APUNTES CURSO: ASPECTOS BÁSICOS EN KOTLIN

Javier Suarez

URL: https://developer.android.com/courses/android-basics-kotlin/course

Contenido

Unidad 1: Kotlin Basics

1.1.-Introduction to Kotlin

Codelab: Write your first program in Kotlin

-Nociones básicas del lenguaje Kotlin. Se muestra cómo hacer *print*s:

```
fun main() {
    println("Hello, world!")
}
```

Codelab: Create a birthday message in Kotlin

-Se muestra, entre otras cosas, cómo se declara una variable y se accede a su valor en un *print*.

-Declarar una variable con **"*val*"** la hace inmutable.

-Declarar una variable con **"*var*"** la hace mutable.

```
val age = 5
println("${age} is the very best age to celebrate!")
```

-También se muestra el uso de funciones auxiliares llamadas desde *main()*.

1.2.-Create your first app

Codelab: Download and Install Android Studio

-**Android Studio** como IDE oficial para desarrollo de Apps en Android. Usa IntelliJ IDEA.

-Se explica proceso de instalación del IDE mostrando requisitos del sistema y cómo realizar la instalación.

Codelab: Create and run your first Android App

-Se muestra cómo crear una primera app a partir de la plantilla común "**Empty Activity**" que ofrece el IDE.

-Se muestra cómo crear un perfil de dispositivo para probar las apps vía el emulador del IDE en la opción "*Device Manager*".

Codelab: Run your App on a Mobile Device

-Se muestra cómo establecer comunicación entre el IDE y un dispositivo físico para realizar pruebas directamente en dispositivo sin uso de emulador.

-Se debe habilitar en "*Developer options*" el "*USB Debugging*", instalar el "*Google USB Driver*" si se está en Windows, conectar el dispositivo al PC y permitir acceso al IDE cuando se solicite.

Codelab: Learn the basics of Android Tests

-Se introduce al testing en el IDE mostrando los dos tipos de prueba: Unitarias y de Instrumentación.

-**Unitarias**: Se evalúa el funcionamiento del código con casos a probar.

-**Instrumentación**: La prueba ejecuta la app y realiza una interacción comprobando si cumple con lo que se especifica en la prueba.

1.3.-Build a Basic Layout

Codelab: Create a Birthday App Card

-Se muestra cómo crear una primera app explicando diversos elementos:

-**View**: Cada uno de los elementos que se ven en pantalla (botones, texto, imágenes, etc.).

-**ViewGroup**: Contenedor de objetos View. Organiza las vistas que contiene.

-**ConstraintLayout**: Es un tipo de ViewGroup que ayuda a organizar las vistas de manera flexible.

-Se introduce al **Editor de Diseño** de Android Studio indicando sus partes y como cambiar los atributos de un **TextView**.

Codelab: Add Images to your Android App

-Se muestra cómo añadir imágenes al proyecto haciendo uso del **Resource Manager** y sus opciones de importar "*drawables*". Se muestra cómo añadir imágenes al diseño a través de **ImageView** especificando sus atributos y restricciones.

-Se muestra cómo hacer uso de **recursos de strings (strings.xml)** que sirven para facilitar las traducciones de estas.

1.4.-Add a button to an app

Codelab: Classes and Object instances in Kotlin

-Se introduce a una app consistente en tirar un dado y mostrar en imágenes el resultado obtenido.

-Se muestra el tipo de datos **IntRange** (1..6) y el método *random()* que se le puede aplicar.

-Se muestra cómo crear **clases** y **objetos** en Kotlin.

Codelab: Create an interactive Dice Roller App

-Se crea la app Dice Roller en Android Studio comenzando por su diseño del botón con la view **Button**.

-Se introduce la **Activity**: proporciona la ventana en la que la app dibuja su IU. **MainActivity** es la primera actividad o también denominada actividad de nivel superior. Cada pantalla se muestra mediante una actividad.

-Método **onCreate()**. Cómo usa el override y configura el MainActivity con el diseño inicial.

-Se muestra cómo habilitar las importaciones automáticas en el IDE.

-Método **findViewById()** para seleccionar una view por su id (**R.id.button**).

-Método **setOnClickListener()** para establecer un listener al botón.

-**Toast** para mostrar mensajes temporales como los print.

-Opción de *Reformat Code* del IDE para cumplir con las buenas prácticas.

Codelab: Add conditional behavior in Kotlin

-Estructuras de control: if-else y **when**.

```
when (rollResult) {
        luckyNumber -> println("You won!")
        1 -> println("So sorry! You rolled a 1. Try again!")
        2 -> println("Sadly, you rolled a 2. Try again!")
        3 -> println("Unfortunately, you rolled a 3. Try again!")
        5 -> println("Don't cry! You rolled a 5. Try again!")
        6 -> println("Apologies! You rolled a 6. Try again!")
}
```

Codelab: Add images to the Dice Roller App

-Se cambia la app para que haga uso de imágenes añadidas mediante el Resource Manager. Se cambia el diseño y el código para referenciar dichas imágenes. Por ejemplo, se hace lo siguiente:

```
val drawableResource = when (diceRoll) {
   1 -> R.drawable.dice_1
   2 -> R.drawable.dice_2
   3 -> R.drawable.dice_3
   4 -> R.drawable.dice_4
   5 -> R.drawable.dice_5
   else -> R.drawable.dice_6

}

diceImage.setImageResource(drawableResource)
```

Codelab: Write unit tests

-Introducción a las pruebas unitarias a través de **JUnit**. Métodos **assert()**.

```
class ExampleUnitTest {
   @Test
   fun addition_isCorrect() {
      assertEquals(4, 2 + 3)
   }

   @Test
   fun generates_number() {
      val dice = Dice(6)
      val rollResult = dice.roll()
      assertTrue("The value of rollResult was not between 1 and 6", rollResult in 1..6)
   }

}
```

Codelab: Intro to debugging

-Se crea un nuevo proyecto para introducir al **Debug** por vía del registro (**Logcat**) y la clase **Log**.

```
fun logging() {
    Log.e(TAG, "ERROR: a serious error like an app crash")
    Log.w(TAG, "WARN: warns about the potential for serious errors")
    Log.i(TAG, "INFO: reporting technical information, such as an operation
succeeding")
    Log.d(TAG, "DEBUG: reporting technical information useful for debugging")
    Log.v(TAG, "VERBOSE: more verbose than DEBUG logs")
}
```

Unidad 2: Layouts

2.1.-Get user input in an app: Part 1

Codelab: Classes and inheritance in Kotlin

-Se explica la Jerarquía de Clases y la Herencia en Kotlin. Ejemplo de las viviendas.

-Clases abstractas y sus subclases.

-Estructura with para simplificar código.

```
with(squareCabin) {
    println("\nSquare Cabin\n============")
    println("Capacity: ${capacity}")
    println("Material: ${buildingMaterial}")
    println("Has room? ${hasRoom()}")
}
```

-En Kotlin las **clases no abstractas** (clases normales) son "definitivas" por lo que, por defecto, no pueden crearse subclases a partir de ellas. Solo se puede heredar de las clases abstractas y de aquellas normales que tengan la palabra clave **open**.

Codelab: Create XML layouts for Android

-Se introduce aquí el editor de diseños de Android Studio en su vista XML.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/my_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

-**xmlns** es el espacio de nombres XML. Ver atributos y elementos (**widgets**) posibles en la web.

Codelab: Calculate the tip

-Se hace uso de la primera versión de **TipTimeApp**.

-**Vinculación de vista**: Método para facilitar el código eliminando la necesidad de usar findViewById() para cada elemento. Para usarlo, se debe agregar lo siguiente a la sección "*android*" del fichero **build.gradle** correspondiente al módulo de la app.

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

-Se cambia el código de MainActivity a lo siguiente para agregar la vinculación de vistas inicializando un objeto guardado en **binding** para acceder a las vistas de **activity_main.xml** y estableciendo la vista de contenido de la actividad a la raíz de la jerarquía de vistas (**binding.root**).

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

-Esto permite lo siguiente:

```
// Old way with findViewById()
val myButton: Button = findViewById(R.id.my_button)
myButton.text = "A button"

// Better way with view binding
val myButton: Button = binding.myButton
myButton.text = "A button"

// Best way with view binding and no extra variable
binding.myButton.text = "A button"
```

-Para la app en cuestión, se pueden hacer cosas como lo siguiente tras setContentView() para añadir un listener que ejecute una función.

```
binding.calculateButton.setOnClickListener{ calculateTip() }
```

2.2.-Get user input in an app: Part 2

Codelab: Change the app theme

-Introducción a **Material Design**, una colección de Google de colores y temas de estilo. Un tema es una colección de estilos que se aplican a una actividad, vista o grupo de vistas.

-Diferentes colores disponibles como colorPrimary, colorSecondary, etc.

-Colores especificados en **colors.xml**.

-Estilos especificados en **themes.xml**.

-Estilos para el tema oscuro especificados en **themes.xml (night)**.

-Herramienta **Color Tool** de la web de Material Design.

Codelab: Change the app icon

-Se explica aquí todo lo relacionado con el ícono de la app que se ve desde el menú de aplicaciones del móvil.

-Dimensiones de mapas de bits:

- mdpi: Recursos para pantallas de densidad media (~160 dpi)
- hdpi: Recursos para pantallas de densidad alta (~240 dpi)
- xhdpi: Recursos para pantallas de densidad muy alta (~320 dpi)
- xxhdpi: Recursos para pantallas de densidad muy muy alta (~480 dpi)
- xxxhdpi: Recursos para pantallas de densidad extremadamente alta (~640 dpi)
- nodpi: Recursos que no están diseñados para su escalamiento, independientemente de la densidad de píxeles de la pantalla
- anydpi: Recursos que se escalan a cualquier densidad

-Iconos adaptables (redondos y cuadrados).

-Herramienta **Image Asset** del Resource Manager.

Codelab: Create a more polished user experience

-Se habla de los **componentes de Material**, widgets de IU comunes que facilitan la implementación de los estilos de Material. Recomendado usarlos siempre.

-Para agregarlos, añadir lo siguiente al fichero build.gradle de la aplicación:

```
dependencies {
    ...
    implementation 'com.google.android.material:material:<version>'
}
```

-Se muestran ejemplos de componentes de esta colección para la TipTimeApp (segunda versión).

-Se muestra como añadir iconos (pngs) dentro de la app.

-**ScrollView** alrededor de ConstraintLayout para permitir el scroll cuando se rota el dispositivo en horizontal.

-Para ocultar tecla Intro al pulsar Enter, hacer lo siguiente:

```
private fun handleKeyEvent(view: View, keyCode: Int): Boolean {
   if (keyCode == KeyEvent.KEYCODE_ENTER) {
       // Hide the keyboard
       val inputMethodManager =
           getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager
       inputMethodManager.hideSoftInputFromWindow(view.windowToken, 0)
       return true
   }
   return false
}
```

-Ajustar app para usar **TalkBack**.

Codelab: Write Instrumentation Tests.

-Se muestra cómo crear las pruebas de instrumentación. Uso de **Espresso**.

```
@RunWith(AndroidJUnit4::class)
class CalculatorTests {

   @get:Rule()
   val activity = ActivityScenarioRule(MainActivity::class.java)

   @Test
   fun calculate_20_percent_tip() {
       onView(withId(R.id.cost_of_service_edit_text)).perform(typeText("50.00"))
       onView(withId(R.id.calculate_button)).perform(click())
       onView(withId(R.id.tip_result))
             .check(matches(withText(containsString("$10.00"))))
   }
}
```

2.3.-Display a scrollable list

Codelab: Use Lists in Kotlin.

-Introducción a las **List**. Listas de solo lectura (List<T>) y modificables (**MutableList<T>**).

-Métodos listOf(), size(), acceso a elementos, first(), last(), contains(), sorted(), add() etc.

-Bucles while y for para recorrer listas.

Codelab: Use RecyclerView to display a scrollable list.

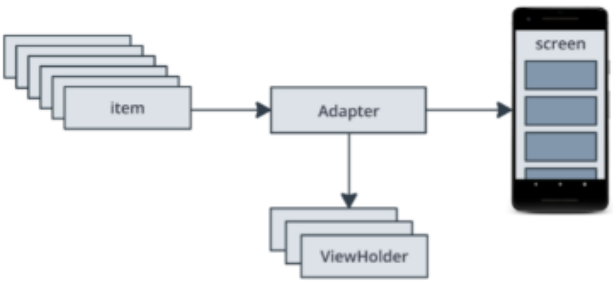
-Se hará uso de la app de **Affirmations**.

-Se crea una **clase de datos**, *Affirmation*, para lo que es necesario añadir la palabra clave "**data**".

-Se crea una clase, *Datasource*, para cargar en una lista los recursos de string de la app.

-**RecyclerView**: está diseñado para apps con listas siendo muy eficiente, incluso con listas grandes, ya que puede reutilizar o reciclar las vistas que se desplazan fuera de la pantalla. Cuando se desplaza un elemento de la lista fuera de la pantalla, RecyclerView reutiliza la vista del siguiente elemento que se mostrará. Esto significa que el elemento se rellena con contenido nuevo que se desplaza hacia la pantalla. Este comportamiento de RecyclerView ahorra mucho tiempo de procesamiento y ayuda a las listas a desplazarse de forma más fluida. Posee diversos elementos:

- **Item**: Un elemento de datos de la lista para mostrar. Representa un objeto Affirmation en tu app.
- **Adapter**: Toma datos y los prepara para que RecyclerView los muestre.
- **ViewHolders**: Un conjunto de vistas para que RecyclerView lo use y reutilice a fin de mostrar afirmaciones.
- **RecyclerView**: Vistas en la pantalla.



-RecyclerView admite mostrar elementos en diferentes formas como verticalmente (**LinearLayoutManager**) o en cuadrícula (**GridLayoutManager**).

-***Adapter*** es un patrón de diseño que adapta los datos a algo que puede usar RecyclerView. Cuando se ejecuta la app, RecyclerView usa el adaptador para descifrar cómo mostrar tus datos en la pantalla. RecyclerView le pide al adaptador que cree una nueva vista de elementos de lista para el primer elemento de datos de tu lista. Una vez que tenga la vista, le solicitará al adaptador que proporcione los datos para dibujar el elemento. Este proceso se repite hasta que RecyclerView no necesita más vistas para cubrir la pantalla. Si solo 3 vistas de elementos de la lista se ajustan a la pantalla a la vez, RecyclerView solo solicita al adaptador que prepare esas 3 vistas de elementos de lista (en lugar de las 10 vistas de elementos de lista).

-Cada elemento de RecyclerView tiene su propio diseño, que se define en un archivo de diseño separado (**list_item.xml**).

-El adaptador (ItemAdapter en la app actual), necesita información sobre cómo resolver los recursos de strings. Esta y otra información sobre la app se almacena en una instancia del objeto **Context**, que puedes pasar a una instancia ItemAdapter.

-**ViewHolder:** RecyclerView no interactúa directamente con las vistas de elementos, sino con ViewHolders. Un objeto ViewHolder representa una sola vista de elementos de lista en RecyclerView y se puede volver a usar cuando sea posible. Una instancia ViewHolder contiene referencias a las vistas individuales dentro de un diseño de elemento de lista (por eso se denomina, "view holder", "contenedor de vistas"). Esto facilita la actualización de la vista de elemento de lista con datos nuevos. Los contenedores de vistas también agregan información que RecyclerView usa para mover las vistas de manera eficiente por la pantalla.

-La clase Adapter extiende de RecyclerView.Adapter

-La clase ViewHolder se suele poner anidada en la Adapter y extiende de RecyclerView.ViewHolder.

-Tres funciones de la clase Adapter a conocer:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemViewHolder {
    TODO("Not yet implemented")
}

override fun getItemCount(): Int {
    TODO("Not yet implemented")
}

override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
    TODO("Not yet implemented")
}
```

-**getItemCount()** debe mostrar el tamaño del conjunto de datos.

-**onCreateViewHolder()** se usa por el administrador de diseño a fin de crear nuevas interfaces de vista para RecyclerView (cuando no hay contenedores de vistas existentes que puedan reutilizarse). Recuerda que un contenedor de vistas representa una sola vista de elementos de lista. **Parámetros:**

-***parent***: es el grupo de vistas al que se adjuntará la nueva vista de elemento de la lista como elemento secundario. El superior es el RecyclerView.

-***viewType***: importante cuando hay varios tipos de vistas de elementos en el mismo RecyclerView. Si tienes diferentes diseños de elementos de lista en RecyclerView, existen diferentes tipos de vistas de elementos. Solo puedes reciclar vistas con el mismo tipo de vista de elemento.

-Implementación:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemViewHolder {
    // create a new view
    val adapterLayout = LayoutInflater.from(parent.context)
        .inflate(R.layout.list_item, parent, false)

    return ItemViewHolder(adapterLayout)
}
```

-onBindViewHolder() es llamado por el administrador de diseño para reemplazar el contenido de una vista de elementos de lista. tiene dos parámetros: ***ItemViewHolder***, creado anteriormente por el método onCreateViewHolder(), y ***un int*** que representa el elemento actual position en la lista. En este método, encontrarás el objeto Affirmation adecuado del conjunto de datos según la posición.

-Implementación:

```
override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
    val item = dataset[position]
    holder.textView.text =  context.resources.getString(item.stringResourceId)
}
```

-Para usar el RecyclerView en MainActivity, hacer lo siguiente en onCreate().

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Initialize data.
        val myDataset = Datasource().loadAffirmations()

        val recyclerView = findViewById<RecyclerView>(R.id.recycler_view)
        recyclerView.adapter = ItemAdapter(this, myDataset)

        // Use this setting to improve performance if you know that changes
        // in content do not change the layout size of the RecyclerView
        recyclerView.setHasFixedSize(true)
}
```

Codelab: Display a list of images using cards.

-En este codelab se añade a partir del anterior soporte para ver imágenes aparte del texto en el RecyclerView.

-En primer lugar, se usa las **anotaciones de recursos**, que agregan información adicional a las clases, los métodos o los parámetros, para indicar en la clase de datos Affirmation las id de los recursos de string y de imagen.

-Tras esto se recogen los recursos de string e imagen en la clase ItemViewHolder y en el método onBindViewHolder() se establecen los valores de holder a dichos recursos.

-Se usan tarjetas para mostrar las imágenes junto al texto.

Codelab: Test Lists and Adapters.

-Este codelab muestra como testear los elementos vistos en el anterior.

Unidad 3: Navigation

3.1.-Navigate between screens

Codelab: Collections in Kotlin

-Se habla de las **Colecciones de Kotlin** (mutables e inmutables) y sus métodos donde List y MutableList son algunas de ellas.

-Se habla de **Set** (sin duplicados y sin importar el orden).

-Se habla de **Map** (clave-valor).

-Se presenta el **foreach** -> `collection.forEach { print("${it.key} is ${it.value}, ") }`

-Se presenta el **map** (No confundir con Map, es para aplicar transformaciones):
-> `println(collection.map { "${it.key} is ${it.value}" }.joinToString(", ") )`

-Se presenta el **filter** -> `collection.filter { it.key.length < 4 }`

-**Funciones Lambda**: función sin nombre que se puede usar de inmediato como expresión. Las asociaciones de Listeners usan Lambdas.

-**Tipos de funciones**: puedes definir un tipo específico de función según sus parámetros de entrada y valor que aparece -> `(Int) -> Int`

-Ejemplo combinado de Lambda y tipos de funciones:

```
val triple: (Int) -> Int = { a: Int -> a * 3 }
```

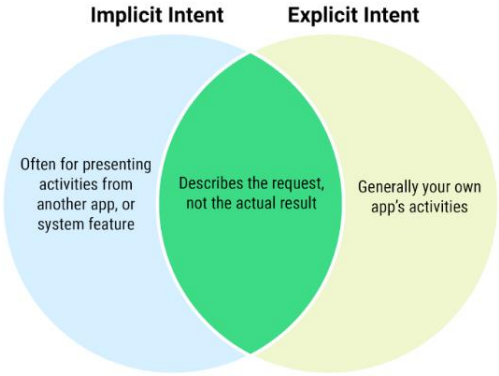
-**Funciones de orden superior**: pasar una función a otra función o mostrar una función desde otra. Las funciones map, filter y forEach son ejemplos de funciones de orden superior.

Codelab: Activities and Intents

-Se presenta la app **Words** que se usará (MainActivity, DetailActivity, Word y Letter Adapter…)

-**Intents**: es un objeto que representa una acción que se realizará. El uso más común de un intent consiste en iniciar una actividad, aunque no es el único. Hay dos tipos de intents: **implícito** y **explícito**. Un **intent explícito** es muy específico y se usa cuando conoces la actividad precisa que se iniciará (a menudo, se trata de una pantalla en la app).

Un **intent implícito** es un poco más abstracto y se usa para indicar al sistema el tipo de acción, como abrir un vínculo, redactar un correo electrónico o realizar una llamada telefónica, y el sistema es responsable de determinar la forma en que completará la solicitud. En general, cuando se muestra una actividad en la app, se usa un intent explícito.



-**Extra**: es un dato al que se le asigna un nombre que se puede recuperar más adelante, como un número o una string. Esto es similar a pasar un argumento cuando se llama a una función.

-**Creación del botón de Menú**: Clase MenuItem y métodos onCreateOptionsMenu y onOptionsItemSelected.

-**onCreateOptionsMenu**: Con él, se aumenta el menú de opciones y se realiza cualquier configuración adicional.

-**onOptionsItemSelected**: Con él, se ejecuta una función a establecer cuando se seleccione el botón.

Codelab: Stages of the Activity lifecycle

-En este codelab se explica el **ciclo de vida** de las actividades usando Logcat para verlo.

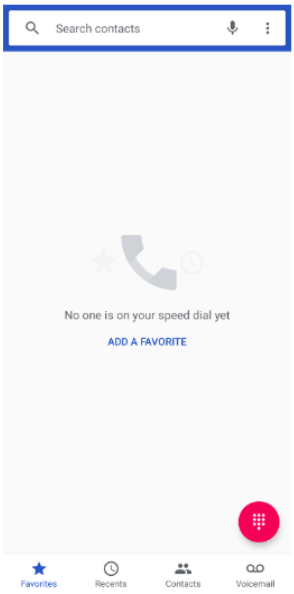
-**Funciones**: onCreate(), onStart(), onRestart(), onResume(), onPause(), onStop() y onDestroy().

-**Función onSaveInstance()**: es una devolución de llamada que se usa para guardar los datos que se podrían necesitar si se destruyera la Activity. En el diagrama de devolución de llamada de ciclo de vida, se llama a onSaveInstanceState() después de que se detiene la actividad. Se llama cada vez que la app pasa a segundo plano. Es análogo al localStorage de programación web.

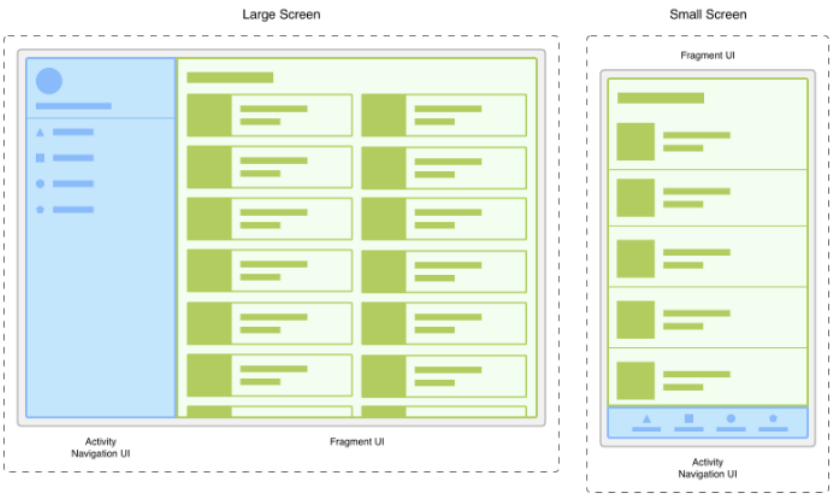
3.2.-Introduction to the Navigation Component

Codelab: Fragments and the Navigation Component

-**Fragmento**: Si bien resulta útil conocer los intents, son solo una parte de la historia de crear interfaces de usuario dinámicas para las apps. Muchas apps para Android no necesitan una actividad separada para cada pantalla. De hecho, muchos patrones de IU comunes, como las pestañas, existen dentro de una única actividad y utilizan algo llamado *fragmentos*. Un **fragmento** es una parte reutilizable de la IU. Los fragmentos se pueden reutilizar e incorporar en una o más actividades.



-En la captura de pantalla anterior, al presionar una pestaña, no se activa un intent para mostrar la siguiente pantalla. En cambio, cuando se cambia de pestaña, simplemente, se cambia el fragmento anterior por otro. Todo esto ocurre sin iniciar otra actividad. Incluso se puede mostrar varios fragmentos a la vez en una única pantalla, como un diseño principal y de detalles en tablets. En el ejemplo que figura a continuación, tanto la IU de navegación de la izquierda como el contenido de la derecha pueden estar en un fragmento independiente. Ambos fragmentos existen simultáneamente en la misma actividad.



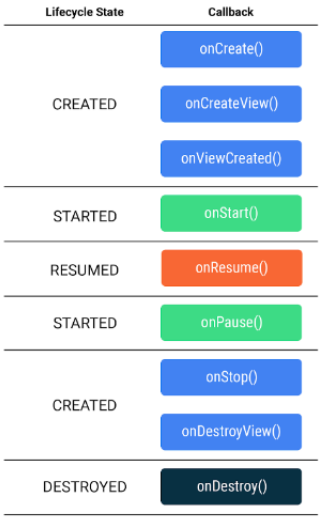
-En este codelab se aprende el uso de Fragmentos y del componente Navigation de **Android Jetpack**. Se modificará la app **Words**.

-Los fragmentos tienen un **ciclo de vida** y pueden responder a las entradas del usuario. Un fragmento siempre se encuentra dentro de la jerarquía de vistas de una actividad cuando se muestra en pantalla. Debido a su énfasis en la reutilización y la modularidad, es posible que varios fragmentos se alojen de forma simultánea en una única actividad. Cada fragmento administra su propio ciclo de vida independiente.

-Ciclo de vida de los fragmentos y sus funciones:

- INICIALIZADO: Se creó una nueva instancia del fragmento.
- CREADO: Se llamó a los primeros métodos del ciclo de vida del fragmento. Durante este estado, también se creará la vista asociada con el fragmento.
- COMENZADO: El fragmento está visible en pantalla, pero no tiene "foco", lo cual significa que no puede responder a las entradas del usuario.
- REANUDADO: El fragmento está visible y tiene foco.
- DESTRUIDO: Se eliminó la instancia del objeto del fragmento.

- `onCreate()`: Se creó la instancia del fragmento y se encuentra en el estado `CREATED`. Sin embargo, todavía no se creó la vista correspondiente.
- `onCreateView()`: Este es el método en el cual se aumenta el diseño. El fragmento pasa al estado `CREATED`.
- `onViewCreated()`: Se llama a este método después de que se crea la vista. En este método, por lo general, debes vincular vistas específicas con propiedades llamando a `findViewById()`.
- `onStart()`: El fragmento pasa al estado `STARTED`.
- `onResume()`: El fragmento pasa al estado `RESUMED` y ahora tiene foco (puede responder a las entradas del usuario).
- `onPause()`: El fragmento vuelve a pasar al estado `STARTED`. El usuario puede ver la IU.
- `onStop()`: El fragmento vuelve a pasar al estado `CREATED`. Se crearon instancias del objeto, pero ya no se presentan en pantalla.
- `onDestroyView()`: Se llama a este método justo antes de que el fragmento pase al estado `DESTROYED`. La vista ya se ha quitado de la memoria, pero el objeto del fragmento aún existe.
- `onDestroy()`: El fragmento pasa al estado `DESTROYED`.



-**Diferencia entre onCreate() de Actividades y Fragmentos**: Con las actividades, se puede usar este método a fin de aumentar el diseño y vincular vistas. No obstante, en el ciclo de vida del fragmento, se llama a onCreate() antes de crear la vista, de modo que no puedes aumentar el diseño aquí. En cambio, se debe hacer en onCreateView(). Después de crear la vista, se llama al método onViewCreated(), en el que se podrá vincular propiedades a vistas específicas.

-**FragmentLetterListBinding** para realizar vinculación de vistas con fragmentos.

-Operador **"?"** para garantizar seguridad nula y Operador "**!!**" cuando se asegura que no será nulo.

-**Componente de Navigation de Jetpack**: Lo proporciona Android Jetpack para facilitar la implementación de la navegación y tiene **tres partes principales**:

-**Navigation Graph**: Es un archivo en formato XML que proporciona una representación visual de la navegación en la app. El archivo consta de *destinos* que corresponden a actividades y fragmentos individuales, así como a acciones que pueden usarse en el código para navegar de un destino a otro. Al igual que los archivos de diseño, Android Studio proporciona un editor visual que permite agregar destinos y acciones al gráfico de navegación.

-**NavHost**: Se usa NavHost para mostrar los destinos de un gráfico de navegación en una actividad. Cuando se navega entre fragmentos, se actualiza el destino que se muestra en NavHost. Existe una implementación integrada, llamada **NavHostFragment**.

-**NavController**: El objeto NavController permite controlar la navegación entre los destinos que se muestran en el NavHost. Cuando se trabaja con intents, se debe llamar a **startActivity** a fin de navegar a una pantalla nueva. Con el componente de Navigation, se puede llamar al método **navigate()** de NavController a efectos de intercambiar el fragmento que se muestra. El NavController también ayuda a manejar tareas comunes, como responder al botón "arriba" del sistema para volver al fragmento que se mostró anteriormente.

-Para usar este componente, se requiere añadir en el archivo build.gradle del nivel de proyecto, en **buildscript > ext**, debajo de *material_version*, la **nav_version** y en el archivo build.gradle del nivel de la app, agrega lo siguiente al grupo de dependencias:

```
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

-**Complemento SafeArgs**: un complemento de Gradle que ayuda con la seguridad de tipo cuando se pasan datos entre fragmentos. Para integrarlo:

-En el archivo build.gradle de nivel superior, en **buildscript > dependencies**, agregar la siguiente Ruta de clase.

```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
```

-En el archivo build.gradle del nivel de la app, dentro de plugins en la parte superior, agrega `androidx.navigation.safeargs.kotlin`

-Clase **FragmentContainerView** para mostrar los destinos del gráfico de navegación al usuario en lugar de usar clase NavGraph.

Codelab: Test Navigation Components

-En este codelab se muestra como probar el componente Navigation con pruebas de instrumentación.

-Se requiere añadir en el archivo build.gradle del módulo de app las siguientes dependencias:

```
androidTestImplementation 'com.android.support.test.espresso:espresso-contrib:3.4.0'
androidTestImplementation 'androidx.navigation:navigation-testing:2.4.2'
debugImplementation 'androidx.fragment:fragment-testing:1.4.1'
```

-Se explican las **anotaciones @Before, @BeforeClass, @After y @AfterClass** para reducir código repetido. @BeforeClass se ejecuta para la clase, @Before se ejecuta antes que las funciones, @After se ejecuta después de las funciones y @AfterClass se ejecuta para la clase.

3.3.-Architecture Components

Codelab: Store data in ViewModel

-Se hará uso de la app **Unscramble**.

-Las **bibliotecas de Android Jetpack** son una colección de bibliotecas que facilitan el desarrollo de apps de Android. Estas bibliotecas ayudan a seguir prácticas recomendadas, liberan de escribir código estándar y simplifican tareas complejas para que poder concentrarse en el código que interesa, como la lógica de la app.

-Los **componentes de la arquitectura de Android** forman parte de las bibliotecas de Android Jetpack para ayudar a diseñar apps con una buena arquitectura. Los componentes de la arquitectura proporcionan orientación sobre la arquitectura de las apps, y es la práctica recomendada.

-Las clases o los componentes principales de la arquitectura de Android son el **controlador de IU (actividad/fragmento), ViewModel, LiveData y Room**. Estos componentes se encargan de la complejidad del ciclo de vida y ayudan a evitar problemas relacionados con el ciclo de vida.

-Controlador de IU (actividad/fragmento): Las actividades y los fragmentos son controladores de IU. Los controladores de IU controlan las IU dibujando vistas en la pantalla, capturando eventos de los usuarios y todo lo relacionado con la IU con la que el usuario interactúa. Los datos de la app o cualquier lógica de toma de decisiones relacionados con esos datos no deberían estar en las clases de los controladores de IU. El sistema Android puede destruir los controladores de IU en cualquier momento en función de ciertas interacciones del usuario o debido a condiciones del sistema, como memoria insuficiente. Debido a que no se puede controlar estos eventos, no se debe almacenar datos ni estados de la app en los controladores de IU. En su lugar, la lógica de toma de decisiones sobre los datos debe agregarse en ViewModel.

-**ViewModel**: es un modelo de los datos de app que se muestran en las vistas. Los modelos son componentes responsables de manejar los datos de una app. Permiten que la app siga el principio de arquitectura de controlar la IU a partir del modelo. almacena los datos relacionados con la app que no se destruyen cuando el framework de Android destruye la actividad o el fragmento y los recrea. Los objetos ViewModel se retienen automáticamente (no se destruyen como la actividad o una instancia de fragmento) durante los cambios de configuración, de manera que los datos que conservan están disponibles de inmediato para la siguiente instancia de fragmento o actividad. **Es lo equivalente al modelo en la arquitectura MVC.**

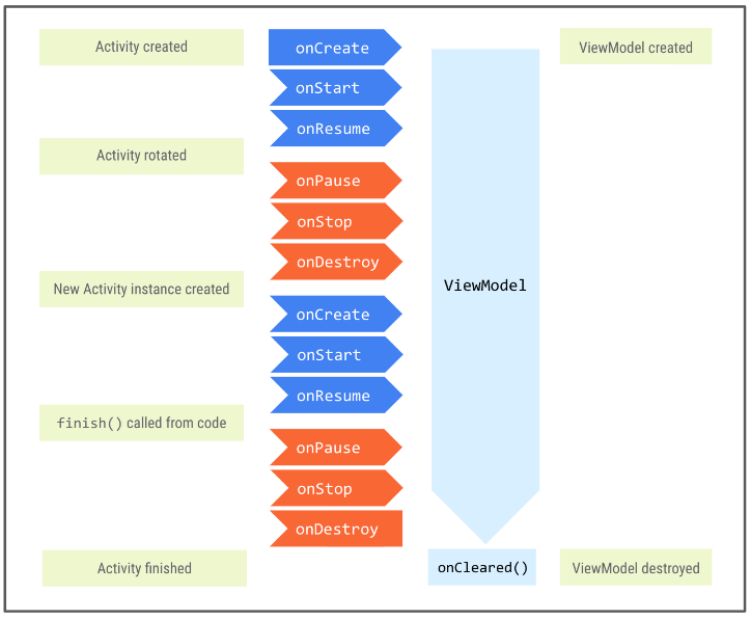
-**Delegado de propiedad de Kotlin**: En Kotlin, cada **propiedad mutable (var)** tiene funciones del método get y el método set que se generan automáticamente para ella. Se llama a las funciones del método get y el método set cuando asignas un valor o lees el valor de la propiedad. En el caso de una **propiedad de solo lectura (val)**, difiere levemente de una propiedad mutable. Solo la función de método get se genera de forma predeterminada. Se llama a esta función de método get cuando se lee el valor de una propiedad de solo lectura. La **delegación de propiedades en Kotlin** permite transferir la responsabilidad del método get y el método set a una clase diferente. Esta clase (que se denomina *clase de delegado*) brinda funciones del método get y el método set de la propiedad y controla sus cambios. La clase delegada crea el objeto viewModel en el primer acceso y retiene su valor mediante los cambios de configuración y muestra el valor cuando se solicita. Una propiedad de delegado se define mediante la **cláusula by** y una instancia de clase delegada:

```
var <property-name> : <property-type> by <delegate-class>()
```

-**Propiedad de copia de seguridad**: Consiste en una copia de solo lectura pública de un atributo privado que si se puede modificar en este ámbito.

```
private var _count = 0
val count: Int get() = _count
```

-**Ciclo de vida de un ViewModel**:

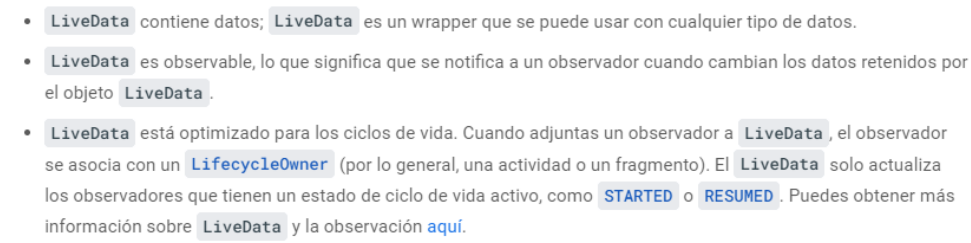


-**Inicialización tardía (lateinit)**: Si es seguro la inicialización de una propiedad antes de usarla, se puede declarar con lateinit. La memoria no se asigna a la variable hasta que se inicializa. Si se intenta acceder a la variable antes de inicializarla, la app fallará.

-La clase **MaterialAlertDialog** permite crear diálogos de alerta hacia el usuario.

Codelab: Use LiveData with ViewModel

-**LiveData**: es una clase de retención de datos observable que tiene en cuenta el ciclo de vida. **MutableLiveData** es su versión mutable. Algunas características:



- `LiveData` contiene datos; `LiveData` es un wrapper que se puede usar con cualquier tipo de datos.
- `LiveData` es observable, lo que significa que se notifica a un observador cuando cambian los datos retenidos por el objeto `LiveData`.
- `LiveData` está optimizado para los ciclos de vida. Cuando adjuntas un observador a `LiveData`, el observador se asocia con un `LifecycleOwner` (por lo general, una actividad o un fragmento). El `LiveData` solo actualiza los observadores que tienen un estado de ciclo de vida activo, como `STARTED` o `RESUMED`. Puedes obtener más información sobre `LiveData` y la observación aquí.

-Método **observe()** para agregar un observador.

```
// Observe the scrambledCharArray LiveData, passing in the LifecycleOwner and the
observer.
viewModel.currentScrambledWord.observe(viewLifecycleOwner,
   { newWord ->
       binding.textViewUnscrambledWord.text = newWord
   })
```

-**Vinculación de Datos**: También forma parte de la biblioteca de Android Jetpack. La vinculación de datos vincula los componentes de IU de los diseños con las fuentes de datos de la app usando un formato declarativo. En términos más simples, la vinculación de datos asocia datos (del código) con las vistas y la vinculación de vistas (vinculación de vistas con el código).

Ejemplo con vinculación de vistas en el controlador de IU:

```
binding.textViewUnscrambledWord.text = viewModel.currentScrambledWord
```

Ejemplo con vinculación de datos en un archivo de diseño:

```
android:text="@{gameViewModel.currentScrambledWord}"
```

El ejemplo anterior muestra cómo usar la biblioteca de vinculación de datos para asignar datos de app a las vistas o al widget directamente en el archivo de diseño. Observar el uso de la sintaxis @{} en la expresión de asignación:

La principal ventaja de usar la vinculación de datos es que permite quitar varias llamadas al framework de la IU en las actividades, que resultan más sencillas y fáciles de mantener. Esto también puede mejorar el rendimiento de la app y ayudar a evitar pérdidas de memoria y excepciones de puntero nulo.

Se requiere hacer lo siguiente para añadir la vinculación de datos:

```
buildFeatures {
   dataBinding = true
}
plugins {
   id 'com.android.application'
   id 'kotlin-android'
   id 'kotlin-kapt'
}
```

-Para especificar una vista de fragmento como dueño del ciclo de vida de una vinculación con el objetivo de permitir que la vinculación pueda ver las actualizaciones del LiveData, hacer lo siguiente:

```
binding.lifecycleOwner = viewLifecycleOwner
```

3.4.-Advanced navigation app examples

Codelab: Shared ViewModel

-Se hará uso de la app **Cupcake**.

-En el siguiente código

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

        val navHostFragment = supportFragmentManager
                .findFragmentById(R.id.nav_host_fragment) as NavHostFragment
        val navController = navHostFragment.navController

        setupActionBarWithNavController(navController)
 }
```

-La llamada a **setupActionBarWithNavController(navController)** pasando la instancia de NavController muestra un título en la barra de la app según la etiqueta de destino y el botón **Up** (←) cuando no se esté en un destino de nivel superior.

-En este codelab se usa un **ViewModel** que se comparte entre diferentes fragmentos.

-Para ello, se hace uso de la vinculación de datos, de los delegados de propiedad y de la función de alcance (apply).

-Función de alcance (apply): Ejecuta un bloque de código dentro del contexto de un objeto. Forma un alcance temporal y, en ese alcance, puedes acceder al objeto sin su nombre. El caso de uso común para apply es configurar un objeto. Estas llamadas se pueden leer como *"aplicar las siguientes asignaciones al objeto"*. Ejemplo:

```
clark.apply {
    firstName = "Clark"
    lastName = "James"
    age = 18
}

// The equivalent code without apply scope function would look like the following.

clark.firstName = "Clark"
clark.lastName = "James"
clark.age = 18
```

-**Vinculaciones de objetos de escucha**: son expresiones lambda que se ejecutan cuando se produce un evento, por ejemplo, un evento onClick. Son similares a las referencias de métodos, como *textview.setOnClickListener(clickListener)*, pero las vinculaciones de objetos de escucha te permiten ejecutar expresiones arbitrarias de vinculación de datos.

-El framework de Android brinda una clase denominada **SimpleDateFormat**, que es una clase para formatear y analizar fechas de una manera que tiene en cuenta la configuración regional. Permite dar formato a las fechas (fecha → texto) y analizar (texto → fecha) las fechas.

Para poder crear una instancia de SimpleDateFormat, pasa una string de patrón y una configuración regional:

```
SimpleDateFormat("E MMM d", Locale.getDefault())
```

-**LifecycleOwner**: es una clase que tiene un ciclo de vida de Android, por ejemplo, una actividad o un fragmento. Un observador LiveData ve los cambios en los datos de la app solo si el propietario del ciclo de vida está en estado activo (STARTED o RESUMED).

-**Métodos de transformación LiveData**: brindan una manera de realizar manipulaciones de datos en la fuente LiveData y mostrar un objeto LiveData resultante. En términos simples, transforman el valor de LiveData en otro valor. Estas transformaciones no se calculan a menos que un observador esté viendo el objeto LiveData. **Transformations.map()** es una de las funciones de transformación. Este método toma la fuente LiveData y una función como parámetros. La función manipula la fuente LiveData y muestra un valor actualizado que también es observable.

Codelab: Navigation and the backstack

-En este codelab se implementa el **botón Up** y un **botón Cancel** de la app de Cupcake que borra todo un pedido y vuelve a pantalla principal.

-Para el botón de Up, se puede usar el siguiente método para darle soporte:

```
override fun onSupportNavigateUp(): Boolean {
   return navController.navigateUp() || super.onSupportNavigateUp()
}
```

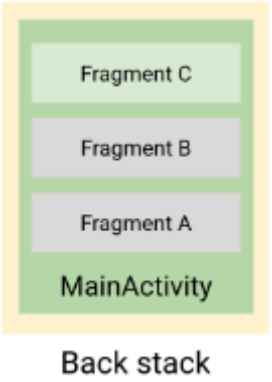
-**Tareas**: Las actividades en Android existen dentro de las tareas. Cuando se abre una app por primera vez desde el ícono de selector, Android crea una tarea nueva con la actividad principal. Una *tarea* es una colección de actividades con las que el usuario interactúa cuando realiza un trabajo determinado (por ejemplo, revisar un correo electrónico, crear un pedido de cupcake o tomar una foto).

Las actividades se organizan en una pila, conocida como ***pila de actividades***, en la que cada actividad nueva que visita el usuario se envía a la pila de actividades para la tarea.

La pila de actividades es útil cuando el usuario quiere navegar hacia atrás. Android puede quitar la actividad actual de la parte superior de la pila, destruirla y volver a iniciarla en la parte inferior. Se conoce como quitar una actividad de la pila y poner la actividad anterior en primer plano para que el usuario interactúe con ella. Si el usuario desea regresar varias veces, Android continuará quitando las actividades de la parte superior de la pila hasta que se acerque a la parte inferior.

Cuando no hay más actividades en la pila, el usuario regresa a la pantalla del selector del dispositivo (o a la app que la inició).

De la misma manera que la pila de actividades puede llevar un seguimiento de las actividades que el usuario abrió, la pila de actividades también puede realizar un seguimiento de los destinos de fragmentos que visitó el usuario con la ayuda del componente de Navigation de Jetpack. La biblioteca de Navigation permite abrir un destino de fragmento fuera de la pila de actividades cada vez que el usuario presiona el botón **Back**. Este comportamiento predeterminado viene gratis, sin necesidad de implementar nada. Solo se debe escribir código si se necesita un comportamiento personalizado de la pila de actividades.



Codelab: Test ViewModel and LiveData

-En este codelab se muestra como probar los ViewModel y los LiveData.

3.5.-Adaptative Layouts

Codelab: Adaptative Layouts

-En este codelab se explica el diseño adaptable a varios dispositivos con la app **Sports**.

-**Patrón de SlidingPaneLayout**: Es posible que una IU de lista y detalles deba comportarse de manera diferente según el tamaño de la pantalla. En las pantallas grandes, hay espacio suficiente para que los paneles de lista y detalles estén uno al lado del otro. Al hacer clic en un elemento de la lista, se muestran los detalles en el panel de detalles. Sin embargo, las pantallas pequeñas podrían parecer abarrotadas. En lugar de mostrar ambos paneles al mismo tiempo, es mejor mostrar uno a la vez. Inicialmente, el panel de lista ocupa toda la pantalla. Al presionar un elemento, se reemplaza el panel de lista con el panel de detalles de ese elemento, que también ocupa la pantalla.

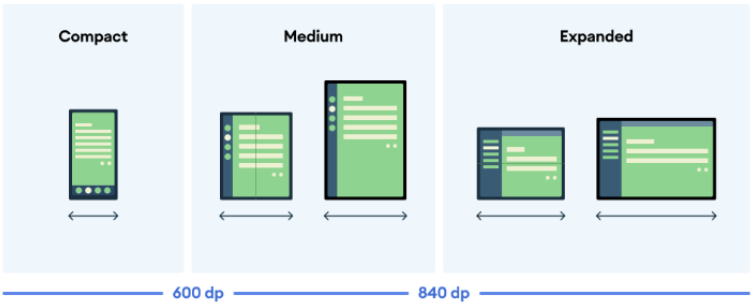
-Existe un XML propio de este patrón.

-Se requiere añadir lo siguiente al fichero build.gradle:

```
dependencies {
...
    implementation "androidx.slidingpanelayout:slidingpanelayout:1.2.0-beta01"
}
```

-Medidas:

Ancho	Punto de interrupción	Representación del dispositivo
Ancho compacto	Menos de 600 dp	99.96% de los teléfonos en orientación vertical
Ancho medio	600 dp o más	El 93.73% de las tablets en portraitLarge desplegaron pantallas internas en orientación vertical
Ancho expandido	840 dp o más	El 97.22% de las tablets en landscapeLarge desplegaron pantallas internas en orientación horizontal.

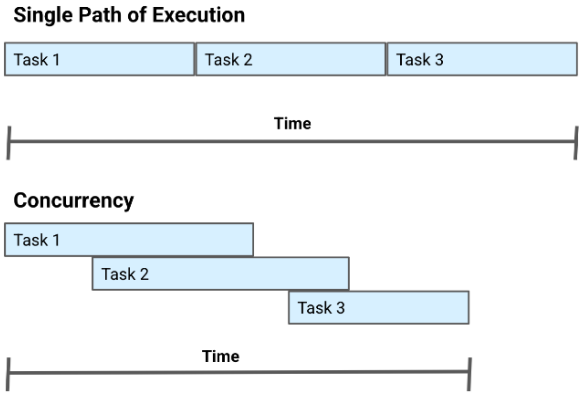


Unidad 4: Connect to the Internet

4.1.-Coroutines

Codelab: Introduction to Coroutines

-La **simultaneidad** permite que varias unidades de código se ejecuten de forma desordenada o que aparenten hacerlo en paralelo, lo que permite un uso más eficiente de los recursos. El sistema operativo puede usar las características del sistema, el lenguaje de programación y la unidad de simultaneidad para administrar varias tareas al mismo tiempo.



-Un **subproceso** es la unidad de código más pequeña que puede programarse y ejecutarse en los límites de un programa.

```
fun main() {
    val thread = Thread {
        println("${Thread.currentThread()} has run.")
    }
    thread.start()
}
```

-Condición de Carrera: se da cuando varios subprocesos intentan acceder al mismo valor en memoria al mismo tiempo. Pueden generar errores difíciles de reproducir y en apariencia aleatorios.

-La creación y el uso de subprocesos para tareas en segundo plano directamente tiene su lugar en Android, pero Kotlin también ofrece **corrutinas** que proporcionan una forma más flexible y fácil de administrar la simultaneidad.

-Corrutinas en Kotlin: permiten realizar varias tareas a la vez, pero ofrecen otro nivel de abstracción por sobre trabajar solamente con subprocesos. Una función clave de las corrutinas es la capacidad de **almacenar el estado**, de modo que pueden detenerse y reanudarse. Una corrutina puede ejecutarse o no hacerlo.

-El **estado**, representado por las ***Continuations***, permite que partes del código indiquen cuándo necesitan otorgar el control o esperar a que otra corrutina complete su trabajo antes de reanudar. Este flujo se denomina realización de varias tareas a la vez de forma cooperativa. La implementación de corrutinas de Kotlin agrega varias funciones para ayudar a realizar varias tareas a la vez.

-Además de las Continuations, la creación de una corrutina integra ese trabajo en un **Job**, una unidad de trabajo cancelable con un ciclo de vida, dentro de un **CoroutineScope**.
Un CoroutineScope es un contexto que aplica la cancelación y otras reglas a sus elementos secundarios y a los secundarios de estos de forma recurrente. Un **Dispatcher** administra el subproceso de copia de seguridad que la corrutina usará para su ejecución, lo que le quitará al desarrollador la responsabilidad de determinar cuándo y dónde usar un subproceso nuevo.

Job	Es una unidad de trabajo cancelable, como una creada con la función `launch()`.
CoroutineScope	Las funciones que se usan para crear corrutinas nuevas como `launch()` y `async()` extienden `CoroutineScope`.
Dispatcher	Determina el subproceso que usará la corrutina. El despachador `Main` siempre ejecutará corrutinas en el subproceso principal, mientras que aquellos como `Default`, `IO` o `Unconfined` usarán otros subprocesos.

```
fun main() {
    repeat(3) {
        GlobalScope.launch {
            println("Hi from ${Thread.currentThread()}")
        }
    }
}
```

-GlobalScope: Es el Alcance Global del código. No se recomienda usar este alcance por los problemas de las corrutinas.

-launch(): crea una corrutina a partir del código encerrado unido a un objeto Job cancelable. launch() se usa cuando no se necesita que se muestre un valor fuera de los límites de la corrutina.

-suspend: indica que un bloque de código o una función se pueden pausar o reanudar.

-runBlocking(): inicia una corrutina nueva y bloquea el subproceso actual hasta que esta se completa. En especial, se usa para conectar los códigos que provocan bloqueos con los que no lo hacen en funciones y pruebas principales. No se suele utilizar en general en el código Android.

-async(): muestra un valor de tipo **Deferred**. Un Deferred es un Job cancelable que puede contener una referencia a un valor futuro. Mediante Deferred, aún se puede llamar a una función como si mostrara de inmediato un valor: solo sirve como marcador de posición, dado que no se puede saber cuándo se mostrará el resultado de una tarea asincrónica. Un Deferred (también llamado **Promise o Future** en otros lenguajes) garantiza que se mostrará un valor a este objeto más adelante. Por otro lado, una tarea asíncrona no bloqueará ni esperará la ejecución de manera predeterminada. A fin de iniciar que la línea de código actual espere el resultado de un Deferred, se puede llamar a **await()** en él. Se mostrará el valor sin procesar.

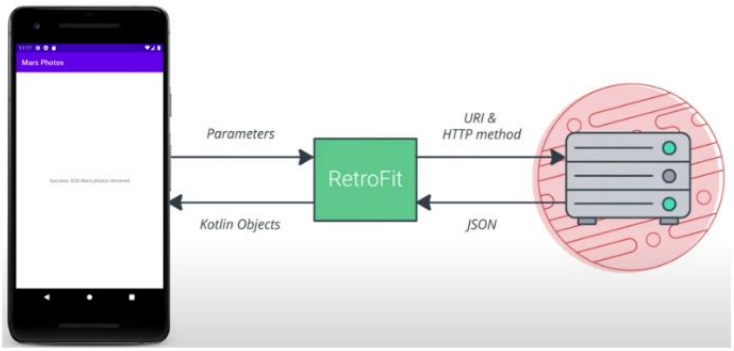
4.2.-Get data from the internet

Codelab: Get data from the internet

-Se usa la app **MarsPhotos**.

-**Servicios web y protocolo HTTP**: GET, POST, PUT, DELETE.

-**Biblioteca Retrofit**: se comunica con el backend server. Crea URI para el servicio web según los parámetros que se le pasan. Retrofit crea una API de red para la app basada en el contenido del servicio web. Recupera datos del servicio web y los enruta a través de una biblioteca de conversor independiente que sabe cómo decodificar los datos y mostrarlos en forma de objetos como String. Retrofit incluye compatibilidad integrada para formatos de datos populares, como XML y JSON. Retrofit crea el código para llamar y consumir este servicio, incluidos los detalles críticos, como la ejecución de solicitudes en subprocesos en segundo plano.



-En archivo build.gradle a nivel de proyecto añadir su **repositorio (jcenter)**.

```
repositories {
   google()
   jcenter()
}
```

-En archivo build.gradle a nivel de módulo, agregar en la sección **dependencias**:

```
// Retrofit
implementation "com.squareup.retrofit2:retrofit:2.9.0"
```

-Comprobar la **compatibilidad con Java 8**

```
android {
  ...

  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }

  kotlinOptions {
    jvmTarget = '1.8'
  }
}
```

-**Retrofit.Builder()**: para compilar y crear un objeto Retrofit.

-Retrofit requiere el **URI base** para el servicio web y una fábrica de conversión para crear una API de servicios web. El conversor le indica a Retrofit qué hacer con los datos que obtiene del servicio web.

-Retrofit requiere de una **interface** para definir cómo se comunica con el servidor web mediante solicitudes HTTP. Dentro de esta interface se definen las funciones anotándolas con el método HTTP deseado.

```
interface MarsApiService {
    @GET("photos")
    fun getPhotos(): String
}
```

-**Declaraciones de objetos**: se usan para declarar objetos **singleton**. Garantiza que una y solo una instancia de un objeto se cree y se tiene un punto de acceso global a ese objeto. La inicialización de la declaración del objeto es segura para los subprocesos y se realiza durante el primer acceso. Kotlin facilita la declaración de singleton. A continuación, se muestra un ejemplo de una declaración de objeto y su acceso. La declaración del objeto siempre tiene un nombre que sigue a la palabra clave object.

```
// Object declaration
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}

// To refer to the object, use its name directly.
DataProviderManager.registerDataProvider(...)
```

-**ViewModelScope**: es el alcance integrado de corrutinas definido para cada ViewModel en la app

-**Permisos de Android**: El objetivo de los permisos de Android es proteger la privacidad de un usuario de Android. Las apps para Android deben declarar o solicitar permisos a fin de acceder a datos sensibles del usuario, como contactos, registros de llamadas y ciertas funciones del sistema, como la cámara o Internet. Para que una app pueda acceder a Internet, necesita el **permiso INTERNET**. La conexión a Internet presenta problemas de seguridad, por lo que las apps no tienen conexión a Internet de forma predeterminada. Se debe declarar explícitamente que la aplicación necesita acceso a Internet. Para ello, hay que ir al fichero Android.manifest y añadir lo siguiente:

```
<uses-permission android:name="android.permission.INTERNET" />
```

-Se explica el **manejo de excepciones** con try-catch.

-Se explican los objetos JSON y su tratamiento con Moshi.

-**Moshi Converter**: es un analizador de JSON de Android que convierte una string JSON en objetos Kotlin. Retrofit tiene un conversor que funciona con Moshi

-En archivo build.gradle a nivel de módulo, agregar, en dependencias:

```
implementation 'com.squareup.moshi:moshi-kotlin:1.9.3'
```

-Para usarlo con Retrofit:

```
// Retrofit with Moshi Converter
implementation 'com.squareup.retrofit2:converter-moshi:2.9.0'
```

-Moshi require de una **clase de datos** para almacenar los objetos Kotlin convertidos a partir de JSON.

-**Anotaciones @Json**: Para usar nombres de variables en la clase de datos que difieren de los nombres de clave en la respuesta JSON.

Codelab: Load and display images from the internet

-**Biblioteca Coil**: Facilita para descargar, almacenar en búfer, decodificar y almacenar en caché las imágenes. Requiere la URL de la imagen y un objeto ImageView para mostrar la imagen.

-En archivo build.gradle a nivel de módulo, agregar, en dependencias:

```
implementation "io.coil-kt:coil:1.1.1"
```

-En archivo build.gradle a nivel de proyecto añadir su **repositorio (mavenCentral())**.

-**Adaptadores de Vinculación**: son métodos anotados que se usan a fin de crear métodos set personalizados para propiedades personalizadas de la vista. Por lo general, cuando se establece un atributo en el XML mediante el código *android:text="Sample Text"*, el sistema Android busca automáticamente un método set con el mismo nombre que el atributo text, que se establece mediante el método *setText(String: text)*. El método *setText(String: text)* es un método set para algunas vistas que proporciona el framework de Android. Un comportamiento similar se puede personalizar con los adaptadores de vinculación; puedes proporcionar un atributo y una lógica personalizados a la que llamará la biblioteca de vinculación de datos.

-Ejemplo:

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:imageUrl="@{product.imageUrl}"/>


@BindingAdapter("imageUrl")
fun bindImage(imgView: ImageView, imgUrl: String?) {
    imgUrl?.let {
        // Load the image in the background using Coil.
        }
    }
}
```

-**Función de alcance let**: es una de las **funciones Scope** de Kotlin que te permite ejecutar un bloque de código dentro del contexto de un objeto. se usa para invocar una o más funciones en los resultados de las cadenas de llamadas.

-Se explican las clases **enum** en Kotlin.

Codelab: Test network requests

-Se explica cómo realizar pruebas para las solicitudes en la red. Se requiere añadir JSON con las solicitudes a realizar con las dependencias necesarias. Se muestra el ejemplo de un test con un MockWebServer.

```
class MarsApiServiceTests : BaseTest() {

    private lateinit var service: MarsApiService

    @Before
    fun setup() {
        val url = mockWebServer.url( path: "/")
        service = Retrofit.Builder()
            .baseUrl(url)
            .addConverterFactory(MoshiConverterFactory.create(
                Moshi.Builder()
                    .add(KotlinJsonAdapterFactory())
                    .build()
            ))
            .build()
            .create(MarsApiService::class.java)
    }

    @Test
    fun api_service() {
        enqueue( fileName: "mars_photos.json")
        runBlocking {  this: CoroutineScope
            val apiResponse = service.getPhotos()

            assertNotNull(apiResponse)
            assertTrue( message: "The list was empty", apiResponse.isNotEmpty())
            assertEquals( message: "The id's did not match",  expected: "424905", apiResponse[0].id)
        }
    }
}
```

Codelab: Debug with breakpoints

-Se muestra cómo establecer los puntos de breakpoint o interrupción en la ejecución del código, así como el uso de condiciones con el objetivo de depuración. Los Watches sirven para supervisar variables específicas.

Unidad 5: Data persistence

5.1.-Introduction to SQL, Room and Flow

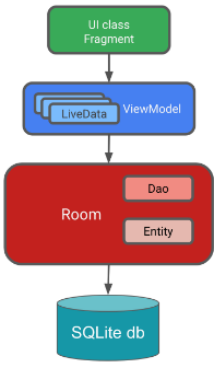
Codelab: SQL Basics

-Se explican conceptos básicos de Bases de Datos como: Bases de Datos Relacionales, Clave Primaria, SELECT, INSERT, UPDATE, DELETE, LIMIT, WHERE, COUNT, SUM, MAX, DISTINCT, ORDER BY, GROUP BY. Se muestra el **Inspector de Base de Datos** de Android Studio.

Codelab: Introduction to Room and Flow

-Se usa la app **BusSchedule**.

-**Biblioteca Room**: es una biblioteca de persistencias que forma parte de Android Jetpack. Es una capa de abstracción que se ubica sobre una base de datos SQLite. SQLite usa un lenguaje especializado (SQL) para realizar operaciones de bases de datos. En lugar de usar SQLite directamente, Room simplifica las tareas de implementar, configurar y usar la base de datos. Room también proporciona comprobaciones de tiempo de compilación de las sentencias de SQLite.



-En archivo build.gradle a nivel de proyecto definir room versión en bloque ext

```
ext {
   kotlin_version = "1.6.20"
   nav_version = "2.4.1"
   room_version = '2.4.2'
}
```

-En archivo build.gradle a nivel de módulo, añadir dependencias

```
implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"

// optional - Kotlin Extensions and Coroutines support for Room
implementation "androidx.room:room-ktx:$room_version"
```

-**Entidad**: Clase que representa una tabla de base de datos para Room (ORM). Se usa la clase de datos para esta clase.

```
@Entity
data class Schedule(
   @PrimaryKey val id: Int,
   @NonNull @ColumnInfo(name = "stop_name") val stopName: String,
   @NonNull @ColumnInfo(name = "arrival_time") val arrivalTime: Int
)
```

-**DAO (Data Access Object)**: es una clase de Kotlin que proporciona acceso a los datos. Es donde se incluyen funciones para leer y manipular datos. Llamar a una función en el DAO es el equivalente a ejecutar un comando de SQL en la base de datos.

```
@Dao
interface ScheduleDao {
     @Query("SELECT * FROM schedule ORDER BY arrival_time ASC")
     fun getAll(): List<Schedule>
}
```

-Se explica como usar los **ViewModel** para separar el código de la IU y su modelo de datos. Para ello, se usan las Factory para crear "fábricas" del elemento en cuestión dentro del ViewModel.

-**Clase de Base de Datos**: Es subclase de RoomDatabase y sirve para interconectar la Entidad, el DAO y la ViewModel. Contiene la base de datos y es el punto de acceso principal para la conexión subyacente a la base de datos de la app. La clase de base de datos proporciona a tu app instancias de los DAO asociados con esa base de datos.

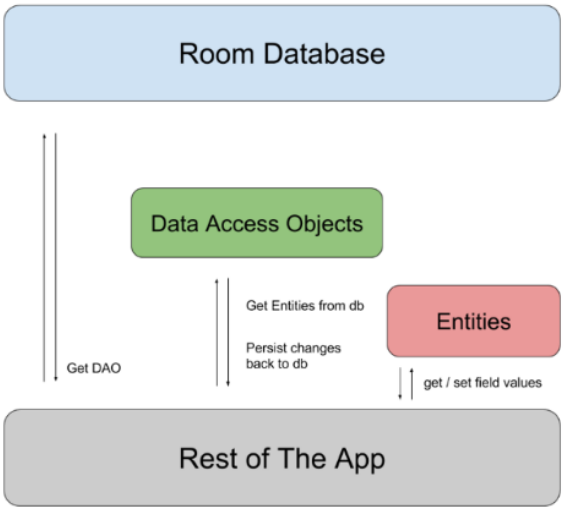
```
/**
 * Defines a database and specifies data tables that will be used.
 * Version is incremented as new tables/columns are added/removed/changed.
 * You can optionally use this class for one-time setup, such as pre-populating a database.
 */
@Database(entities = arrayOf(Schedule::class), version = 1)
abstract class AppDatabase: RoomDatabase() {
    abstract fun scheduleDao(): ScheduleDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            return INSTANCE ?: synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                    context,
                    AppDatabase::class.java,
                     name: "app_database")
                    .createFromAsset( databaseFilePath: "database/bus_schedule.db")
                    .build()
                INSTANCE = instance

                instance  ^synchronized
            }
        }
    }
}
```

-Una vez creada, se debe proporcionar una subclase personalizada de la clase **Application** y crear una propiedad **lazy** que contenga el resultado de getDatabase().

```
import android.app.Application
import com.example.busschedule.database.AppDatabase

class BusScheduleApplication : Application() {
    val database: AppDatabase by lazy { AppDatabase.getDatabase( context: this) }
}
```

-En el **Android Manifest**:

```
<application
    android:name="com.example.busschedule.BusScheduleApplication"
```



-Para los **ListAdapter** de las **RecyclerView**, existe **AsyncListDiffer** para determinar las diferencias entre una lista de datos antigua y una nueva. Luego, la vista de reciclador solo se actualiza en función de las diferencias entre las dos listas. Como resultado, la vista de reciclador es más eficaz cuando se manejan datos que se actualizan con frecuencia, como se realiza en una aplicación de base de datos.

-**Flow**: Consiste en una función de Kotlin que permite las actualizaciones dinámicas en la app, evitando la necesidad de reiniciarla cuando se producen cambios en la base de datos. Para ello, esta función permite al DAO emitir datos de forma continua desde la base de datos y con métodos propios actualizar la IU.

5.2.-Use Room for data persistence

Codelab: Persist data with Room

-Este codelab profundiza en conceptos del anterior definiéndolos completamente y proporciona las imágenes y definiciones de elementos como Entidad o DAO del anterior apartado. Además, realiza la creación de los elementos para la app **Inventory**.

Codelab: Read and update data with Room

-El codelab amplia funciones de la app **Inventory** para editar y eliminar artículos.

-**Funciones de extensión**: Kotlin permite extender una clase con funcionalidades nuevas sin tener que heredar contenido de la clase ni modificar su definición existente. Eso significa que se puede agregar funciones a una clase existente sin necesidad de acceder a su código fuente. Esto se hace mediante declaraciones especiales llamadas *extensiones*. permiten usar la notación de puntos cuando se llama a la función en objetos de esa clase.

```
class Square(val side: Double){
        fun area(): Double{
        return side * side;
    }
}

// Extension function to calculate the perimeter of the square
fun Square.perimeter(): Double{
        return 4 * side;
}

// Usage
fun main(args: Array<String>){
      val square = Square(5.5);
      val perimeterValue = square.perimeter()
      println("Perimeter: $perimeterValue")
      val areaValue = square.area()
      println("Area: $areaValue")
}
```

-**Función copy()**: se proporciona de forma predeterminada a todas las instancias de clases de datos. Esta función se usa para copiar un objeto a fin de cambiar algunas de sus propiedades, pero no modificar el resto.

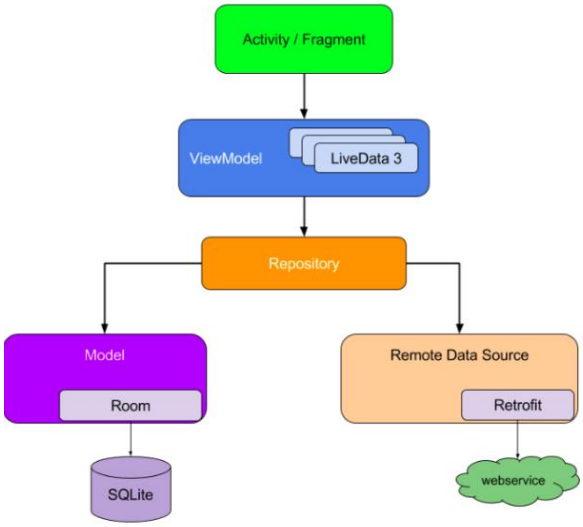
```
// Data class
data class User(val name: String = "", val age: Int = 0)

// Data class instance
val jack = User(name = "Jack", age = 1)

// A new instance is created with its age property changed, rest of the properties
unchanged.
val olderJack = jack.copy(age = 2)
```

Codelab: Repository Pattern

-En este codelab, se **mejora la UX** para una app con **almacenamiento en caché sin conexión**. Muchas apps dependen de datos de la red. Si la app recupera datos del servidor en cada inicio y el usuario ve una pantalla de carga, es posible que una mala experiencia del usuario los lleve a desinstalar la app. Debido a que la app podrá obtener datos de la red y guardar una caché sin conexión de los resultados descargados anteriormente, se necesita una manera de organizar todas estas fuentes de datos. Para ello, se implementará una **clase de repositorio**, que servirá como única fuente de verdad para los datos de la app y se abstraerá la fuente de los datos (red, caché, etc.) fuera del modelo de vista.

-**Patrón de Repositorios**: es un patrón de diseño que aísla la capa de datos del resto de la app. La capa de datos hace referencia a la parte de la app, independiente de la IU, que controla los datos y la lógica empresarial de la app, lo que expone API coherentes de modo que el resto de la app acceda a esos datos. Mientras que la IU presenta información al usuario, la capa de datos incluye elementos como el código de red, las bases de datos de Room, el manejo de errores y cualquier código que lea o manipule datos.

Un repositorio puede resolver conflictos entre fuentes de datos (como modelos persistentes, servicios web y cachés) y centralizar los cambios en estos datos.



-**Ventajas de usar un Repositorio**: Un módulo de repositorio controla operaciones de datos y permite usar varios backends. En una app real típica, el repositorio implementa la lógica para decidir si debe recuperar datos de una red o usar resultados almacenados en caché de una base de datos local. Con un repositorio, se puede intercambiar los detalles de la implementación, como la migración a una biblioteca de persistencia diferente, sin afectar el código de llamada, como los modelos de vista. Esto también permite que el código sea modular y se pueda probar. Se puede simular con facilidad el repositorio y probar el resto del código. Un repositorio debe funcionar como una única fuente de verdad para una parte específica de los datos de la app. Cuando se trabaja con varias fuentes de datos, como un recurso conectado en red y una caché sin conexión, el repositorio garantiza que los datos de la app sean lo más precisos y actualizados, lo que proporcionará la mejor experiencia posible incluso cuando la app esté sin conexión.

-**Almacenamiento en caché**: Se define el concepto de caché y se muestran varias formas de implementar el almacenamiento en caché en la red para Android.

Técnica de almacenamiento en caché	Usos
Retrofit es una biblioteca de herramientas de redes que se usa a fin de implementar un cliente de REST de tipo seguro para Android. Puedes configurar Retrofit de modo que almacene una copia de cada resultado de red de manera local.	Es una buena solución para respuestas y solicitudes simples, llamadas de red poco frecuentes o conjuntos de datos pequeños.
Puedes usar `DataStore` para almacenar pares clave-valor.	Es una buena solución para una pequeña cantidad de claves y valores simples, como la configuración de la app. No puedes usar esta técnica a los efectos de almacenar grandes cantidades de datos estructurados.
Puedes acceder al directorio de almacenamiento interno de la app y guardar los archivos de datos allí. El nombre del paquete de tu app especifica el directorio de almacenamiento interno de la app, que se encuentra en una ubicación especial del sistema de archivos de Android. Este directorio es privado para tu app y se borra cuando esta se desinstala.	Resulta una buena solución si tienes necesidades específicas que un sistema de archivos puede resolver, por ejemplo, si necesitas guardar archivos multimedia o de datos, y debes administrarlos por tu cuenta. No puedes usar esta técnica para almacenar datos estructurados y complejos que tu app necesite consultar.
Puedes almacenar datos en caché con Room, una biblioteca de asignación de objetos de SQLite que proporciona una capa de abstracción sobre SQLite.	Se trata de la solución recomendada para datos estructurados complejos que se pueden consultar, ya que la mejor manera de almacenar datos estructurados en el sistema de archivos de un dispositivo es mediante una base de datos SQLite local.

-El resto del codelab muestra como crear un repositorio usando la opción Room e integrarlo con la app **RepositoryPattern.**

Codelab: Preferences DataStore

-En este codelab, se presenta **Jetpack DataStore**. DataStore, que se basa en corrutinas y flujo de Kotlin, proporciona dos implementaciones diferentes: **Proto DataStore**, que almacena objetos escritos, y **Preferences DataStore**, que almacena pares clave-valor. En este codelab práctico, se aprende a usar Preferences DataStore. Se vuelve a usar la app de **Words**.

-**Preferences DataStore**: es ideal para conjuntos de datos pequeños y simples, como el almacenamiento de datos de inicio de sesión, la configuración del modo oscuro, el tamaño de la fuente, entre otros. DataStore no es adecuado para conjuntos de datos complejos, como una lista de inventario de tiendas de alimentos en línea o una base de datos de alumnos. Si se necesita almacenar conjuntos de datos grandes o complejos, se recomienda usar Room en lugar de DataStore. Con la biblioteca de Jetpack DataStore, se puede crear una API simple, segura y asíncrona para almacenar datos. Esta biblioteca ofrece dos implementaciones diferentes: Preferences DataStore y Proto DataStore. Si bien Preferences y Proto DataStore permiten el almacenamiento de datos, lo hacen de diferente manera:

- **Preferences DataStore** accede a los datos y los almacena en función de claves, sin definir un esquema (modelo de base de datos) por adelantado.
- **Proto DataStore** define el esquema mediante búferes de protocolo. El uso de estos búferes, llamados Protobufs, permite **conservar los datos de tipado fuerte**. Son más rápidos, más pequeños, más simples y menos ambiguos que XML y otros formatos de datos similares.

-En build.gradle a nivel de Módulo, añadir las dependencias:

```
implementation "androidx.datastore:datastore-preferences:1.0.0"
implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.3.1"
```

-**Funciones de tipo de clave**: Preferences DataStore no usa un esquema predefinido como Room, sino funciones de tipo de clave correspondientes a fin de definir una clave para cada valor que se almacene en la instancia de DataStore<Preferences>. Por ejemplo, si se desea definir una clave para un valor int, usar **intPreferencesKey()** y, para un valor string, usar **stringPreferencesKey()**. En general, estos nombres de funciones tienen el prefijo del tipo de datos que se quiere almacenar en la clave.

-**Escribir en Preferences DataStore**: Preferences DataStore proporciona una función de suspensión **edit()** que actualiza los datos de forma transaccional en DataStore. El parámetro de transformación de la función acepta un bloque de código en el que se puede actualizar los valores según sea necesario. Todo el código que esté en el bloque de transformación se tratará como una sola transacción. De forma interna, el trabajo de la transacción se mueve a **Dispacter.IO**, así que no olvidar establecer la función **suspend** cuando se llame a la función edit().

-**Leer desde Preferences DataStore**: Preferences DataStore expone los datos almacenados en un **Flow<Preferences>** que se emiten cada vez que se modifique una preferencia. No se recomienda exponer todo el objeto Preferences, sino solo el valor Boolean.

Unidad 6: WorkManager

6.1.-Schedule tasks with WorkManager

Codelab: Background Work with WorkManager

-Este codelab trata sobre **WorkManager**, una biblioteca con retrocompatibilidad, flexible y simple para realizar trabajos diferibles en segundo plano. WorkManager es el programador de tareas recomendado de Android para realizar trabajos diferibles, lo que garantiza su ejecución. Se usa la app **Blur-O-Matic**.

-**WorkManager** es parte de Android Jetpack y un componente de la arquitectura para trabajos en segundo plano que requieren una ejecución tanto oportunista como garantizada. La ejecución oportunista implica que WorkManager realizará el trabajo en segundo plano tan pronto como sea posible. La ejecución garantizada implica que WorkManager se encargará de la lógica a los efectos de iniciar el trabajo en diferentes situaciones, incluso si se sale de la app.

WorkManager es una biblioteca extremadamente flexible que cuenta con muchos beneficios adicionales. Por ejemplo:

- Es compatible con tareas asíncronas únicas y periódicas.
- Admite restricciones, como condiciones de red, espacio de almacenamiento y estado de carga.
- Encadena solicitudes de trabajo complejas, incluido el trabajo de ejecución en paralelo.
- Usa el resultado de una solicitud de trabajo como entrada para la siguiente.
- Controla la compatibilidad con el nivel de API 14 (consulta la nota).
- Trabaja con o sin los Servicios de Google Play.
- Sigue las prácticas recomendadas sobre el estado del sistema.
- Ofrece compatibilidad con LiveData a fin de mostrar fácilmente el estado de la solicitud de trabajo en la IU.

-WorkManager se apoya en algunas **API**, como **JobScheduler** y **AlarmManager**. WorkManager elige las API adecuadas para usar en función de condiciones como el nivel de API del dispositivo del usuario.

-Cuándo usarla: La biblioteca de WorkManager es una buena opción para las tareas que resultan útiles de completar, incluso si el usuario sale de una pantalla en particular o de la app. WorkManager ofrece una ejecución garantizada, y no todas las tareas lo necesitan. Por consiguiente, su función no es la de ejecutar todas las tareas del subproceso principal. Algunos ejemplos de tareas que muestran un buen uso de WorkManager:

- Subir registros
- Aplicar filtros a imágenes y guardar la imagen
- Sincronizar datos locales con la red de forma periódica

-En build.gradle a nivel de Módulo, añadir las dependencias:

```
implementation "androidx.work:work-runtime-ktx:$versions.work"
```

-Ubicar la versión más reciente en el apartado correspondiente de buld.gradle.

-Clases de WorkManager:

- `Worker`: Aquí es donde colocas el código del trabajo real que deseas realizar en segundo plano. Extenderás esta clase y anularás el método `doWork()`.
- `WorkRequest`: Esta clase representa una solicitud para realizar algunos trabajos. Como parte de la creación de tu `WorkRequest`, pasarás el `Worker`. Cuando hagas la `WorkRequest`, también podrás especificar elementos como `Constraints` sobre el momento en que se debe ejecutar el `Worker`.
- `WorkManager`: Esta clase programa tu `WorkRequest` y la ejecuta. Programa `WorkRequest`s de manera que se distribuya la carga sobre los recursos del sistema, respetando las restricciones que hayas especificado.

-La entrada y el resultado se pasan en un sentido y otro por medio de objetos **Data**. Los objetos Data son contenedores livianos para pares clave-valor. Tienen el propósito de almacenar una **pequeña** cantidad de datos que podrían pasar desde **WorkRequests** y hacia ellas.

-Cadenas de trabajo único: A veces, se desea que solo una cadena de trabajo se ejecute a la vez. Por ejemplo, tal vez se tenga una cadena de trabajo que sincroniza los datos locales con el servidor. Sería bueno permitir que la primera sincronización de datos termine antes de comenzar una nueva. Para hacerlo, se debe usar **beginUniqueWork** en lugar de **beginWith** y proporcionarle un nombre de String único. Esto nombrará la cadena **completa** de solicitudes de trabajo a fin de que se pueda hacer consultas y búsquedas en todas ellas.

-**WorkInfo**: es un objeto que contiene detalles sobre el estado actual de una WorkRequest, incluido lo siguiente:

- Si el trabajo fue `BLOCKED`, `CANCELLED`, `ENQUEUED`, `FAILED`, `RUNNING` o `SUCCEEDED`.
- Si se completó `WorkRequest`, los datos de salida del trabajo

-En la siguiente tabla, se muestran tres formas diferentes de obtener objetos LiveData<WorkInfo> o LiveData<List<WorkInfo>>, así como lo que hace cada uno.

Tipo	Método de WorkManager	Descripción
Obtener trabajo con un **ID**	`getWorkInfoByIdLiveData`	Cada `WorkRequest` tiene un ID único generado por WorkManager. Puedes usarlo a fin de obtener un único `LiveData` para esa `WorkRequest` exacta.
Obtener trabajo con un **nombre de cadena único**	`getWorkInfosForUniqueWorkLiveData`	Como acabas de ver, las `WorkRequest` pueden ser parte de una cadena única. Esto muestra el `LiveData>` de todo el trabajo en una sola cadena única de `WorkRequests`.
Obtener trabajo con una **etiqueta**	`getWorkInfosByTagLiveData`	Por último, también puedes etiquetar cualquier WorkRequest por medio de una String. Puedes etiquetar varias `WorkRequest` con la misma etiqueta a fin de asociarlas. Esto muestra el `LiveData>` de cualquier etiqueta individual.

-WorkManager admite **Constraints**. Para crear un objeto Constraints, se debe usar un **Constraints.Builder.**