Tema 3.4 - ViewModel

Descargar estos apuntes pdf o html

Índice

Introducción

En anteriores temas cuando hablamos de la **arquitectura de aplicaciones de Android** ya mencionamos este componente como parte de la Capa de UI. En este tema vamos a centrarnos en él para concretar la implementación de MVVM en nuestra capa UI de la arquitectura.

`ViewModel` es una de esas clases que **Google** definió, allá por 2018, en la primera versión de **Jetpack** para ayudar a los desarrolladores a crear aplicaciones de Android más robustas y fáciles de mantener. `ViewModel` es una clase que está **diseñada para almacenar y administrar datos relacionados con la interfaz de usuario de una manera que sobrevive a los cambios de configuración**, como la rotación de la pantalla.

Los beneficios clave de la clase ViewModel son básicamente dos:

- Te permite conservar el estado de la IU.
- Proporciona acceso a la lógica empresarial, idealmente a través de casos de uso definidos en el dominio.

Alcance de un ViewModel (ViewModel Scope)

Cuando se crea una instancia de `ViewModel`, se pasa un objeto que implementa la interfaz `ViewModelStoreOwner`.

Los objetos que implementan `ViewModelStoreOwner` puede ser por ejemplo:

- Una `Activity` o `ComponentActivity`.
- Un `Fragment`.
- Un destino de Navigation `NavBackStackEntry`.

> **Importante**
>
> El alcance de tu ViewModel se define en el Ciclo de vida del `ViewModelStoreOwner`. Esto es, continúa en la memoria hasta que su `ViewModelStoreOwner` desaparece de forma permanente.

Cuando se destruye el fragmento o la actividad para los que se definió el alcance del ViewModel, el trabajo asíncrono continúa en el ViewModel específico. Esta es la clave de la persistencia.

Cuando se definió la clase `ViewModel` en la primera versión de **Jetpack**, aún no existia **Compose** y se usaba asociado a una vista como un `Activity` o a un `Fragment`. Nuestras aplicaciones de Android se componían de una o más `Activity` o `Fragment` y cada uno de ellos tenía su propio `ViewModel` que se creaba como una **instancia única**. `ViewModel` se usaba para almacenar datos que se necesitaban en la interfaz de usuario de la `Activity/Fragment` o el `Fragment` y queríamos que sobrevivieran a los cambios de configuración o queríamos compartir datos entre ellos.

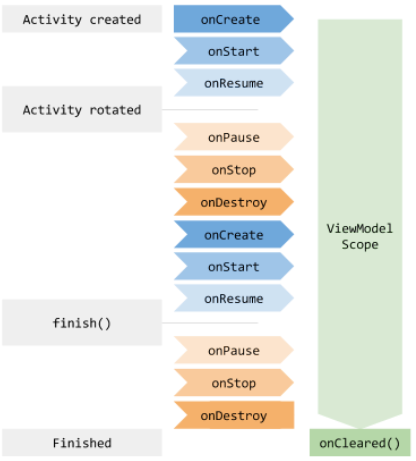
Con la llegada de **Jetpack Compose** este enfoque ha cambiado y Google ahora **recomienda aplicaciones de actividad única** donde las diferentes pantallas se cargan como contenido dentro de la misma actividad. Por tanto, un ViewModel utilizado por una actividad permanece en memoria hasta que la actividad finalice esto es hasta que la aplicación finalice.

Ciclo de vida de un ViewModel

- En el caso de una `Activity`, hasta que termina. (**El más común**)
- ~~En el caso de un `Fragment`, cuando se desvincula.~~ (No se usa en Compose)
- En el caso de un `NavBackStackEntry`, hasta volvemos atrás en el grafo.

En la imagen de la derecha vemos que el objeto `ViewModel` cuyo '***propietario***' (el `ViewModelStoreOwner`) es una `Activity` que tiene un cambio de configuración (rotación de pantalla) y por tanto se destruye y vuelve a crearse su vista asociada.

1. Se crea al llamarse al método `onCreate()` de la `Activity` **solo la primera vez** y aunque se vuelva a llamar a `onCreate()` por un cambio de configuración, **no se vuelve a crear** permanece el mismo objeto `ViewModel` que se creó en la primera llamada.
2. **No se destruye** aunque la `Activity` destruya su vista y permanece en memoria hasta la finalización de la `Activity`, en ese momento se llama al método `onCleared()` del `ViewModel` para que realice las tareas de limpieza necesarias.



Instanciando un ViewModel

Hay muchas formas de instanciarlo y eso nos puede llevar a confusión. Pero nosotros vamos a usar la más sencilla y recomendada por Google. De todas formas, si quieres profundizar puedes ver el siguiente ***'cheatsheet'*** creado por los desarrolladores de Google.

1. Para definirlo deberemos crear una clase que herede de `ViewModel` y que implemente la lógica de negocio que necesitemos.

   ```
   class MiViewModel : ViewModel() { ... }
   ```

2. ❌ **No debemos hacer jamás una instancia direcra**.

   ```
   val miVm = MiViewModel() //💀💀
   ```

   Ya que el ciclo de vida de un ViewModel está asociado a un `ViewModelStoreOwner` y por tanto es la clase `ViewModelProvider` la que se encarga de crearlo y mantenerlo en memoria.

   ```
   // activity es el objeto que implementa ViewModelStoreOwner
   val miVm = ViewModelProvider(activity).get(MiViewModel::class.java)
   ```

   El código anterior, nosotros **no tendremos que crearlo así nunca**, pues **Jetpack** nos proporciona formas más sencillas de hacerlo.

3. ⚠️ Como en el fondo debe ser `ViewModelProvider` quien lo crea, **si pasamos parámetros al constructor** de la clase `MiViewModel` no nos va a dejar, más adelante tendremos formas más sencillas de hacerlo. Por tanto, de momento, no deberíamos pasar parámetros al constructor.

   ```
   class MiViewModel(val param: String) : ViewModel() { ... }  //💀💀
   ```

   Si necesitáramos pasar parámetros al constructor, deberemos crear una clase que implemente `ViewModelProvider.Factory` y que se encargue de crear el objeto `MiViewModel` como se indica en la **documentación oficial**.

   > **Importante**
   >
   > Ya veremos más adelante que la librería **Hilt** (Jetpack) para inyección de dependencias me facilita mucho esta tarea. Así pues, no tendremos nunca que crear una clase que implemente `ViewModelProvider.Factory` cuando pasemos parámetros.

4. Delegado de creación `by viewModels()`, si el propietario es una `Activity`.

```
class MainActivity : ComponentActivity() {
    // Opción 1
    // Lo defino como propiedad de la clase y delego su creación que será al ser usa
    // primera vez después de llamarse el método onCreate() de la Activity y puede s
    // accedido desde cualquier método de la Activity que usemos después del onCreat
    // El delegado internamente llama a ViewModelProvider
    val miVm: MiViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Opción 2
        // Lo defino en el método onCreate() y lo clausuramos al definir el árbol de
        // seguirá siendo destruido por ViewModelProvider al destruirse la
        // Activity que es el ViewModelStoreOwner
        val miVm: MiViewModel by viewModels()

        setContent {
            // Clausura de miVm
        }
    }
}
```

5. Creación con `viewModel()`, si estamos dentro de una función `@Composable`
   El `ViewModelStoreOwner` será la `Activity` que renderiza la composición. No importa que `viewModel()` sea llamado más veces a lo largo de las recomposiciones pues solo se instanciará en la primera llamada.

```
@Composable
fun MiScreen(miVm: MiViewModel = viewModel()) { ... }
```

   Esta última forma, tal y como indica el `cheatSheet` de Google, necesita de la siguiente dependencia:

```
dependencies {
    // androidx.lifecycle:lifecycle-viewmodel-compose:<version>
    implementation(libs.androidx.lifecycle.viewmodel.compose)
}
```

   por lo que en el `libs.versions.toml` deberemos añadir la entrada correspondiente.

```
[libraries]
androidx-lifecycle-viewmodel-compose = {
    group = "androidx.lifecycle",
    name = "lifecycle-viewmodel-compose",
    version.ref = "lifecycleRuntimeKtx"
}
```

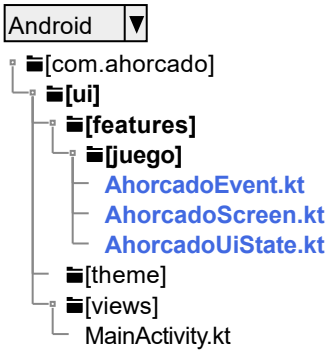
Nota

Existen más formas de crear un `ViewModel` que veremos más adelante.

Ejemplo de uso y definición de un ViewModel

Vamos a crear un simple juego del ahorcado para ver cómo se implementa un `ViewModel`. Para ello vamos a partir de un proyecto donde ya hemos diseñado una UI con **Jetpack Compose**.

- Puedes descargar el proyecto de partida de este enlace **Ahorcado.zip**.



Vamos a explicar la implementación del interfaz de usuario antes de ver la implementación del `ViewModel`. Para ello, partimos de la estructura del proyecto que se muestra en la imagen de la derecha donde tenemos un `AhorcadoScreen.kt`, un `AhorcadoUiState.kt` que guardará el '*State*' del UI y un `AhorcadoEvent.kt` que gestionará los eventos que se produzcan en el UI.

AhorcadoUiState.kt

Como hemos comentado guardará el '*State*' del UI definido en `AhorcadoScreen.kt` y que básicamente será el estado del conocido juego del ahorcado.

```
data class AhorcadoUiState(
    // Última letra introducida por el usuario, es una cadena de una sola letra.
    val letraIntroducida: String = "",
    // Palabra que se tiene que adivinar, procederá de alguna fuente de datos.
    val palabra: String = "",
    // Cadena que lleva las letras que se han fallado en el juego.
    val letrasFalladas: String = "",
    // Cadena que lleva las letras que se han acertado de la palabra a adivinar.
    val letrasAcertadas: String = "",
    // Valor enumerado que guarda en que momento del juego estoy.
    val estado: EstadoJuego = EstadoJuego.EMPEZAR
) {
    enum class EstadoJuego {
        EMPEZAR, // El juego aún no ha empezado y no se ha generado la palabra a adivinar.
        JUGANDO, // Estamos ya jugando y ya hay una palabra a adivinar.
        GANADO,  // El juego ha terminado ya y se ha ganado o se ha perdido. El usuario debe
        PERDIDO  // el resultado y debe aceptarlo para poder volver al estado EMPEZAR.
    }
}
```

AhorcadoEvent.kt

Lo eventos posibles que vamos a manejar en el UI y que se producirán al interactuar con el mismo los agrupamos en el siguiente `sealed interface`.

```
sealed interface AhorcadoEvent {
    // Estamos en estado EMPEZAR y pasamos a JUGANDO.
    object OnEmpezarJuego : AhorcadoEvent
    // Ha finalizado el juego, estamos en estado GANADO y PERDIDO y pasamos al estado EMPEZAR
    object OnAceptarResultadoFinalJuego : AhorcadoEvent
    // Estamos JUGANDO hay una **`letraIntroducida`** y la enviamos para que se compruebe.
    object OnLetraEnviada : AhorcadoEvent
    // El usuario introduce una letra y tenemos que comprobarla para ver si pertenece al
    // alfabeto castellano (sin tildes) y no está entre las acertadas y falladas.
    // De esta manera después podremos jugar con ella.s
    data class OnCambiaLetraIntroducida(val letra: String) : AhorcadoEvent
}
```

AhorcadoScreen.kt

Definimos el UI usando `Jetpack Compose` y aplicando '*State Hoisting* ' con funciones *composables* sin estado o '*stateless*'. Es la parte más extensa y vamos a describir los **componentes más importantes**.

```
// Recibe el número de fallos y los representa con una
// de la imágenes del ahorcado definidas en los recursos.
@Composable
fun MuestraAhorcado(modifier: Modifier = Modifier, fallos: Int) {
    val idRecurso = when (fallos) {
        0 -> R.drawable.ahorcado0   // No hay fallos
        1 -> R.drawable.ahorcado1   // 1 fallo
        2 -> R.drawable.ahorcado2   // 2 fallos
        3 -> R.drawable.ahorcado3   // ...
        4 -> R.drawable.ahorcado4
        5 -> R.drawable.ahorcado5
        6 -> R.drawable.ahorcado6
        else -> throw Exception("Número de fallos incorrecto")
    }
    Image(
        modifier = modifier.then(Modifier.fillMaxSize()),
        painter = painterResource(idRecurso),
        contentDescription = "Dibujo del ahorcado",
        colorFilter = ColorFilter.tint(MaterialTheme.colorScheme.primary)
    )
}
```

```
// Recibe la palabra a adivinar y las letras acertadas y muestra las letras acertadas y
// las que faltan por acertar representadas por '_'. Tenemos pues un Row con Text para cada l
@Composable
fun Palabra(palabra: String, letrasAcertadas: String) = Row(
    modifier = Modifier.fillMaxWidth(),
    horizontalArrangement = Arrangement.Center
) { ... }

// Muestras las letras falladas en un Row con Text para cada letra.
@Composable
fun Fallos(letrasFalladas: String) = Row(
    modifier = Modifier.fillMaxWidth(),
    horizontalArrangement = Arrangement.Start
) { ... }

// Muestra el estado del juego mientras estoy JUGANDO, esto es,
// la palabra parcialmente adivinada y las letras falladas.
@Composable
fun EstadoJuego(
    palabra: String, letrasAcertadas: String, letrasFalladas: String
) {
    Column(
        modifier = Modifier.fillMaxWidth(),
        verticalArrangement = Arrangement.Top,
    ) {
        Spacer(modifier = Modifier.size(20.dp))
        Palabra(palabra, letrasAcertadas)
        Spacer(modifier = Modifier.size(20.dp))
        Fallos(letrasFalladas)
    }
}
```

```
// TextField encargado de introducir una letra. Su estado está representado por letraState.
// Si hay letra parra jugar se muestra un Button para enviarla.
// Por esa razón elevamos dos eventos como la introducción de una letra y el envío de la mism
// Esto solo se podrá hacer si hay una letra válida para jugar.
@Composable
fun JuegaLetra(
    letraState: String, onCambiaLetraIntroducida: (String) -> Unit, onLetraEnviada: () -> Uni
) {
    Row(
        modifier = Modifier.fillMaxWidth(),
        verticalAlignment = Alignment.CenterVertically
    ) {
        // Hace que el foco recaiga sobre el OutlinedTextField
        val focusRequester = remember { FocusRequester() }
        OutlinedTextField(
            modifier = Modifier.width(100.dp)
                               .padding(start = 5.dp, end = 5.dp)
                               .focusRequester(focusRequester),
            value = letraState,
            singleLine = true,
            onValueChange = { onCambiaLetraIntroducida(it.uppercase()) },
            label = { Text(text = "Letra") },
        )
        // Hace que el foco recaiga sobre el OutlinedTextField
        LaunchedEffect(Unit) { focusRequester.requestFocus() }
        // Si hay letra para jugar se muestra el Button para enviarla.
        if (letraState.isNotEmpty()) {
            val controller = LocalSoftwareKeyboardController.current
            Button(
                modifier = Modifier.padding(start = 5.dp, end = 5.dp),
                onClick = {
                    // Ocultamos el teclado si está visible antes de enviar la letra.
                    controller?.hide()
                    onLetraEnviada()
                }
            ) { Text(text = "Jugar") }
        }
    }
}

// Componente que se muestra si el juego ha terminado y el estado es GANADO o PERDIDO.
// Mostramos el aviso correspondiente y un Button para aceptar el resultado y volver al estad
// Podría ser una AlertDialog.
```

```
@Composable
fun ResultadoFinalJuego(
    modifier: Modifier = Modifier,
    estadoJuego: AhorcadoUiState.EstadoJuego,
    onAceptarResultadoFinalJuego: () -> Unit
) { }
```

```
// Gestión de los estados de juego y su composición de la UI.
@Composable
fun MuestraInterface(
    modifier: Modifier = Modifier,
    ahorcadoState: AhorcadoUiState,
    onEventSent: (AhorcadoEvent) -> Unit
) {
    if (ahorcadoState.estado == AhorcadoUiState.EstadoJuego.EMPEZAR) {
        BotonEmpezar(
            modifier = modifier,
            onClick = { onEventSent(AhorcadoEvent.OnEmpezarJuego) }
        )
    } else {
        val scrollState: ScrollState = rememberScrollState()
        Column(
            verticalArrangement = Arrangement.Top,
            modifier = modifier.then(
                Modifier
                    .fillMaxWidth()
                    .verticalScroll(scrollState)
            )
        ) {
            EstadoJuego(
                palabra = ahorcadoState.palabra,
                letrasAcertadas = ahorcadoState.letrasAcertadas,
                letrasFalladas = ahorcadoState.letrasFalladas
            )
            Spacer(modifier = Modifier.size(20.dp))
            if (ahorcadoState.estado == AhorcadoUiState.EstadoJuego.JUGANDO) {
                JuegaLetra(
                    letraState = ahorcadoState.letraIntroducida,
                    onCambiaLetraIntroducida = { letra ->
                        onEventSent(AhorcadoEvent.OnCambiaLetraIntroducida(letra))
                    },
                    onLetraEnviada = { onEventSent(AhorcadoEvent.OnLetraEnviada) })
            } else {
                ResultadoFinalJuego(
                    estadoJuego = ahorcadoState.estado,
                    onAceptarResultadoFinalJuego = { onEventSent(AhorcadoEvent.OnAceptarResul
                )
            }
        }
```

```
    }
}
```

Layouts principales del juego y que gestionan la visualización de su estado en función de la orientación del dispositivo. Vamos a permitir el cambio de orientación para probar qué sucede con el estado del juego guardado en `ArhorcadoUiState`.

```
@Composable
fun AhorcadoPortrait(ahorcadoState: AhorcadoUiState, onEventSent: (AhorcadoEvent) -> Unit) {

@Composable
fun AhorcadoLandsacape(ahorcadoState: AhorcadoUiState, onEventSent: (AhorcadoEvent) -> Unit)

@Composable
fun AhorcaDoScreen(
    ahorcadoState: AhorcadoUiState,
    onEventSent: (AhorcadoEvent) -> Unit
) {
    // Este es el layout principal que permite accder al tamaño del ViewPort de la aplicació
    // maxWidth y maxHeight. En función de estos valores se decide como se distribuyen los co
    BoxWithConstraints(
        modifier = Modifier
            .fillMaxSize()
            .background(MaterialTheme.colorScheme.surface)
    ) {
        // Query que decide cómo se pinta el UI en función del ancho del ViewPort.
        if (maxWidth < 600.dp) {
            AhorcadoPortrait(
                ahorcadoState = ahorcadoState,
                onEventSent = onEventSent
            )
        } else {
            AhorcadoLandsacape(
                ahorcadoState = ahorcadoState,
                onEventSent = onEventSent
            )
        }
    }
}
```

AhorcadoViewModel.kt

Esta definición irá dentro del paquete `com.ahorcado.ui.features.juego`. Aquí definiremos el estado del UI y la lógica de negocio del juego. Hasta aquí llegarán todos los eventos que se produzcan en la UI.

```
class AhorcadoViewModel : ViewModel() {

    // En esta sección definimos todas las propiedades y métodos de clase que necesitemos.
    companion object {
        private val MAXIMO_FALLOS = 6

        // Comprueba si ya hemos acertado todas las letras de la palabra buscada.
        fun palabraAcertada(palabra: String, letrasAcertadas: String): Boolean {
            var acertada = true
            for (letra in palabra) {
                if (!letrasAcertadas.contains(letra)) {
                    acertada = false
                    break
                }
            }
            return acertada
        }
    }

    // Definimos el estado del UI
    // No podemos usar el API remenber pues no estamos dentro de un @Composable y además
    // al estar un ViewModel permanace en memoria hasta que se destruya la Activity.
    var ahorcadoState by mutableStateOf(AhorcadoUiState())

    ...
```

Definiremos lo métodos que gestionan los cambios del estado de la UI

```
    private fun empiezaJuego() {
        // Podría ser una fuente de datos.
        val palabras = listOf("XUSA", "JUANJO", "PEPE", "COMPOSABLE")
        ahorcadoState = ahorcadoState.copy(
            // Tomamos una palabra aleatoria a adivinar.
            palabra = palabras[Random.nextInt(0, palabras.size)],
            // El juego pasa a estado JUGANDO.
            estado = AhorcadoUiState.EstadoJuego.JUGANDO
        )
    }
```

```
    private fun reiniciaJuego() {
        ahorcadoState = AhorcadoUiState()
    }

    // Método encargado de ver si la letra introducida en el TextField de la
    // UI es válida para jugar y si es así, guardar su estado para reflejarlo en el mismo.
    // Aunque esté relacionado con dicho elemento de la UI, fíjate que en ningún
    // indicarlo en los nombres, ya que el ViewModel debe permanecer 'agnóstico' de
    // momento debemos cómo es la implementación en la UI.
    private fun gestionaLetraIntroducida(letra: String) {

        // Si solo hay una letra, no está entre las acertadas
        // ni entre las falladas y está entre la A-Z incluida Ñ
        ahorcadoState = if (letra.length == 1
            && !ahorcadoState.letrasAcertadas.contains(letra[0])
            && !ahorcadoState.letrasFalladas.contains(letra[0])
            && Regex("^[A-ZÑ]$").matches(letra)
        )
            ahorcadoState.copy(letraIntroducida = letra)
        else
            ahorcadoState.copy(letraIntroducida = "")
    }

    // Método encargado de jugar la letra introducida
    private fun juegaLetraIntroducida() {
        if (ahorcadoState.palabra.contains(ahorcadoState.letraIntroducida[0])) {
            val letrasAcertadas = ahorcadoState.letrasAcertadas
                                  + ahorcadoState.letraIntroducida
            ahorcadoState = ahorcadoState.copy(
                letraIntroducida = "",
                letrasAcertadas = letrasAcertadas,
                estado = if (palabraAcertada(ahorcadoState.palabra, letrasAcertadas))
                    AhorcadoUiState.EstadoJuego.GANADO
                else
                    AhorcadoUiState.EstadoJuego.JUGANDO
            )
        } else {
            val letrasFalladas = ahorcadoState.letrasFalladas
                                 + ahorcadoState.letraIntroducida
            ahorcadoState = ahorcadoState.copy(
                letraIntroducida = "",
                letrasFalladas = letrasFalladas,
                estado = if (letrasFalladas.length == MAXIMO_FALLOS)
```

```
                    AhorcadoUiState.EstadoJuego.PERDIDO
                else
                    AhorcadoUiState.EstadoJuego.JUGANDO
            )
        }
    }
```

```
    // Método encargado de gestionar los eventos que se produzcan en la UI.
    fun onEventoAhorcado(evento : AhorcadoEvent) {
        when (evento) {
            is AhorcadoEvent.OnEmpezarJuego -> empiezaJuego()
            is AhorcadoEvent.OnAceptarResultadoFinalJuego -> reiniciaJuego()
            is AhorcadoEvent.OnCambiaLetraIntroducida -> gestionaLetraIntroducida(evento.letr
            is AhorcadoEvent.OnLetraEnviada -> juegaLetraIntroducida()
        }
    }
} // Fin de la clase AhorcadoViewModel
```

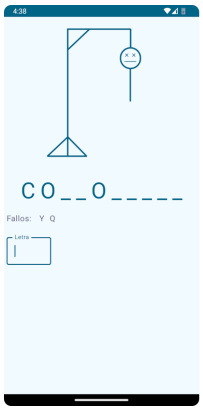
Asociando el ViewModel a la UI en MainActivity.kt

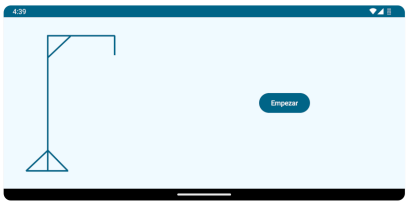
Vamos ahora a usar nuestro `ViewModel`...

1. ❌ Supongamos que cramos el objeto `ViewModel` sin más en el `onCreate()` sin usar ningún tipo de `ViewModelProvider`

```
class MainActivity : ComponentActivity() {
    lateinit var ahorcadoViewModel: AhorcadoViewModel
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        ahorcadoViewModel = AhorcadoViewModel() // 💀💀
        setContent {
            AhorcadoTheme {
                AhorcaDoScreen(
                    ahorcadoViewModel.ahorcadoState,
                    ahorcadoViewModel::onEventoAhorcado
                )
            }
        }
    }
}
```

   Si empezamos a jugar y giramos el movil se vuelve a llamar a `onCreate()` y el estado se pierde.

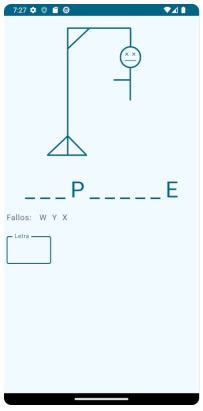


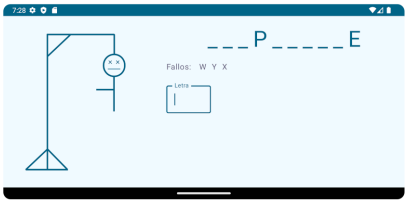


2. Lo hacemos correctamente u asignaremos el `ViewModel` a la `Activity` que es el `ViewModelStoreOwner` primero usando `VieModelProvider` con el delegado `viewModels()`

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
4       val ahorcadoViewModel: AhorcadoViewModel by viewModels() // ✅
        setContent {
            AhorcadoTheme {
                AhorcaDoScreen(
                    ahorcadoViewModel.ahorcadoState,
                    ahorcadoViewModel::onEventoAhorcado
                )
            }
        }
    }
}
```

Ahora si volvemos a jugar y giramos el movil el `ViewModel` permanece y el estado se redibuja.





3. Por último, vamos a probar a crear el `ViewModel` dentro de una función `@Composable` usando la función *composable* `viewModel()` que **requiere de las dependencias** que hemos comentado antes.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AhorcadoTheme {
6               val ahorcadoViewModel: AhorcadoViewModel = viewModel() // ✅
                AhorcaDoScreen(
                    ahorcadoViewModel.ahorcadoState,
                    ahorcadoViewModel::onEventoAhorcado
                )
            }
        }
    }
}
```

✋ Prácticas no recomendadas

Las siguientes son varias prácticas recomendadas clave que debes seguir cuando implementes `ViewModel`:

- ❌ **NO** definas `ViewModel` para *composables* reutilizables en tu UI o para componentes de IU que no sean de nivel superior. Deberíamos definirlos a nivel de Screen (pantalla).
- ❌ Los ViewModels **NO deberían conocer los detalles de implementación de la IU**. Mantén los nombres de los métodos que expone la API de ViewModel y los de los campos del UIState lo más genéricos posible.
- ❌ Como pueden tener una vida más larga que el `ViewModelStoreOwner`, los ViewModels **NO deberían contener ninguna referencia de APIs relacionadas con el ciclo de vida**, como `Context` o `Resources` para evitar fugas de memoria.
- ❌ **NO pases ViewModels a funciones ni otros componentes de la IU**. Esto evita que los componentes de nivel inferior accedan a más datos y lógica de los que necesitan.
- ❌ Derivado del anterior **NO instancies directamente un objeto ViewModels**. En su lugar, usa algún tipo de *'proveedor de ViewModels'* para obtener una instancia del mismo.