1.Wprowadzenie `LiveData`

W poprzednim ćwiczeniu używałeś `ViewModel` w aplikacji GuessTheWord, aby dane aplikacji mogły przetrwać zmiany konfiguracji urządzenia. W tym ćwiczeniu nauczysz się integrować `LiveData` z danymi w klasach `ViewModel` i `LiveData`, `LiveData` jest jednym ze składników architektury Android (Android Architecture Components), pozwala budować obiekty danych, które powiadamiają widoki o zmianach bazy danych.

Aby skorzystać z klasy `LiveData`, należy skonfigurować „obserwatorów" (na przykład activities lub fragmenty), które obserwują zmiany danych aplikacji. `LiveData` rozpoznaje cykl życia, więc aktualizuje tylko obserwatory składników aplikacji, które są w aktywnym stanie cyklu.

2. Czego się nauczysz

- Co sprawia, że obiekty `LiveData` są użyteczne.
- Jak dodać `LiveData` do danych przechowywanych w `ViewModel`.
- Kiedy i jak korzystać z `MutableLiveData`.
- Jak dodać metody obserwatora, aby obserwować zmiany w `LiveData`.
- Jak enkapsulować `LiveData` za pomocą backing property.
- Jak komunikować się między kontrolerem interfejsu użytkownika a odpowiadającym mu modułem `ViewModel`.

Co będziesz robić

- Use `LiveData` for the word and the score in the GuessTheWord app.
- Add observers that notice when the word or the score changes.
- Update the text views that display changed values.
- Use the `LiveData` observer pattern to add a game-finished event.
- Implement the **Play Again** button.

3. W tym zadaniu możesz użyć aplikacji GuessTheWord, którą wbudowałeś w części a laboratorium.

4. Zadanie: dodaj LiveData do GameViewModel

`LiveData` (observable data holder class) to obserwowalna klasa posiadacza danych, która rozpoznaje cykl życia. Na przykład możesz owinąć (wrap) LiveData wokół bieżącego wyniku w aplikacji GuessTheWord. W tym ćwiczeniu poznasz kilka cech `LiveData`:

- `LiveData` jest obserwowalna, co oznacza, że obserwator jest powiadamiany o zmianie danych przechowywanych przez obiekt `LiveData`.
- `LiveData` przechowuje dane; `LiveData` o opakowanie (wrapper), którego można używać z dowolnymi danymi
- `LiveData` jest świadomy cyklu życia, co oznacza, że aktualizuje tylko obserwatorów, którzy są w stanie aktywnego cyklu życia, takich jak `STARTED` lub `RESUMED`.

W tym zadaniu nauczysz się, jak zawijać dowolny typ danych w obiekty `LiveData` , konwertując dane score i current word w `GameViewModel` do `LiveData`. W późniejszym zadaniu dodajesz obserwatora do tych obiektów `LiveData` i nauczysz się, jak obserwować `LiveData`.

Step 1: Zmień score i word aby korzystać z LiveData

1. Pod pakietem `screens/game` package, otwórz plik `GameViewModel`.
2. Zmień typ zmiennych `score` i `word` na `MutableLiveData`.

   `MutableLiveData` to `LiveData` , którego wartość można zmienić. `MutableLiveData` to klasa ogólna(generic class), dlatego należy określić typ przechowywanych danych..

```
// The current word
val word = MutableLiveData<String>()
// The current score
val score = MutableLiveData<Int>()
```

3. W `GameViewModel`, wewnątrz bloku `init` block, zainicjuj `score` i `word`. Aby zmienić wartość zmiennej `LiveData`, użyj metody `setValue()` w zmiennej. W Kotlin można wywołać `setValue()` za pomocą właściwości value.

```
init {

   word.value = ""
   score.value = 0
  ...
}
```

Step 2: Zaktualizuj odwołanie do obiektu LiveData

Zmienne `score` i `word` są teraz typu `LiveData`. W tym kroku zmieniasz odwołania do tych zmiennych, używając właściwości `value`.

1. W `GameViewModel`, w metodzie `onSkip()` zmień `score` na `score.value`. Zauważ że `score` może być `null`.
2. Aby uniknąć błędu `null` uzyj konstrukcji Kotlin z pytajnikiem: `null`-safety.

```
fun onSkip() {
   if (!wordList.isEmpty()) {
       score.value = (score.value)?.minus(1)
   }
   nextWord()
}
```

3. Zaktualizuj metodę `onCorrect()` w ten sam sposób: użyj funkcji `plus()`.

```
fun onCorrect() {
   if (!wordList.isEmpty()) {
       score.value = (score.value)?.plus(1)
   }
   nextWord()
}
```

4. W `GameViewModel`, w metodzie `nextWord()` zmień odwołanie do `word` na **`word`**.*`value`*.

```
private fun nextWord() {
   if (!wordList.isEmpty()) {
       //Select and remove a word from the list
       word.value = wordList.removeAt(0)
   }
}
```

5. W `GameFragment`, w metodzie `updateWordText()` w metodzie **`viewModel`**`.word` na **`viewModel.word`**.*`value`*.

```
/** Methods for updating the UI **/
private fun updateWordText() {
   binding.wordText.text = viewModel.word.value
}
```

6. W `GameFragment`, w metodzie `updateScoreText()` zmień odwołanie do **`viewModel`**`.score` na **`viewModel.score`**.*`value`*.

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

7. W `GameFragment`, w metodzie `gameFinished()` zmień odniesienie do **`viewModel`**`.score` na **`viewModel.score`**.*`value`*. Dodaj wymaganą kontrolę bezpieczeństwa (`null`-safety check).

```
private fun gameFinished() {
   Toast.makeText(activity, "Game has just finished",
Toast.LENGTH_SHORT).show()
   val action = GameFragmentDirections.actionGameToScore()
   action.score = viewModel.score.value?:0
   NavHostFragment.findNavController(this).navigate(action)
}
```

8. Upewnij się, że w kodzie nie ma błędów. Skompiluj i uruchom swoją aplikację. Funkcjonalność aplikacji powinna być taka sama jak wcześniej.

5. Zadanie: dołącz obserwatorów do obiektów LiveData

To zadanie jest ściśle powiązane z poprzednim zadaniem, w którym przekształcono dane score i word w obiekty `LiveData`. W tym zadaniu dołączasz obiekty `Observer` do tych obiektów `LiveData`.

1. W `GameFragment`, w metodzie `onCreateView()` dołącz obiekt `Observer` do obiektu `LiveData` dla bieżącego wyniku, `viewModel.score`. Użyj metody `observe()` i umieść kod po inicjalizacji `viewModel`. Użyj wyrażenia lambda (lambda expression), aby uprościć kod. (Wyrażenie lambda jest anonimową funkcją, która nie jest zadeklarowana, ale jest przekazywana natychmiast jako wyrażenie).

```
viewModel.score.observe(this, Observer { newScore ->
```

```
})
```

Rozwiąż odniesienie do obserwatora (`Observer`). Aby to zrobić, kliknij `Observer`, naciśnij `Alt+Enter` , i zaimportuj `androidx.lifecycle.Observer`.

2. Nowo utworzony obserwator odbiera zdarzenie, gdy zmienią się dane przechowywane przez obserwowany obiekt `LiveData` Wewnątrz obserwatora zaktualizuj score `TextView` na nowy score.

```
/** Setting up LiveData observation relationship **/
viewModel.score.observe(this, Observer { newScore ->
   binding.scoreText.text = newScore.toString()
})
```

3. Dołącz obiekt `Observer` do bieżącego słowa obiekt `LiveData` Zrób to w ten sam sposób, w jaki dołączyłeś obiekt `Observer` do bieżącego wyniku.

```
/** Setting up LiveData observation relationship **/
viewModel.word.observe(this, Observer { newWord ->
   binding.wordText.text = newWord
})
```

Gdy wartość `score` lub `word` zmienia się, `score` lub `word` wyświetlane na ekranie są teraz aktualizowane automatycznie.

4. W `GameFragment`, usuń metody `updateWordText()` i `updateScoreText()`,oraz wszystkie odniesienia do nich. Już ich nie potrzebujesz, ponieważ widoki tekstowe są aktualizowane metodami obserwatora `LiveData`.
5. Uruchom aplikację. Twoja aplikacja powinna działać dokładnie tak jak poprzednio, ale teraz korzysta z `LiveData` i `LiveData` observers.

6. Zadanie: Hermetyzuj (encapsulate) LiveData

Encapsulation Hermetyzacja to sposób na ograniczenie bezpośredniego dostępu do niektórych pól obiektu. Podczas enkapsulacji obiektu ujawnia się zestaw publicznych metod modyfikujących prywatne pola wewnętrzne. Za pomocą enkapsulacji kontrolujesz, w jaki sposób inne klasy manipulują tymi wewnętrznymi polami.

W bieżącym kodzie dowolna klasa zewnętrzna może modyfikować zmienne score i word przy użyciu właściwości value, na przykład przy użyciu viewModel.score.value. W aplikacji opracowywanej w tym kodzie kodu może nie mieć znaczenia, ale w aplikacji produkcyjnej chcesz kontrolować dane w obiektach ViewModel.

Tylko ViewModel powinien edytować dane w Twojej aplikacji. Jednak kontrolery interfejsu użytkownika muszą czytać dane, więc pola danych nie mogą być całkowicie prywatne. Do enkapsulacji danych aplikacji używasz zarówno obiektów `MutableLiveData` jak i `LiveData`.

`MutableLiveData` vs. `LiveData`:

- Dane w obiekcie MutableLiveData można zmieniać, jak sugeruje nazwa. Wewnątrz ViewModel dane powinny być edytowalne, więc ViewModel używa `MutableLiveData`.
- Dane w obiekcie LiveData można odczytać, ale nie można ich zmienić. Dane spoza ViewModel powinny być czytelne, ale nie edytowalne, więc dane powinny być widoczne jako `LiveData`.

Aby zrealizować tę strategię, korzystasz z właściwości Kotlin backing property. Właściwość kopii zapasowej pozwala zwrócić coś z elementu (getter) innego niż dokładny obiekt. W tym zadaniu implementujesz właściwość backing property dla obiektów `score` i `word`.

Dodaj backing property do score i word

1. W `GameViewModel`, ustaw obiekt `score` na `private`.
2. Aby postępować zgodnie z konwencją nazewnictwa stosowaną we właściwościach kopii zapasowej, zmień `score` na `_score`. Właściwość `_score` jest teraz zmienną wersją (mutable version) wyniku gry, do użytku wewnętrznego.
3. Utwórz publiczną wersję typu `LiveData` zwaną `score`.

```
// The current score
private val _score = MutableLiveData<Int>()
val score: LiveData<Int>
```

4. Widzisz błąd inicjalizacji. Ten błąd występuje, ponieważ w `GameFragment`, **`score`** jest referencją `LiveData` reference, a `score` nie może już uzyskać dostępu do swojego ustawiacza (setter). Aby dowiedzieć się więcej na temat :Getters and Setters.

   Aby rozwiązać błąd, zastąp metodę `get()` dla obiektu `score` w `GameViewModel` i zwróć właściwość, `_score`.

```
val score: LiveData<Int>
    get() = _score
```

5. W `GameViewModel`, zmień odniesienia do `score` na jego wewnętrzną, zmienną wersję, (mutable version), `_score`.

```
init {
    ...
    _score.value = 0
    ...
}

...
fun onSkip() {
    if (!wordList.isEmpty()) {
        _score.value = (score.value)?.minus(1)
    }
   ...
}

fun onCorrect() {
    if (!wordList.isEmpty()) {
        _score.value = (score.value)?.plus(1)
```

```
    }
    ...
}
```

6. Zmień nazwę obiektu `word` na `_word` i dodaj dla niego backing property , tak jak w przypadku obiektu `score`.

```
// The current word
private val _word = MutableLiveData<String>()
val word: LiveData<String>
    get() = _word
...
init {
    _word.value = ""
    ...
}
...
private fun nextWord() {
    if (!wordList.isEmpty()) {
        //Select and remove a word from the list
        _word.value = wordList.removeAt(0)
    }
}
```

Świetna robota, obiekty `LiveData` `word` i `score` są teraz hermetyzowane.

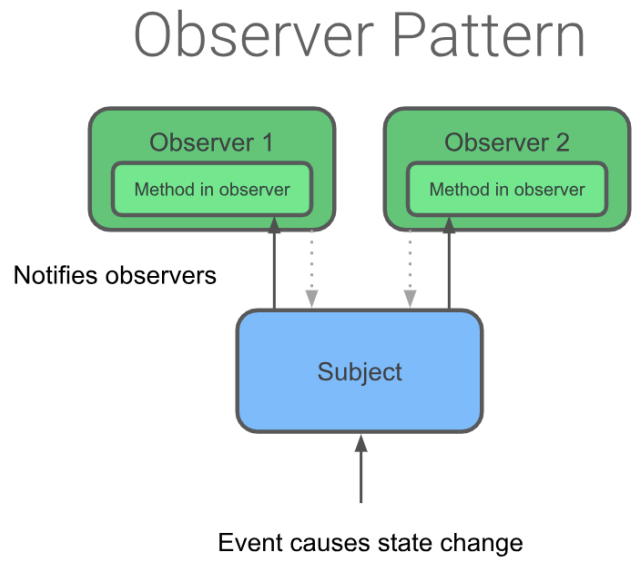
7. Zadanie: Dodaj zdarzenie zakończenia gry.

Twoja bieżąca aplikacja przechodzi do ekranu wyników, gdy użytkownik stuknie przycisk Zakończ grę. Chcesz również, aby aplikacja nawigowała do ekranu wyników, gdy gracze przejdą przez wszystkie słowa. Gdy gracze dojdą do ostatniego słowa, chcesz, aby gra zakończyła się automatycznie, aby użytkownik nie musiał naciskać przycisku.

, gdy wszystkie słowa zostaną wyświetlone. Aby to zrobić, użyj wzorca obserwatora LiveData do modelowania zdarzenia: (game-finished event).

Wzór obserwatora (The observer pattern)

Observer pattern Wzorzec obserwatora to wzorzec projektowania oprogramowania. Określa komunikację między obiektami: obserwowalnym („podmiotem” obserwacji) i obserwatorami. Obserwowalny to obiekt, który powiadamia obserwatorów o zmianach jego stanu.



W przypadku LiveData w tej aplikacji obserwowalnym (podmiotem) jest obiekt LiveData, a obserwatorami są metody w kontrolerach interfejsu użytkownika, takie jak fragmenty. Zmiana stanu następuje za każdym razem, gdy zmieniają się dane opakowane w LiveData. Klasy LiveData odgrywają kluczową rolę w komunikacji z ViewModel do fragmentu.

Krok 1: Użyj LiveData do wykrycia zdarzenia game-finished

W tym zadaniu używasz wzorca obserwatora LiveData do modelowania zdarzenia końca gry (game-finished event).

1. W `GameViewModel`, utwórz obiekt `Boolean MutableLiveData` o nazwie **`_eventGameFinish`**. Ten obiekt będzie gospodarzem wydarzenia game-finished.
2. Po zainicjowaniu obiektu **`_eventGameFinish`** utwórz i zainicjuj właściwość kopii zapasowej (backing property) o nazwie **`eventGameFinish`**.

```
// Event which triggers the end of the game
private val _eventGameFinish = MutableLiveData<Boolean>()
val eventGameFinish: LiveData<Boolean>
   get() = _eventGameFinish
```

3. W `GameViewModel`, dodaj metodę `onGameFinish()` W metodzie ustaw zdarzenie game-finished, **`eventGameFinish`**, na `true`.

```
/** Method for the game completed event **/
fun onGameFinish() {
   _eventGameFinish.value = true
}
```

4. W `GameViewModel`, w metodzie `nextWord()` zakończ grę, jeśli lista słów jest pusta.

```
private fun nextWord() {
   if (wordList.isEmpty()) {
       onGameFinish()
```

```
    } else {
        //Select and remove a _word from the list
        _word.value = wordList.removeAt(0)
    }
}
```

5. W `GameFragment` , wewnątrz `onCreateView()`,po zainicjowaniu `viewModel`, dołącz obserwatora do `eventGameFinish`. Użyj metody `observe()`. Wewnątrz funkcji lambda wywołaj metodę `gameFinished()`.

```
// Observer for the Game finished event
viewModel.eventGameFinish.observe(this, Observer<Boolean> { hasFinished ->
    if (hasFinished) gameFinished()
})
```

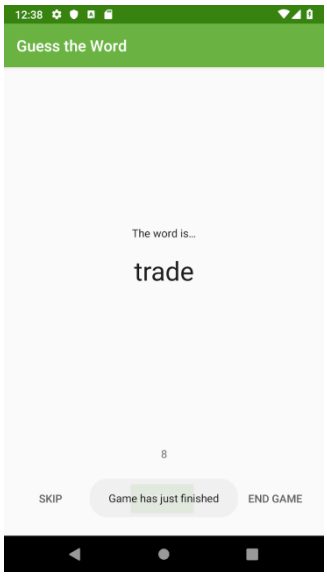
6. Uruchom aplikację, zagraj w grę i przejrzyj wszystkie słowa. Aplikacja automatycznie przechodzi do ekranu wyników, zamiast pozostać w fragmencie gry, dopóki nie dotkniesz opcji **End Game**.

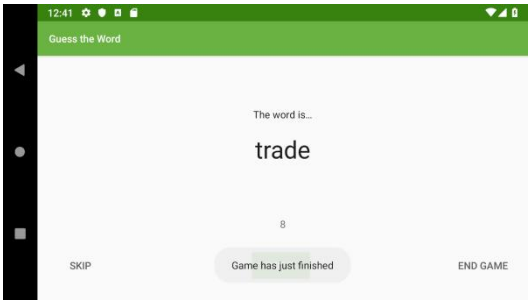
   Gdy lista słów jest pusta, ustawiane jest, **`eventGameFinish`** wywoływana jest powiązana metoda obserwatora we fragmencie gry, a aplikacja przechodzi do screen fragment.
7. Dodany kod wprowadził problem cyklu życia. Aby zrozumieć problem, w klasie `GameFragment` skomentuj kod nawigacyjny w metodzie `gameFinished()` Pamiętaj, aby zachować wiadomość `Toast` w metodzie.

```
private fun gameFinished() {
       Toast.makeText(activity, "Game has just finished",
Toast.LENGTH_SHORT).show()
//        val action = GameFragmentDirections.actionGameToScore()
//        action.score = viewModel.score.value?:0
//        NavHostFragment.findNavController(this).navigate(action)
    }
```

8. Uruchom aplikację, zagraj w grę i przejrzyj wszystkie słowa. Na dole ekranu gry pojawia się na krótko komunikat "Game has just finished" Jest to oczekiwane zachowanie.

Teraz obróć urządzenie lub emulator. Toast pojawia się ponownie! Obróć urządzenie jeszcze kilka razy, a prawdopodobnie za każdym razem zobaczysz toast. Jest to błąd, ponieważ toast powinien wyświetlać się tylko raz, po zakończeniu gry. Tosty nie powinny być wyświetlane za każdym razem, gdy fragment jest odtwarzany ponownie. Rozwiązujesz ten problem w następnym zadaniu.





Step 2: Zresetuj wydarzenie game-finished event

Zwykle LiveData dostarcza obserwatorom aktualizacje tylko w przypadku zmiany danych. Wyjątkiem od tego zachowania jest to, że obserwatorzy otrzymują również aktualizacje, gdy obserwator zmienia stan z nieaktywnego na aktywny.

Właśnie dlatego zakończony grą toast jest wielokrotnie uruchamiany w Twojej aplikacji. Gdy fragment gry zostanie ponownie utworzony po obrocie ekranu, przechodzi z stanu nieaktywnego do aktywnego. Obserwator we fragmencie jest ponownie połączony z istniejącym `ViewModel` i odbiera bieżące dane. Metoda `gameFinished()` zostaje ponownie uruchomiona i pojawia się toast.

W tym zadaniu rozwiązujesz ten problem i wyświetlasz toast tylko raz, resetując flagę **`eventGameFinish`** flag w `GameViewModel`.

1. w `GameViewModel`, dodaj metodę `onGameFinishComplete()` aby zresetować zdarzenie zakończenia gry, **`_eventGameFinish`**.

```
/** Method for the game completed event **/

fun onGameFinishComplete() {
    _eventGameFinish.value = false
}
```

2. W `GameFragment`, na końcu `gameFinished()`,wywołaj `onGameFinishComplete()` na obiekcie **`viewModel`** (Pozostaw kod nawigacyjny w `gameFinished()` na razie skomentowany).

```
private fun gameFinished() {
    ...
    viewModel.onGameFinishComplete()
}
```

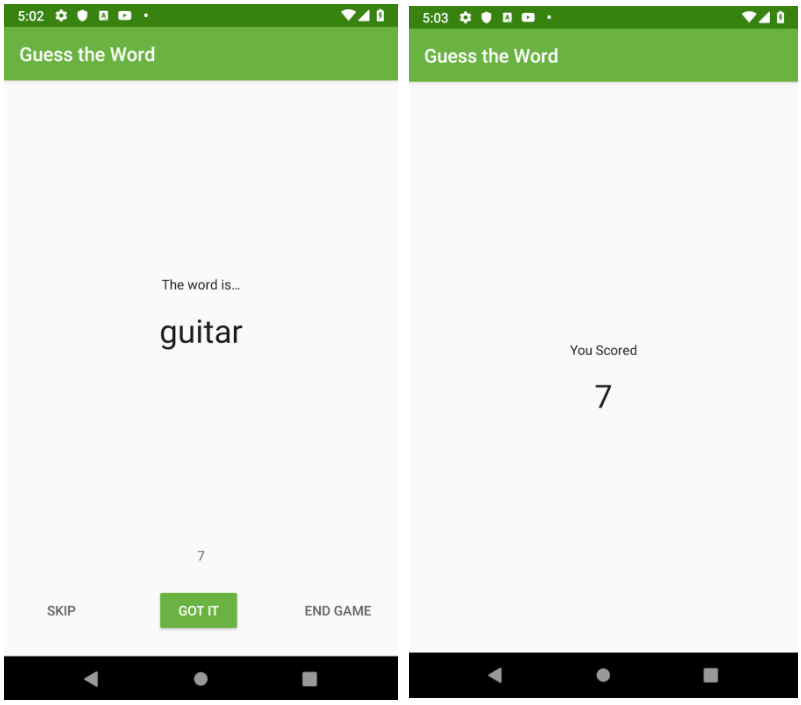
3. Uruchom aplikację i zagraj w grę. Przejrzyj wszystkie słowa, a następnie zmień orientację ekranu urządzenia. Tost jest wyświetlany tylko raz.
4. W `GameFragment`, w metodzie `gameFinished()` usuń komentarz z kodu nawigacji.

   Aby anulować komentarz w Android Studio, zaznacz linie, które zostały zakomentowane i naciśnij klawisze `Control+`.

```
private fun gameFinished() {
   Toast.makeText(activity, "Game has just finished",
Toast.LENGTH_SHORT).show()
   val action = GameFragmentDirections.actionGameToScore()
   action.score = viewModel.score.value?:0
   findNavController(this).navigate(action)
   viewModel.onGameFinishComplete()
}
```

Jeśli Android Studio wyświetli monit, zaimportuj `androidx.navigation.fragment.NavHostFragment.findNavController`.

5. Uruchom aplikację i zagraj w grę. Upewnij się, że aplikacja automatycznie przechodzi do ekranu końcowego wyniku po przejściu wszystkich słów.



8. Zadanie: dodaj LiveData do ScoreViewModel

W tym zadaniu zmienisz score na obiekt `LiveData` w `ScoreViewModel` i dołączysz do niego obserwatora. To zadanie jest podobne do tego, co zrobiłeś po dodaniu `LiveData` do `GameViewModel`.

Wprowadzasz te zmiany w `ScoreViewModel` dla kompletności, aby wszystkie dane w Twojej aplikacji korzystały z `LiveData`.

1. W `ScoreViewModel`, zmień typ zmiennej `score` na `MutableLiveData`. Zmień nazwę zgodnie z konwencją na **`_score`** i dodaj właściwość backing property.

```
private val _score = MutableLiveData<Int>()
val score: LiveData<Int>
   get() = _score
```

2. W `ScoreViewModel`, , wewnątrz bloku `init` zainicjuj **`_score`**. Możesz usunąć lub pozostawić dziennik w bloku inicjującym, jak chcesz.`init`.

```
init {
   _score.value = finalScore
}
```

3. W `ScoreFragment`, , wewnątrz `onCreateView()`,po zainicjowaniu **`viewModel`**, dołącz obserwatora do obiektu score `LiveData` object. Wewnątrz wyrażenia lambda ustaw wartość score w score text view. Remove Usuń kod, który bezpośrednio przypisuje text view wartość score z `ViewModel`.

Kod do dodania:

```
// Add observer for score
viewModel.score.observe(this, Observer { newScore ->
   binding.scoreText.text = newScore.toString()
})
```

Kod do usunięcia:

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

Po wyświetleniu monitu przez Android Studio zaimportuj `androidx.lifecycle.Observer`.

4. Uruchom aplikację i zagraj w grę. Aplikacja powinna działać jak poprzednio, ale teraz używa `LiveData` obserwatora do aktualizacji wyniku.

9. Zadanie: Dodaj przycisk Play Again

W tym zadaniu dodsz przycisk Odtwórz ponownie do ekranu wyników i zaimplementujesz detektor kliknięć za pomocą zdarzenia `LiveData` event. Przycisk uruchamia zdarzenie, aby przejść z ekranu wyników do ekranu gry.

Kod startowy aplikacji zawiera przycisk **Play Again**, ale przycisk jest ukryty.

W `res/layout/score_fragment.xml`, dla przycisku **`play_again_button`** zmień wartość atrybutu **`visibility`** na **`visible`**.

```
<Button
   android:id="@+id/play_again_button"
...
   android:visibility="visible"
 />
```

2. W `ScoreViewModel`, dodaj obiekt `LiveData` , aby przechować wartość logiczną o nazwie **`_eventPlayAgain`**. Ten obiekt służy do zapisywania zdarzenia `LiveData` w celu przejścia z ekranu wyników do ekranu gry.

```
private val _eventPlayAgain = MutableLiveData<Boolean>()
val eventPlayAgain: LiveData<Boolean>
   get() = _eventPlayAgain
```

3. In `ScoreViewModel`, define methods to set and reset the event, **`_eventPlayAgain`**.

```
fun onPlayAgain() {
   _eventPlayAgain.value = true
}
fun onPlayAgainComplete() {
   _eventPlayAgain.value = false
}
```

4. W `ScoreFragment`, dodaj obserwatora dla `eventPlayAgain`. Umieść kod na końcu `onCreateView()`,przed instrukcją `return` Wewnątrz wyrażenia lambda wróć do ekranu gry i zresetuj `eventPlayAgain`.

```
// Navigates back to game when button is pressed
viewModel.eventPlayAgain.observe(this, Observer { playAgain ->
   if (playAgain) {
      findNavController().navigate(ScoreFragmentDirections.actionRestart())
       viewModel.onPlayAgainComplete()
   }
})
```

Zaimportuj `androidx.navigation.fragment.findNavController`, gdy pojawi się monit z Android Studio.

5. W `ScoreFragment`, wewnątrz funkcji `onCreateView()`,dodaj detektor kliknięć do przycisku **PlayAgain** i wywołaj **`viewModel`**`.onPlayAgain()`.

```
binding.playAgainButton.setOnClickListener {  viewModel.onPlayAgain()  }
```

6. Uruchom aplikację i zagraj w grę. Po zakończeniu gry ekran wyników pokazuje wynik końcowy i przycisk Odtwórz ponownie. Naciśnij przycisk PlayAgain, a aplikacja przejdzie do ekranu gry, abyś mógł ponownie grać w grę.