentów jako parametr konstruktora do `ScoreViewModelFactory()`.

```
viewModelFactory =
ScoreViewModelFactory(ScoreFragmentArgs.fromBundle(arguments!!).score)
```

9. W funkcji `onCreateView()`,po zainicjowaniu **`viewModelFactory`**, zainicjuj obiekt **`viewModel`** object. Wywołaj metodę `ViewModelProviders.of()`, przekaż powiązany kontekst score fragment i **`viewModelFactory`**. Spowoduje to utworzenie obiektu `ScoreViewModel` przy użyciu metody fabrycznej zdefiniowanej w klasie **`viewModelFactory`**.

```
viewModel = ViewModelProviders.of(this, viewModelFactory)
       .get(ScoreViewModel::class.java)
```

10. W metodzie `onCreateView()` po zainicjowaniu **`viewModel`**, ustaw tekst widoku `scoreText` na końcowy wynik zdefiniowany w `ScoreViewModel`.

```
binding.scoreText.text = viewModel.score.toString()
```

11. Uruchom aplikację i zagraj w grę. Przełączaj niektóre lub wszystkie słowa i dotknij Zakończ grę. Zauważ, że fragment wyniku wyświetla teraz wynik końcowy.

Uwaga: W tej aplikacji nie jest absolutnie konieczne dodawanie ViewModelFactory dla ScoreViewModel, ponieważ można przypisać wynik bezpośrednio do zmiennej viewModel.score. Ale czasami potrzebujesz danych bezpośrednio po zainicjowaniu viewModel.
W zadaniu zaimplementowano ScoreFragment, aby korzystać z ViewModel. Nauczyłeś się również, jak tworzyć sparametryzowany konstruktor dla ViewModel przy użyciu interfejsu ViewModelFactory.