MODUL 8

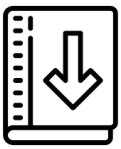
ViewModel dan ViewModelProvider

CAPAIAN PEMBELAJARAN

1. Mahasiswa mampu menggunakan ViewModel dan ViewModelPrivider dalam aplikasi untuk menyimpan dan mengelola data terkait UI

KEBUTUHAN ALAT/BAHAN/SOFTWARE

1. Android Studio 3.5
2. Handphone Android versi 7.0 (Nougat)
3. Kabel data USB.
4. Driver ADB.



DASAR TEORI

1. Selamat Datang

Anda menggunakan kelas **ViewModel** untuk menyimpan dan mengelola data terkait UI dengan cara yang sadar siklus. Kelas **ViewModel** memungkinkan data bertahan dari perubahan konfigurasi perangkat seperti rotasi layar dan perubahan pada ketersediaan keyboard.

Anda menggunakan kelas **ViewModelFactory** untuk membuat instance dan mengembalikan objek **ViewModel** yang bertahan dari perubahan konfigurasi.

Apa yang akan Anda pelajari

- Cara menggunakan arsitektur aplikasi Android yang direkomendasikan.
- Cara menggunakan kelas **Lifecycle**, **ViewModel**, dan **ViewModelFactory** di aplikasi Anda.
- Cara mempertahankan data UI melalui perubahan konfigurasi perangkat.
- Apa pola desain metode pabrik dan bagaimana menggunakannya.
- Cara membuat objek **ViewModel** menggunakan antarmuka **ViewModelProvider.Factory**.

Apa yang akan Anda lakukan

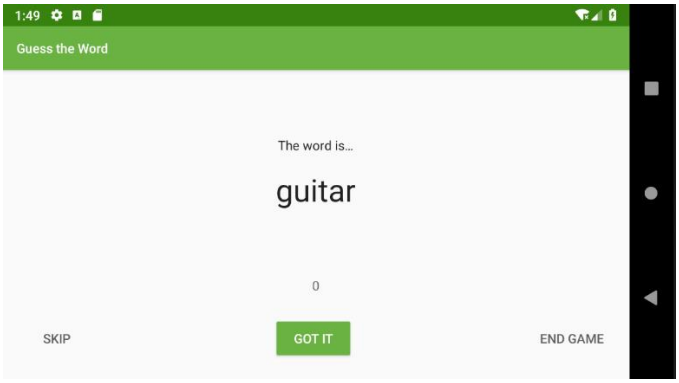
- Tambahkan **ViewModel** ke aplikasi, untuk menyimpan data aplikasi sehingga data bertahan dari perubahan konfigurasi.
- Gunakan **ViewModelFactory** dan pola desain metode pabrik untuk membuat instance objek **ViewModel** dengan parameter konstruktor.

2. Ikhtisar aplikasi

Dalam codelab Pelajaran 5 ini, Anda mengembangkan aplikasi **GuessTheWord**, dimulai dengan kode permulaan. **GuessTheWord** adalah permainan gaya sandiwara dua pemain, di mana para pemain berkolaborasi untuk mencapai skor setinggi mungkin.

Pemain pertama melihat kata-kata di aplikasi dan memerankannya secara bergantian, pastikan untuk tidak menampilkan kata tersebut ke pemain kedua. Pemain kedua mencoba menebak kata tersebut.

Untuk memainkan game, pemain pertama membuka aplikasi di perangkat dan melihat sebuah kata, misalnya "guitar", seperti yang ditunjukkan pada gambar di bawah.



Pemain pertama memerankan kata, berhati-hatilah agar tidak benar-benar mengucapkan kata itu sendiri.

- Saat pemain kedua menebak kata dengan benar, pemain pertama menekan tombol **Got It**, yang menambah hitungan satu dan menampilkan kata berikutnya.
- Jika pemain kedua tidak dapat menebak kata, pemain pertama menekan tombol **Skip**, yang akan mengurangi hitungan satu kali dan melompat ke kata berikutnya.
- Untuk mengakhiri permainan, tekan tombol **End Game**. (Fungsi ini tidak ada dalam kode awal untuk codelab pertama dalam seri ini.)

PRAKTIK

3. Tugas: Jelajahi kode starter

Dalam tugas ini, Anda mendownload dan menjalankan aplikasi starter dan memeriksa kodenya.

Langkah 1: Memulai

1. Unduh kode GuessTheWord-Starter dan buka proyek di Android Studio.

2. Jalankan aplikasi di perangkat yang diberdayakan Android, atau di emulator.
3. Ketuk tombolnya. Perhatikan bahwa tombol **Skip** menampilkan kata berikutnya dan mengurangi skor satu per satu, dan tombol **Got It** menampilkan kata berikutnya dan meningkatkan skor satu per satu. Tombol **End Game** tidak diterapkan, jadi tidak ada yang terjadi saat Anda mengetuknya.

Langkah 2: Lakukan panduan kode

1. Di Android Studio, jelajahi kode untuk merasakan cara kerja aplikasi.
2. Pastikan untuk melihat file yang dijelaskan di bawah ini, yang sangat penting.

MainActivity.kt

File ini hanya berisi kode default yang dihasilkan template.

res / layout / main_activity.xml

File ini berisi tata letak utama aplikasi. **NavHostFragment** menghosting fragmen lain saat pengguna menavigasi aplikasi.

Fragmen UI

Kode awal memiliki tiga fragmen dalam tiga paket berbeda di bawah paket **com.example.android.guesstheword.screens**:

- **title/TitleFragment** untuk layar judul
- **game/GameFragment** untuk layar game
- **score/ScoreFragment** untuk layar skor

screens/title/TitleFragment.kt

Fragmen judul adalah layar pertama yang ditampilkan saat aplikasi diluncurkan. Penangan klik diatur ke tombol **Play**, untuk menavigasi ke layar permainan.

screens/game/GameFragment.kt

Ini adalah fragmen utama, di mana sebagian besar aksi permainan berlangsung:

- Variabel ditentukan untuk kata saat ini dan skor saat ini.
- **WordList** yang didefinisikan di dalam metode **resetList()** adalah contoh daftar kata yang akan digunakan dalam permainan.
- Metode **onSkip()** adalah pengendali klik untuk tombol **Skip**. Ini mengurangi skor sebesar 1, kemudian menampilkan kata berikutnya menggunakan metode **nextWord()**.
- Metode **onCorrect()** adalah pengendali klik untuk tombol **Got It**. Metode ini diimplementasikan mirip dengan metode **onSkip()**. Satu-satunya perbedaan adalah bahwa metode ini menambahkan 1 ke skor.

screens/score/ScoreFragment.kt

ScoreFragment adalah layar terakhir dalam permainan, dan ini menampilkan skor akhir pemain. Dalam codelab ini, Anda menambahkan implementasi untuk menampilkan layar ini dan menunjukkan skor akhir.

res/navigation/main_navigation.xml

Grafik navigasi menunjukkan bagaimana fragmen terhubung melalui navigasi:

- Dari fragmen judul, pengguna dapat membuka fragmen game.
- Dari fragmen game, pengguna dapat membuka fragmen skor.
- Dari fragmen skor, pengguna dapat menavigasi kembali ke fragmen game.

4. Tugas: Temukan masalah di aplikasi starter

Dalam tugas ini, Anda menemukan masalah dengan aplikasi starter **GuessTheWord**.

1. Jalankan kode awal dan mainkan permainan melalui beberapa kata, ketuk **Skip** atau **Got It** setelah setiap kata.

2. Layar permainan sekarang menunjukkan kata dan skor saat ini. Ubah orientasi layar dengan memutar perangkat atau emulator. Perhatikan bahwa skor saat ini hilang.
3. Jalankan permainan melalui beberapa kata lagi. Saat layar game ditampilkan dengan beberapa skor, tutup dan buka kembali aplikasi. Perhatikan bahwa permainan dimulai ulang dari awal, karena status aplikasi tidak disimpan.
4. Mainkan game melalui beberapa kata, lalu ketuk tombol **End Game**. Perhatikan bahwa tidak ada yang terjadi.

Masalah di aplikasi:

- Aplikasi pemula tidak menyimpan dan memulihkan status aplikasi selama perubahan konfigurasi, seperti saat orientasi perangkat berubah, atau saat aplikasi dimatikan dan dimulai ulang. Anda bisa mengatasi masalah ini menggunakan callback **onSaveInstanceState()**. Namun, menggunakan metode **onSaveInstanceState()** mengharuskan Anda menulis kode tambahan untuk menyimpan status dalam bundel, dan menerapkan logika untuk mengambil status tersebut. Selain itu, jumlah data yang dapat disimpan minimal.
- Layar game tidak mengarah ke layar skor saat pengguna mengetuk tombol **End Game**.

Anda dapat menyelesaikan masalah ini menggunakan komponen arsitektur aplikasi yang Anda pelajari di codelab ini.

Arsitektur aplikasi

Arsitektur aplikasi adalah cara mendesain kelas aplikasi Anda, dan hubungan di antara mereka, sehingga kode diatur, bekerja dengan baik dalam skenario tertentu, dan mudah digunakan. Dalam kumpulan empat codelab ini, penyempurnaan yang Anda lakukan pada aplikasi **GuessTheWord** mengikuti pedoman arsitektur aplikasi Android, dan Anda menggunakan Komponen Arsitektur Android. Arsitektur aplikasi Android mirip dengan pola arsitektur **MVVM** (**model-view-viewmodel**).

Aplikasi **GuessTheWord** mengikuti prinsip desain pemisahan masalah dan dibagi menjadi beberapa kelas, dengan setiap kelas menangani masalah yang terpisah. Dalam codelab pertama pelajaran ini, kelas yang Anda gunakan adalah **UI controller**, **ViewModel**, dan **ViewModelFactory**.

UI Controller

Pengontrol UI (UI controller) adalah kelas berbasis UI seperti Aktivitas atau Fragmen. Pengontrol UI hanya boleh berisi logika yang menangani UI dan interaksi sistem operasi seperti menampilkan tampilan dan menangkap input pengguna. Jangan letakkan logika pengambilan keputusan, seperti logika yang menentukan teks yang akan ditampilkan, ke dalam pengontrol UI.

Dalam kode awal GuessTheWord, pengontrol UI terdiri dari tiga fragmen: **GameFragment**, **ScoreFragment**, dan **TitleFragment**. Mengikuti prinsip desain "pemisahan perhatian", **GameFragment** hanya bertanggung jawab untuk menggambar elemen game ke layar dan mengetahui kapan pengguna mengetuk tombol, dan tidak lebih. Saat pengguna mengetuk tombol, informasi ini diteruskan ke **GameViewModel**.

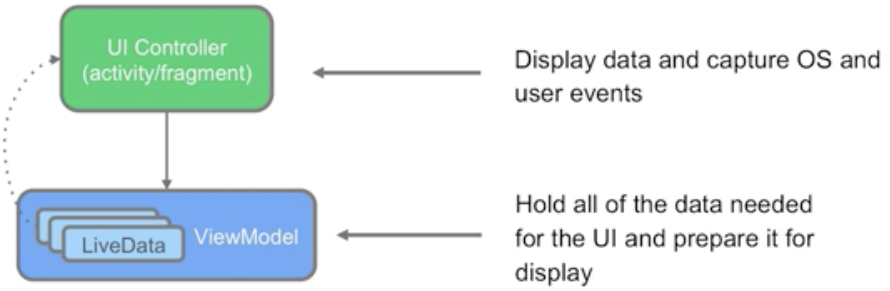
ViewModel

ViewModel menyimpan data untuk ditampilkan dalam fragmen atau aktivitas yang terkait dengan ViewModel. ViewModel dapat melakukan penghitungan dan transformasi sederhana pada data untuk menyiapkan data untuk ditampilkan oleh pengontrol UI. Dalam arsitektur ini, ViewModel melakukan pengambilan keputusan.

GameViewModel menyimpan data seperti nilai skor, daftar kata, dan kata saat ini, karena ini adalah data yang akan ditampilkan di layar. GameViewModel juga berisi logika bisnis untuk melakukan penghitungan sederhana guna memutuskan bagaimana status data saat ini.

ViewModelFactory

Sebuah **ViewModelFactory** membuat instance objek **ViewModel**, dengan atau tanpa parameter konstruktor.



Di codelab selanjutnya, Anda mempelajari Komponen Arsitektur Android lain yang terkait dengan pengontrol UI dan ViewModel.

5. Tugas: Membuat GameViewModel

Kelas **ViewModel** dirancang untuk menyimpan dan mengelola data terkait UI. Dalam aplikasi ini, setiap ViewModel dikaitkan dengan satu fragmen.

Dalam tugas ini, Anda menambahkan ViewModel pertama ke aplikasi Anda, **GameViewModel** untuk **GameFragment**. Anda juga mempelajari apa artinya **ViewModel sadar-siklus**.

Langkah 1: Tambahkan kelas GameViewModel

1. Buka file **build.gradle (module: app)**. Di dalam blok **dependencies**, tambahkan dependensi Gradle untuk **ViewModel**.

Jika Anda menggunakan versi terbaru pustaka, aplikasi solusi harus dikompilasi seperti yang diharapkan. Jika tidak, coba selesaikan masalah, atau kembalikan ke versi yang ditunjukkan di bawah.

```
//ViewModel
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0'
```

2. Di paket folder **screens/game/**, buat kelas Kotlin baru bernama **GameViewModel**.
3. Buat kelas **GameViewModel** memperluas kelas abstrak **ViewModel**.
4. Untuk membantu Anda lebih memahami bagaimana **ViewModel** sadar siklus hidup, tambahkan blok **init** dengan pernyataan **log**.

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

Langkah 2: Ganti onCleared () dan tambahkan logging

ViewModel dimusnahkan saat fragmen terkait dilepas, atau saat aktivitas selesai. Tepat sebelum ViewModel dihancurkan, callback **onCleared()** dipanggil untuk membersihkan sumber daya.

1. Di kelas **GameViewModel**, override metode **onCleared ()**.
2. Tambahkan pernyataan log di dalam **onCleared()** untuk melacak siklus hidup **GameViewModel**.

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

Langkah 3: Kaitkan GameViewModel dengan fragmen game

ViewModel harus dikaitkan dengan pengontrol UI. Untuk mengaitkan keduanya, Anda membuat referensi ke **ViewModel** di dalam pengontrol UI.

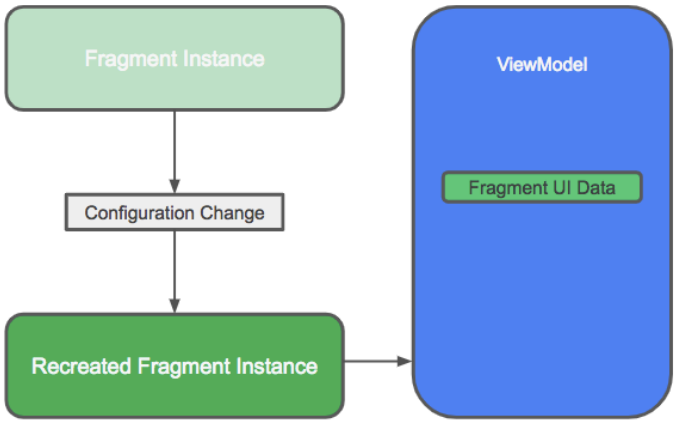
Pada langkah ini, Anda membuat referensi **GameViewModel** di dalam pengontrol UI yang sesuai, yaitu **GameFragment**.

1. Di kelas **GameFragment**, tambahkan field tipe **GameViewModel** di bagian atas sebagai variabel kelas.

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

Langkah 4: Inisialisasi ViewModel

Selama perubahan konfigurasi seperti rotasi layar, pengontrol UI seperti fragmen dibuat ulang. Namun, instance **ViewModel** tetap bertahan. Jika Anda membuat instance **ViewModel** menggunakan kelas **ViewModel**, objek baru dibuat setiap kali fragmen dibuat ulang. Sebagai gantinya, buat instance **ViewModel** menggunakan **ViewModelProvider**.



Penting: Selalu gunakan **ViewModelProvider** untuk membuat objek **ViewModel** daripada langsung membuat instance ViewModel.

Cara kerja ViewModelProvider:

- **ViewModelProvider** mengembalikan **ViewModel** yang ada jika ada, atau membuat yang baru jika belum ada.
- **ViewModelProvider** membuat instance **ViewModel** terkait dengan cakupan yang diberikan (aktivitas atau fragmen).

- **ViewModel** yang dibuat dipertahankan selama cakupannya hidup. Misalnya, jika cakupannya adalah fragmen, ViewModel akan dipertahankan hingga fragmen dilepaskan.

Inisialisasi **ViewModel**, menggunakan metode **ViewModelProvider.get()** untuk membuat **ViewModelProvider**:

1. Di kelas **GameFragment**, inisialisasi variabel viewModel. Letakkan kode ini di dalam **onCreateView()**, setelah definisi variabel binding. Gunakan metode **ViewModelProvider.get()**, dan teruskan dalam konteks **GameFragment** terkait dan kelas **GameViewModel**.
2. Di atas inisialisasi objek **ViewModel**, tambahkan pernyataan log ke log panggilan metode **ViewModelProvider.get()**.

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

3. Jalankan aplikasinya. Di Android Studio, buka panel **Logcat** dan filter di **Game**. Ketuk tombol **Play** di perangkat atau emulator Anda. Layar permainan terbuka.

Seperti yang ditunjukkan di Logcat, metode **onCreateView()** dari **GameFragment** memanggil metode **ViewModelProvider.get ()** untuk membuat **GameViewModel**. Pernyataan logging yang Anda tambahkan ke GameFragment dan GameViewModel muncul di Logcat.

```
I/GameFragment: Called ViewModelProvider.get
I/GameViewModel: GameViewModel created!
```

4. Aktifkan pengaturan putar otomatis pada perangkat atau emulator Anda dan ubah orientasi layar beberapa kali. **GameFragment** dimusnahkan dan dibuat ulang setiap kali, jadi **ViewModelProvider.get()** dipanggil setiap kali. Tapi **GameViewModel** dibuat hanya sekali, dan tidak dibuat ulang atau dimusnahkan untuk setiap panggilan.

```
I/GameFragment: Called ViewModelProvider.get
I/GameViewModel: GameViewModel created!
I/GameFragment: Called ViewModelProvider.get
I/GameFragment: Called ViewModelProvider.get
I/GameFragment: Called ViewModelProvider.get
```

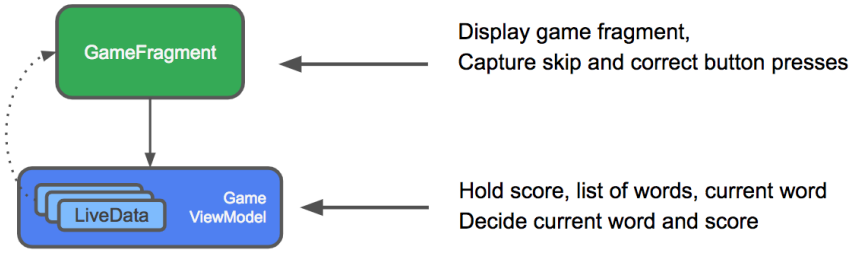
5. Keluar dari game atau keluar dari fragmen game. **GameFragment** dihancurkan. **GameViewModel** terkait juga dimusnahkan, dan callback **onCleared()** dipanggil.

```
I/GameFragment: Called ViewModelProvider.get
I/GameViewModel: GameViewModel created!
I/GameFragment: Called ViewModelProvider.get
I/GameFragment: Called ViewModelProvider.get
I/GameFragment: Called ViewModelProvider.get
I/GameViewModel: GameViewModel destroyed!
```

6. Tugas: Mengisi GameViewModel

ViewModel bertahan dari perubahan konfigurasi, jadi ini adalah tempat yang baik untuk data yang perlu bertahan dari perubahan konfigurasi:

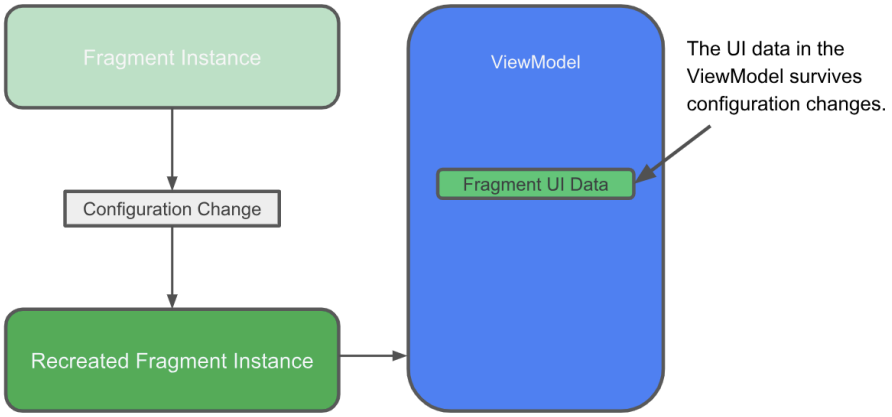
- Letakkan data yang akan ditampilkan di layar, dan kode untuk memproses data tersebut, di **ViewModel**.
- **ViewModel** tidak boleh berisi referensi ke fragmen, aktivitas, atau tampilan, karena aktivitas, fragmen, dan tampilan tidak bertahan dari perubahan konfigurasi.



Sebagai perbandingan, berikut ini cara penanganan data UI **GameFragment** di aplikasi pemula sebelum Anda menambahkan **ViewModel**, dan setelah Anda menambahkan **ViewModel**:

- Sebelum Anda menambahkan **ViewModel**: Saat aplikasi mengalami perubahan konfigurasi seperti rotasi layar, fragmen game dihancurkan dan dibuat ulang. Datanya hilang.
- Setelah Anda menambahkan **ViewModel** dan memindahkan data UI fragmen game ke dalam **ViewModel**: Semua data yang dibutuhkan fragmen untuk ditampilkan sekarang menjadi **ViewModel**. Saat aplikasi

mengalami perubahan konfigurasi, **ViewModel** bertahan, dan datanya dipertahankan.



Dalam tugas ini, Anda memindahkan data UI aplikasi ke kelas **GameViewModel**, bersama dengan metode untuk memproses data. Anda melakukan ini agar data disimpan selama perubahan konfigurasi.

Langkah 1: Pindahkan field data dan pemrosesan data ke ViewModel

Pindahkan field data dan metode berikut dari **GameFragment** ke **GameViewModel**:

1. Pindahkan field data **word**, **score**, dan **wordList**. Pastikan **word** dan **score** tidak bersifat **private**.

Jangan pindahkan variabel binding, **GameFragmentBinding**, karena berisi referensi ke tampilan. Variabel ini digunakan untuk memekarkan tata letak, menyiapkan pendengar klik, dan menampilkan data di layar — tanggung jawab fragmen.

2. Pindahkan metode **resetList()** dan **nextWord()**. Metode ini memutuskan kata apa yang akan ditampilkan di layar.
3. Dari dalam metode **onCreateView()**, pindahkan panggilan metode ke **resetList()** dan **nextWord()** ke blok **init GameViewModel**.

Metode ini harus ada di blok **init**, karena Anda harus menyetel ulang daftar kata saat **ViewModel** dibuat, bukan setiap kali fragmen dibuat. Anda dapat menghapus pernyataan **log** di blok **init** di **GameViewModel**.

Penangan klik **onSkip()** dan **onCorrect()** di **GameFragment** berisi kode untuk memproses data dan memperbarui UI. Kode untuk memperbarui UI harus tetap dalam fragmen, tetapi kode untuk memproses data perlu dipindahkan ke **ViewModel**.

Untuk saat ini, letakkan metode yang identik di kedua tempat:

1. Salin metode **onSkip()** dan **onCorrect()** dari **GameFragment** ke **GameViewModel**.
2. Di **GameViewModel**, pastikan metode **onSkip()** dan **onCorrect()** tidak bersifat **private**, karena Anda akan mereferensikan metode ini dari fragmen.

Berikut adalah kode untuk kelas **GameViewModel**, setelah melakukan pemfaktoran ulang:

```
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() {
        score--
        nextWord()
    }

    fun onCorrect() {
        score++
        nextWord()
    }

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

Berikut adalah kode untuk kelas **GameFragment**, setelah pemfaktoran ulang:

```
/**
* 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 ViewModelProvider.get")
       viewModel = ViewModelProvider(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() {
       score--
       nextWord()
   }

   private fun onCorrect() {
       score++
       nextWord()
```

```
    }


    /** Methods for updating the UI **/

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

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

Langkah 2: Perbarui referensi ke penangan klik dan field data di GameFragment

1. Di GameFragment, perbarui metode **onSkip()** dan **onCorrect()**. Hapus kode untuk memperbarui skor dan sebagai gantinya panggil metode **onSkip()** dan **onCorrect()** yang sesuai di **viewModel**.
2. Karena Anda memindahkan metode **nextWord()** ke **ViewModel**, fragmen game tidak bisa lagi mengaksesnya.

Di **GameFragment**, dalam metode **onSkip()** dan **onCorrect()**, ganti panggilan ke **nextWord()** dengan **updateScoreText()** dan **updateWordText()**. Metode ini menampilkan data di layar.

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

3. Di **GameFragment**, perbarui variabel **score** dan **word** untuk menggunakan variabel **GameViewModel**, karena variabel ini sekarang ada di **GameViewModel**.

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

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

Pengingat: Karena aktivitas, fragmen, dan tampilan aplikasi tidak bertahan dari perubahan konfigurasi, ViewModel tidak boleh berisi referensi ke aktivitas, fragmen, atau tampilan aplikasi.

4. Dalam **GameViewModel**, di dalam metode **nextWord()**, hapus panggilan ke metode **updateWordText()** dan **updateScoreText()**. Metode ini sekarang dipanggil dari **GameFragment**.
5. Bangun aplikasi dan pastikan tidak ada kesalahan. Jika Anda mengalami kesalahan, bersihkan dan bangun kembali proyek tersebut.
6. Jalankan aplikasi dan mainkan game melalui beberapa kata. Saat Anda berada di layar game, putar perangkat. Perhatikan bahwa skor saat ini dan kata saat ini dipertahankan setelah orientasi berubah.

Kerja bagus! Sekarang semua data aplikasi Anda disimpan dalam **ViewModel**, sehingga disimpan selama perubahan konfigurasi.

7. Tugas: Menerapkan click listener untuk tombol End Game

Dalam tugas ini, Anda menerapkan pemroses klik untuk tombol **End Game**.

1. Di **GameFragment**, tambahkan metode yang disebut **onEndGame()**. Metode **onEndGame()** akan dipanggil saat pengguna mengetuk tombol **End Game**.

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

2. Di **GameFragment**, di dalam metode **onCreateView()**, temukan kode yang menyetel listener klik untuk tombol **Got It** dan **Skip**. Tepat di bawah dua baris ini, setel pendengar klik untuk tombol **End Game**. Gunakan variabel **binding**. Di dalam pemroses klik, panggil metode **onEndGame()**.

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

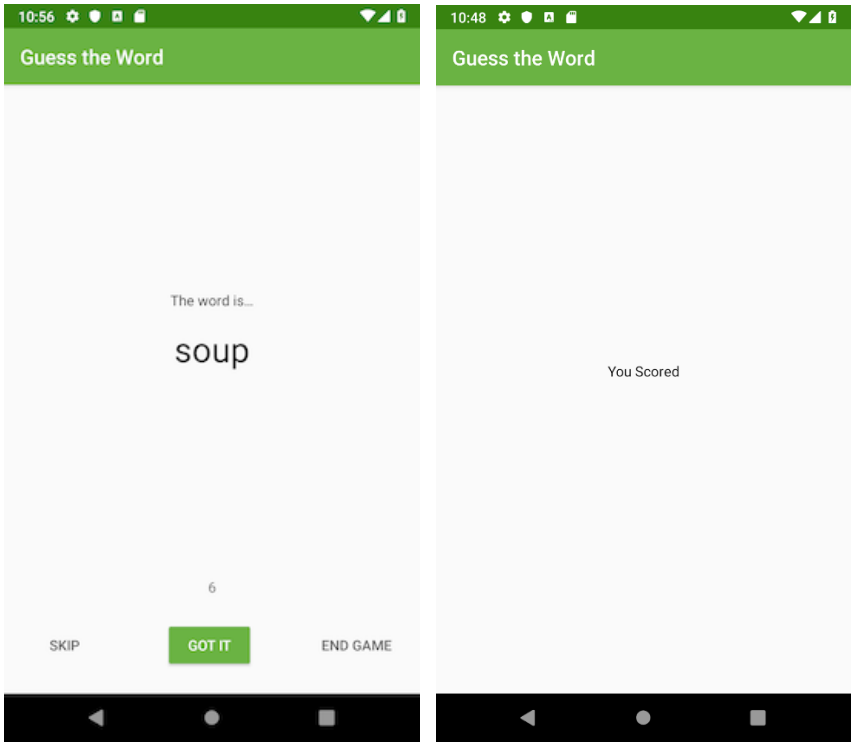
3. Di **GameFragment**, tambahkan metode yang disebut **gameFinished()** untuk menavigasi aplikasi ke layar skor. Berikan skor sebagai argumen, menggunakan **Safe Args**.

```
/**
* 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. Dalam metode **onEndGame()**, panggil metode **gameFinished ()**.

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

5. Jalankan aplikasi, mainkan game, dan putar beberapa kata. Ketuk tombol **End Game**. Perhatikan bahwa aplikasi menavigasi ke layar skor, tetapi skor akhir tidak ditampilkan. Anda memperbaikinya di tugas berikutnya.



8. Tugas: Menggunakan ViewModelFactory

Saat pengguna mengakhiri permainan, **ScoreFragment** tidak menampilkan skor. Anda ingin **ViewModel** yang memegang skor akan ditampilkan oleh **ScoreFragment**. Anda akan meneruskan nilai skor selama inisialisasi **ViewModel** menggunakan pola metode pabrik (*factory method pattern*).

Pola metode pabrik adalah pola desain kreasi yang menggunakan metode pabrik untuk membuat objek. Metode pabrik adalah metode yang mengembalikan instance dari kelas yang sama.

Dalam tugas ini, Anda membuat **ViewModel** dengan konstruktor berparameter untuk fragmen skor dan metode pabrik untuk membuat instance **ViewModel**.

1. Di bawah paket **score**, buat kelas Kotlin baru yang disebut **ScoreViewModel**. Kelas ini akan menjadi **ViewModel** untuk fragmen score.
2. Perluas kelas **ScoreViewModel** dari **ViewModel**. Tambahkan parameter konstruktor untuk skor akhir. Tambahkan blok **init** dengan pernyataan log.
3. Di kelas **ScoreViewModel**, tambahkan variabel yang disebut **score** untuk menyimpan skor akhir.

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

4. Di bawah paket **score**, buat kelas Kotlin yang dinamai **ScoreViewModelFactory**. Kelas ini akan bertanggungjawab untuk instantiasi objek **ScoreViewModel**.
5. Perluas kelas **ScoreViewModelFactory** dari **ViewModelProvider.Factory**. Tambahkan parameter konstruktor untuk skor akhir.

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

6. Dalam **ScoreViewModelFactory**, Android Studio menunjukkan error tentang unimplemented abstract member. Untuk menyelesaikan error, override metode **create()**. Dalam metode **create()**, kembailkan objek **ScoreViewModel** yang baru diciptakan.

```
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. Dalam **ScoreFragment**, ciptakan variabel kelas untuk **ScoreViewModel** dan **ScoreViewModelFactory**.

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

8. Dalam **ScoreFragment**, didalam **onCreateView()**, setelah inisialisasi variabel **binding**, inisialisasi **viewModelFactory**. Gunakan **ScoreViewModelFactory**. Lewatkan dalam skor akhir dari bundle argument, sebagaimana parameter konstruktor ke **ScoreViewModelFactory()**.

```
viewModelFactory =
ScoreViewModelFactory(ScoreFragmentArgs.fromBundle(requireArguments()).score)
```

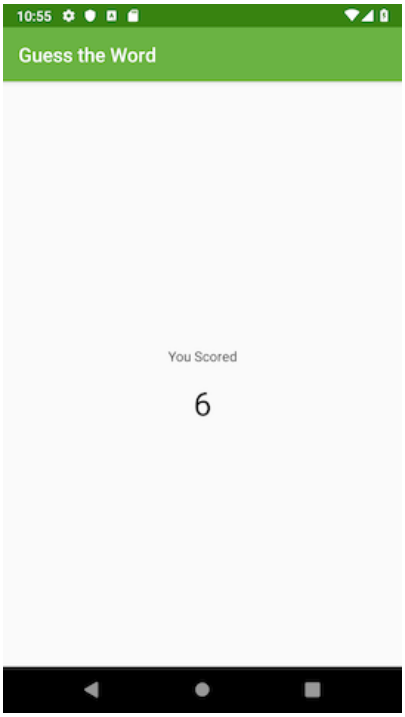
9. Di **onCreateView()**, setelah menginisialisasi **viewModelFactory**, inisialisasi objek **viewModel**. Panggil metode **ViewModelProvider.get()**, teruskan konteks fragmen skor terkait dan **viewModelFactory**. Ini akan membuat objek **ScoreViewModel** menggunakan metode pabrik yang ditentukan di kelas **viewModelFactory**.

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

10. Dalam metode **onCreateView()**, setelah menginisialisasi **viewModel**, setel teks tampilan **scoreText** ke skor akhir yang ditentukan dalam **ScoreViewModel**.

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

11. Jalankan aplikasi Anda dan mainkan gamenya. Gilir beberapa atau semua kata dan ketuk **End Game**. Perhatikan bahwa fragmen skor sekarang menampilkan skor akhir.



12. Opsional: Periksa log **ScoreViewModel** di Logcat dengan memfilter di **ScoreViewModel**. Nilai skor harus ditampilkan.

```
2019-02-07 10:50:18.328 com.example.android.guesstheword I/ScoreViewModel:
Final score is 15
```

LATIHAN

TUGAS

1. Buat aplikasi baru dengan menerapkan ViewModel dan ViewModelProvider dengan click listener

REFERENSI

1. https://developer.android.com/codelabs/kotlin-android-training-view-model#0
2. https://developer.android.com/jetpack/guide