Capitulo 6

UD06. MAD (MODERN ANDROID DEVELOPMENT)

Resultados de avaliación

RA1. Aplica tecnoloxías de desenvolvemento para dispositivos móbiles, e avalía as súas características e as súas capacidades.

RA2. Desenvolve aplicacións para dispositivos móbiles, para o que analiza e emprega as tecnoloxías e as librarías específicas.

Criterios de avaliación

- CA1.8 Realizáronse modificacións sobre aplicacións existentes.
- CA1.9 Utilizáronse emuladores para comprobar o funcionamento das aplicacións.
- CA2.1 Xerouse a estrutura de clases necesaria para a aplicación.
- CA2.4 Utilizáronse as clases necesarias para a conexión e a comunicación con dispositivos sen fíos.
- CA2.5 Utilizáronse as clases necesarias para o intercambio de mensaxes de texto e multimedia.
- CA2.8 Realizáronse probas de interacción entre o usuario e a aplicación para mellorar as aplicacións desenvolvidas a partir de emuladores.
- CA2.9 Empaquetáronse e despregáronse as aplicacións desenvolvidas en dispositivos móbiles reais.
- CA2.10 Documentáronse os procesos necesarios para o desenvolvemento das aplicacións

BC1. Análise de tecnoloxías para desenvolvemento de aplicacións en dispositivos móbiles.

- Contidos
- Contornos integrados de traballo.
- Módulos para o desenvolvemento de aplicacións móbiles.
- Emuladores.
- Estrutura dunha aplicación para dispositivo móbil.
- Modificación de aplicacións existentes.
- Uso do contorno de execución do administrador de aplicacións.
- Ferramentas e fases de construción.
- Técnicas de animación e son.
- Comunicacións: clases asociadas. Tipos de conexións.
- Xestión da comunicación sen fíos.
- Envío e recepción de mensaxes de texto: seguridade e permisos.
- Envío e recepción de mensaxaría multimedia: sincronización de contido; seguridade e permisos.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE UD06. MAD (MODERN ANDROID DEVELOPMENT)

INTRODUCCIÓN

A lo largo de los años, las arquitecturas típicas de las aplicaciones Android han experimentado una evolución significativa. En el pasado, los modelos constructivos se basaban en patrones como MVC (Model-View-Controller) o MVP (Model-View-Presenter). Sin embargo, con la introducción de nuevas soluciones y tecnologías, los proyectos más recientes han adoptado patrones como MVVM (Model-View-ViewModel) o MVI (Model-View-Intent).

En este contexto de cambio, es fundamental explorar algunas de estas tecnologías incluidas en Jetpack, una colección de bibliotecas y herramientas recomendadas por Android:

- **View Binding:** Esta tecnología permite vincular los componentes de la interfaz de usuario con el código de manera más segura y eficiente, reemplazando la necesidad de usar `findViewById`.
- **View Models:** Los ViewModel son componentes que ayudan a gestionar y mantener datos relacionados con la interfaz de usuario, lo que facilita la gestión de ciclos de vida y la separación de preocupaciones.
- **Live Data:** LiveData es una clase diseñada para almacenar y observar datos de manera reactiva, lo que simplifica la actualización de la interfaz de usuario cuando cambian los datos subyacentes.
- **Data Binding, Room databases, …:** Estas tecnologías amplían aún más las capacidades de desarrollo, permitiendo la vinculación de datos directamente a la interfaz de usuario, gestionando bases de datos locales con facilidad y mejorando la calidad y la eficiencia del código en general.

La adopción de estas tecnologías y patrones arquitectónicos ha revolucionado la forma en que se desarrollan aplicaciones Android, mejorando la organización del código y la experiencia del usuario.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

MODEL-VIEW-VIEWMODEL

La arquitectura MVVM (Model-View-ViewModel) es un patrón de diseño ampliamente utilizado en el desarrollo de aplicaciones Android. Su objetivo principal es separar las preocupaciones y mejorar la organización del código en las aplicaciones, lo que facilita el mantenimiento y la escalabilidad. Aquí te explico cómo funciona:

1. **Model (Modelo):** El modelo representa los datos y la lógica de negocio de tu aplicación. Puede incluir clases que representen los objetos de datos, así como

componentes para acceder y administrar esos datos, como bases de datos, servicios web o repositorios. El modelo no tiene conocimiento de la interfaz de usuario.
2. **View (Vista):** La vista es la capa de la interfaz de usuario que se encarga de mostrar los datos y permitir la interacción del usuario. En Android, la vista suele estar representada por las actividades, fragmentos y diseños de interfaz de usuario.
3. **ViewModel:** El ViewModel actúa como intermediario entre el Modelo y la Vista. Contiene la lógica de presentación y se encarga de preparar y proporcionar los datos necesarios para que la vista los muestre. También maneja las interacciones del usuario y cualquier lógica relacionada con la vista, como validaciones y formateo de datos.

El flujo de datos en una arquitectura MVVM típica es el siguiente:

1. El **ViewModel** solicita datos al Modelo (por ejemplo, a través de un repositorio) y los almacena en propiedades observables.
2. La Vista se suscribe a estas propiedades observables del ViewModel.
3. Cuando los datos en el ViewModel cambian, se notifica a la Vista y se actualiza automáticamente la interfaz de usuario.
4. Cuando el usuario interactúa con la interfaz de usuario, la Vista envía eventos al ViewModel para que procese acciones, como guardar datos o realizar operaciones adicionales.

Una característica importante de MVVM es la capacidad de utilizar la vinculación de datos (data binding) para que los cambios en el ViewModel se reflejen automáticamente en la Vista, y viceversa. Esto simplifica la actualización de la interfaz de usuario y reduce la necesidad de código manual para sincronizar datos y vistas.

En Android, MVVM se implementa comúnmente con el uso de bibliotecas como LiveData para la comunicación entre el ViewModel y la Vista, y Data Binding para la vinculación de datos. Estas herramientas ayudan a simplificar la implementación de la arquitectura MVVM en aplicaciones Android, mejorando la organización y mantenimiento del código.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

VIEW BINDING

La implementación de View Binding en Android simplifica la escritura de código que interactúa con las vistas de la interfaz de usuario de una manera altamente eficiente y segura. Una vez habilitada, esta característica genera una clase de vinculación (binding class) para cada layout XML, estableciendo un mapeo directo con cada vista dentro del diseño. Esta funcionalidad se aplica tanto a las actividades como a los fragmentos en una aplicación.

Una de las ventajas más notables de View Binding es su capacidad para eliminar la necesidad de usar `findViewById`, lo que conlleva beneficios significativos:

- **Eficiencia:** Al evitar la búsqueda a través de la jerarquía de vistas en tiempo de ejecución, el código resulta más eficiente en términos de rendimiento, ya que Android no necesita realizar búsquedas innecesarias.
- **Seguridad:** A diferencia de `findViewById`, View Binding es type-safe. Elimina la necesidad de realizar casting y garantiza que las referencias sean seguras y coincidan

con las vistas correctas. Además, elimina el riesgo de referencias nulas devueltas por el método en caso de que el ID suministrado sea inválido, ya que los problemas se detectan en tiempo de compilación, lo que mejora la robustez del código.

Si deseas obtener más información sobre View Binding, puedes consultar la documentación oficial en este enlace.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE VIEW BINDING

DEPENDENCIA

Para habilitar View Binding en tu proyecto, es necesario realizar un ajuste en el archivo build.gradle de la aplicación de la siguiente manera:

```
android {
    ...
    buildFeatures {
        viewBinding true
    }
}
```

Una vez habilitado, View Binding generará **automáticamente** una clase de vinculación para cada actividad o fragmento. El nombre de esta clase se forma tomando el nombre de la actividad o fragmento al que está asociada, al cual se le agrega "**Binding**". Este objeto de vinculación se encarga de mapear el diseño de la interfaz de usuario.

Para acceder a este objeto de vinculación en tus componentes, puedes seguir estos pasos:

- En el caso de actividades, define una propiedad que se inicializará en el método `onCreate`.
- Para fragmentos, debido a que estos pueden persistir más allá de la vida de sus vistas, crea una propiedad que se inicialice en `onCreateView` y se anule en `onDestroyView`.

Esto proporciona una manera segura y eficiente de interactuar con los elementos de la interfaz de usuario en tus actividades y fragmentos.

Si deseas obtener información adicional sobre View Binding, puedes consultar la documentación oficial en este enlace.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

VIEW BINDING EN EL PROYECTO CHRONOS

Para actualizar la aplicación Chronos y utilizar View Binding, sigue estos pasos:

1. Activar Componente en el build.gradle de la aplicación:

Añade la siguiente configuración en el bloque `buildFeatures`:

```
android {
    ...
    buildFeatures {
        viewBinding = true
    }
}
```

Guarda el archivo y actualiza el proyecto.

2. Implementar View Binding en la Actividad Principal:

```
// Importa la clase generada para el View Binding
import com.example.yourpackage.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    // Variable para almacenar la referencia del View Binding
    private lateinit var binding: ActivityMainBinding

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

        // Inflar el diseño utilizando View Binding
        binding = ActivityMainBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)

        // Acceso a los elementos de la interfaz mediante el objeto binding
        val chrono = binding.chrTemporizador
        val btnStart = binding.btnStart
        val btnPause = binding.btnPause

        // Resto del código...
    }
}
```

Con estos cambios, ahora puedes acceder directamente a los elementos de la interfaz a través del objeto `binding`, eliminando la necesidad de usar `findViewById` y simplificando el código.

La implementación de View Binding hace que el código sea más claro y menos propenso a errores al proporcionar referencias seguras y directas a los elementos de la interfaz de usuario.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

VIEW BINDING EN EL PROYECTO PIZZA APP

En esta aplicación, aprenderemos a activar y utilizar un componente en un fragmento. Si recordamos, esta aplicación fue creada en base a un fragmento.

Activar Componente

Primero, activaremos el componente. Para ello, vamos al archivo `build.gradle` de la aplicación y añadimos:

```
buildFeatures {
    viewBinding = true
}
```

Con esto, hemos completado la activación. Ahora, solo nos falta actualizar el proyecto:

```
sync now
```

View Binding en Fragmento

En este caso, declararemos una variable `binding` para almacenar los recursos y será de la clase `FragmentOrderBinding`:

```
private lateinit var binding: FragmentOrderBinding
```

Sin embargo, con los fragmentos, surge un problema, ya que el ciclo de vida de un fragmento puede diferir de los elementos que tiene en su diseño. En este caso, los elementos pueden destruirse, pero el fragmento puede seguir existiendo, lo que impediría acceder a los mismos.

Para resolver esto, crearemos otra variable (que comenzará con `_` para diferenciarla) que aceptará nulos:

```
private var _binding: FragmentOrderBinding? = null
private val binding: FragmentOrderBinding
    get() = _binding!!
```

Ahora, indicaremos que si se destruye el fragmento, esa variable sea nula:

```
override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
```

Luego, recuperaremos la referencia al objeto y obtendremos una referencia al padre:

```
_binding = FragmentOrderBinding.inflate(inflater, container, false)
val view = binding.root
```

Con esto, podemos eliminar todas las referencias a los elementos utilizando la variable `binding`. Por ejemplo:

```
binding.chipParmesano.isChecked
```

VIEW MODEL

La clase ViewModel actúa como un contenedor que encapsula el estado y la lógica a nivel de pantalla en una aplicación. Su función principal es **exponer** el estado de la aplicación a la interfaz de usuario (UI) y **encapsular** la lógica relacionada.

Una de las principales **ventajas** de utilizar ViewModel radica en su capacidad para gestionar automáticamente la persistencia durante la navegación entre actividades y destinos de Navigation, así como en cambios de configuración, como rotaciones de pantalla.

Un ViewModel puede estar asociado a una actividad o fragmento, y su ciclo de vida está vinculado al ciclo de vida de ese componente. Además, permite que dos o más fragmentos compartan un mismo objeto ViewModel si está creado y asociado al contenedor principal (ya sea una actividad o un fragmento). Este mecanismo facilita la compartición de datos entre diferentes destinos en la aplicación.

Para generar un modelo, existen varias formas, siendo la más sencilla el uso de la función `viewModels()`. Esta función devuelve una propiedad delegada que proporciona acceso al modelo. Por defecto, su alcance será el del fragmento o actividad donde se declara. Por ejemplo:

```
class MyFragment : Fragment() {
    val viewModel: MyViewModel by viewModels()
}
```

Si se desea modificar el alcance, se puede hacer mediante el parámetro `ownerProducer`:

```
class MyFragment : Fragment() {
    val viewModel: MyViewModel by viewModels(
        ownerProducer = { this.requireActivity() }
    )
}
```

Esta flexibilidad en la creación y alcance de ViewModels proporciona una gestión eficiente del estado y la lógica de la aplicación en entornos Android. Para obtener más detalles, se puede consultar la documentación oficial en este enlace.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE VIEW MODEL

CREAR PROYECTO

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.

2. **Crear un Nuevo Proyecto**:
    - Selecciona “Start a new Android Studio project” en la pantalla de inicio.
    - Elige “Phone and Tablet” como tipo de dispositivo.
    - Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
    - En la siguiente pantalla, completa la información básica sobre el proyecto:
        - **Name**: Ingresa “UF1_UD06_3_GuessGame”.
        - **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
        - **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
        - **Language**: Selecciona “Kotlin” como lenguaje de programación.
        - **Minimum API level**: Selecciona “API 35: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
    - Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
    - Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad `MainActivity.kt` y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

DEPENDENCIA

Activar View Binding

Para habilitar View Binding en tu proyecto, realiza el siguiente ajuste en el archivo build.gradle de la aplicación:

```
android {
    . . .
    buildFeatures {
        viewBinding = true
    }
}
```

Asegúrate de sincronizar los cambios para aplicar la configuración.

Dependencia para Crear Fragmentos

Para la creación de fragmentos, añade la siguiente dependencia en tu archivo build.gradle:

```
implementation("androidx.navigation:navigation-fragment-ktx:2.7.5")
```

Después de agregar la dependencia, realiza la sincronización para incorporar los cambios en tu proyecto.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

GAME FRAGMENT

Vamos a crear un nuevo fragmento asociado a la pantalla de “Game”.

Layout del Fragmento

Actualizamos el diseño de nuestro fragmento. A continuación se muestra el nuevo XML del layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:id="@+id/txt_word"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:letterSpacing="0.2"
        android:textSize="35sp"/>

    <!-- Cuantas vidas nos quedan -->
    <TextView
        android:id="@+id/txt_lives"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:letterSpacing="0.2"
        android:textSize="16sp"/>

    <!-- Introducir letras -->
    <EditText
        android:id="@+id/txt_guess"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:inputType="text"
        android:maxLength="1"
        android:layout_marginTop="8dp"
        android:textSize="16sp"
        android:gravity="center"
        android:hint="Introduce una letra"/>

    <!-- Boton siguiente -->
    <Button
        android:id="@+id/button_next"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Prueba!"
        android:layout_gravity="center"
        android:layout_marginTop="8dp" />

</LinearLayout>
```

Código del Fragmento

Actualizamos el código del fragmento, eliminamos el código innecesario e incorporamos la propiedad binding:

```
class GameFragment : Fragment() {

    private var _binding: FragmentGameBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root
        return view
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

}
```

Con estos cambios, hemos definido las propiedades del `viewBinding` para acceder a las propiedades del layout. Además, se ha simplificado el código del fragmento eliminando las partes innecesarias.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

RESULT FRAGMENT

Vamos a crear un nuevo fragmento asociado a la pantalla de “Result”.

Layout del Fragmento

Actualizamos el diseño de nuestro nuevo fragmento. A continuación, se presenta el nuevo XML del layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:id="@+id/txt_result"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:gravity="center"
        android:textSize="28sp"/>

    <!-- Boton Reiniciar -->
    <Button
        android:id="@+id/button_new_game"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Nueva Partida"
        android:layout_gravity="center"
        android:layout_marginTop="8dp" />

</LinearLayout>
```

Código del Fragmento

Actualizamos el código del fragmento y eliminamos el código innecesario:

```
package com.example.ud06_3_guessgame

import android.os.Bundle
import androidx.fragment.app.Fragment
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import com.example.ud06_3_guessgame.databinding.FragmentResultBinding

class ResultFragment : Fragment() {

    private var _binding: FragmentResultBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = FragmentResultBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

Con estos cambios, hemos definido las propiedades del `viewBinding` para acceder a las propiedades del layout. Además, se ha simplificado el código del fragmento eliminando las partes innecesarias.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

GRAFO DE NAVEGACIÓN

Generaremos un grafo de navegación denominado “nav_graph.xml”, en el cual incorporaremos ambos fragmentos y estableceremos las rutas correspondientes:

Posteriormente, ajustaremos las propiedades de ambas rutas, especialmente la propiedad “PopUpTo”, para garantizar que no existan fragmentos previos al realizar la navegación:

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

MAINACTIVITY

Vamos a realizar ajustes en la actividad principal para incorporar un contenedor de fragmentos. A continuación, se presenta el nuevo diseño XML:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/fragment_container_view"
    tools:context=".MainActivity"
    android:name="androidx.navigation.fragment.NavHostFragment"
    app:navGraph="@navigation/nav_graph"
    app:defaultNavHost="true"
    >

</androidx.fragment.app.FragmentContainerView>
```

Con estos cambios, no será necesario realizar ninguna otra modificación en la actividad principal, ya que nuestro primer fragmento se cargará automáticamente.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

GAME MODEL

Para implementar un ViewModel, debemos crear una clase llamada `GameViewModel` que herede de la clase `ViewModel`:

```
package com.example.ud06_3_guessgame

import androidx.lifecycle.ViewModel

class GameViewModel : ViewModel() {
}
```

Para realizar una prueba, generaremos una lista de palabras y seleccionaremos una palabra al azar de la lista, transformándola a mayúsculas:

```
val words = listOf("Android", "Fragment", "Kotlin", "Model")
var secretWord = words.random().uppercase()
```

Desde el fragmento, incluiremos una referencia a esta nueva clase:

```
val model: GameViewModel by viewModels()
```

La vida de este modelo está vinculada a la vida del fragmento, lo que nos permite acceder a cualquier método o propiedad del modelo desde el propio fragmento:

```
binding.buttonNext.setOnClickListener {
    model.secretWord = "Prueba de modelo"
    view.findNavController().navigate(R.id.action_gameFragment_to_resultFragment)
}
```

Recuperaremos el modelo en el fragmento de Resultado y generaremos un Toast al hacer clic en el botón de nuevo juego:

```
binding.buttonNewGame.setOnClickListener {
    Toast.makeText(activity, model.secretWord, Toast.LENGTH_LONG).show()
}
```

En `GameFragment`, creamos un objeto a partir de nuestro modelo, modificamos el valor de una variable (`secretWord`) y navegamos al siguiente fragmento. Al avanzar al siguiente fragmento, mostramos con un Toast el valor de la variable.

Sin embargo, al ejecutar la aplicación, observamos que no se muestra la palabra que hemos definido en el fragmento.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

COMPARTIR MODELO

Cuando creamos nuestro modelo, este queda vinculado al fragmento o a la actividad donde se crea, es decir, su alcance está limitado a ese contexto específico. Debido a esto, al navegar entre fragmentos, la información no se comparte automáticamente. Para abordar este desafío y asegurar que el modelo esté vivo durante toda la navegación entre fragmentos, necesitamos asociar su ciclo de vida al contenedor padre común de **ambos fragmentos.**

En este caso, indicamos que el propietario del modelo será la actividad que contiene el fragmento:

```
val model: GameViewModel by viewModels(
    ownerProducer = { this.requireActivity() }s
)
```

Este enfoque se aplica en ambos fragmentos. Android verificará la existencia de esa actividad y, en caso de que no exista (como en la primera carga), creará el objeto. Sin embargo, al avanzar al siguiente fragmento y al intentar crear otro fragmento asociado a la misma actividad principal, Android detectará que el objeto ya existe. En consecuencia, almacenará la referencia al modelo existente en lugar de generar uno nuevo. Esto permite que los fragmentos subsiguientes accedan al modelo creado en el proceso anterior.

Esto resulta beneficioso para compartir información entre fragmentos, ya que la asociación ocurre antes de que se carguen los fragmentos. Además, podemos utilizar nuestras propias clases implementadas. Además, en situaciones como cambios de configuración (por ejemplo, rotación de la pantalla) donde la información podría perderse, no es necesario almacenarla en el bundle para recuperarla posteriormente.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

LÓGICA DEL JUEGO

En este momento vamos a implementar la lógica de nuestro juego. Para ello, nos dirigimos a la clase del **modelo** y creamos un método que permita especificar las palabras:

```
class GameViewModel: ViewModel() {

    // Seleccionamos una palabra aleatoria de esta lista de palabras
    val words = listOf("Android", "Fragment", "Kotlin", "Model")
    var secretWord  = words.random().uppercase()
    // String que se mostrará en la pantalla (guiones y letras a medida que las vamos descubriendo)
    var secretWordDisplay = ""
    // Intentos del usuario. Caracteres que vaya probando el usuario.
    var guesses = mutableListOf<Char>()
    // Vidas
    var lives = 8

    init {
        // Inicializamos la palabra con _
        secretWordDisplay = generateSecretWordDisplay()
    }

    // Genera la representación visual de la palabra oculta
    fun generateSecretWordDisplay() =
        // Recorremos cada uno de los caracteres de la palabra
        secretWord.map {
            // Si el caracter está en la lista, lo añadimos; sino, continuamos con _
            if (it in guesses) it
            else '_'
        }.joinToString("")

    // Realiza un intento de adivinanza por parte del usuario
```

```
    fun makeGuess(guess: String){
        if(guess.length > 0) {
            // Extraemos la letra inicial (aunque solo nos pueden introducir un caracter)
            val letter = guess.uppercase()[0]
            // La añadimos a la lista de intentos
            guesses.add(letter)

            secretWordDisplay = generateSecretWordDisplay()
            if(!secretWord.contains(letter)) lives -= 1
        }
    }

    // Función para verificar si ganamos
    fun win() = secretWord == secretWordDisplay
    // Función para comprobar si nos quedan vidas
    fun lost() = lives <= 0
}
```

Este modelo encapsula la lógica del juego y la representación visual de la palabra oculta, permitiendo que las interacciones del usuario se reflejen de manera coherente en la interfaz de usuario.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

USAR EL MODELO

En este punto vamos a utilizar el modelo en nuestra actividad prinicipal:

```
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        _binding = FragmentGameBinding.inflate(inflater, container, false)
        val view = binding.root

        //Inicializamos la palabra
        updateScreen()

        binding.buttonNext.setOnClickListener {
            //model.secretWord = "Prueba de modelo"
            if(binding.txtGuess.text.length>0){
                //Comprobar la letra introducida
                model.makeGuess(binding.txtGuess.text.toString())
                //Actualizamos la pantalla
                updateScreen()
                //Si acertamos la palabra o nos quedamos sin vidas
                if (model.win() || model.lost())
                    view.findNavController().navigate(R.id.action_gameFragment_to_resultFragment)
            }else{
                //Sino se introduce ningún texto mostramos un aviso
                Snackbar.make(view, "Introduce una letra", Snackbar.LENGTH_SHORT).show()
            }
        }
        return view
    }
```

```
    fun updateScreen(){
        binding.txtWord.text = model.secretWordDisplay
        binding.txtLives.text = "Te quedan ${model.lives} vidas"
        binding.txtGuess.text = null
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
```

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

LÓGICA RESULT FRAGMENT

Vamos a generar la lógica del fragmento del resultado. Para ello vamos a nuestro viewModel:

Funciones en el Modelo

```
class GameViewModel : ViewModel() {

    // ...

    fun restart() {
        guesses.clear()
        lives = 8
        secretWord = words.random().uppercase()
        secretWordDisplay = generateSecretWordDisplay()
    }

    fun resultMessage() =
        if (win()) "Ganaste!\n  La palabra secreta era $secretWord"
        else  "Oops, perdiste!\n  La palabra secreta era $secretWord"

    // ...
}
```

Lógica del Fragmento:

```
// ...

binding.txtResult.text = model.resultMessage()

binding.buttonNewGame.setOnClickListener {
    model.restart()
    view.findNavController().navigate(R.id.action_resultFragment_to_gameFragment)
    // Toast.makeText(activity, model.secretWord, Toast.LENGTH_LONG).show()
}

// ...
```

Estas funciones amplían la lógica del juego y proporcionan mensajes personalizados para mostrar al usuario después de completar o perder el juego.

1. **restart()**: Esta función reinicia el juego. Limpia la lista de intentos (`guesses`), restablece el número de vidas a 8, y selecciona una nueva palabra secreta aleatoria. Luego, genera la representación visual de la nueva palabra.
2. **resultMessage()**: Esta función devuelve un mensaje personalizado en función del resultado del juego. Si el jugador ha ganado, muestra un mensaje de victoria junto con la palabra secreta. Si el jugador ha perdido, muestra un mensaje de derrota con la palabra secreta.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

Capitulo 5

LIVE DATA

LiveData en Android: Mejora de Observabilidad

LiveData es una parte esencial de Jetpack que permite la declaración de datos observables, aplicando el patrón de diseño Observer. Este enfoque permite un seguimiento eficiente de las modificaciones en los datos, facilitando la actualización de la interfaz de usuario por parte de fragmentos y actividades.

Ventajas clave de LiveData:

1. **Actualización Automática de la UI:** LiveData permite que los fragmentos y actividades actualicen automáticamente su interfaz de usuario en respuesta a cambios en los datos. El observador se encarga de gestionar estas actualizaciones.
2. **Gestión del Ciclo de Vida:** LiveData gestiona automáticamente el ciclo de vida de los componentes, lo que significa que solo actualiza a los observadores activos. Esto previene problemas asociados con las fugas de memoria y actualizaciones innecesarias.
3. **Trabajo Conjunto con ViewModel:** LiveData y ViewModel suelen trabajar en conjunto. En el modelo (ViewModel), se establecen los atributos a observar mediante MutableLiveData.

Para obtener más detalles y ejemplos prácticos, puedes consultar la documentación oficial de LiveData en este enlace.

Vamos a mejorar la observabilidad de las variables `lives` y `secretWordDisplay` utilizando LiveData.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE LIVE DATA

DEPENDENCIA

Para incorporar la funcionalidad deseada, agrega la siguiente dependencia a tu archivo `build.gradle`:

```
implementation "androidx.lifecycle:lifecycle-livedata-core-ktx:2.6.2"
```

Este paso es esencial para utilizar las características más recientes y eficientes de LiveData en tu aplicación Android.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

PROPIEDADES MUTABLES

Propiedades LiveData y Acceso a Valores

Para utilizar propiedades LiveData, es necesario emplear una clase genérica junto con el tipo de dato que se desea almacenar.

```
var secretWordDisplay = MutableLiveData<String>()
var lives = MutableLiveData<Int>(8)
```

Acceder a los valores de estas propiedades requiere el uso de métodos específicos:

```
secretWordDisplay.value = generateSecretWordDisplay()
```

En el caso de propiedades que podrían contener valores nulos, como `lives`, se utiliza el operador Elvis para manejar nulos:

```
lives.value = lives.value?.minus(1)
fun lost() = lives.value ?: 0 <= 0
```

Observadores

Se pueden definir observadores para realizar acciones específicas cuando los datos cambian. Aquí hay un ejemplo de cómo crear un observador para un tipo de dato específico:

```
val nameObserver = Observer<String> { newName ->
    // Actualizar la interfaz de usuario, en este caso, un TextView.
    nameTextView.text = newName
}
```

Este observador, en este caso, llamado `nameObserver`, reaccionará a los cambios en el LiveData de tipo `String`. Puedes personalizar las acciones dentro del bloque lambda para actualizar la interfaz de usuario según tus necesidades.