Android

2020-2021

1. Introduction

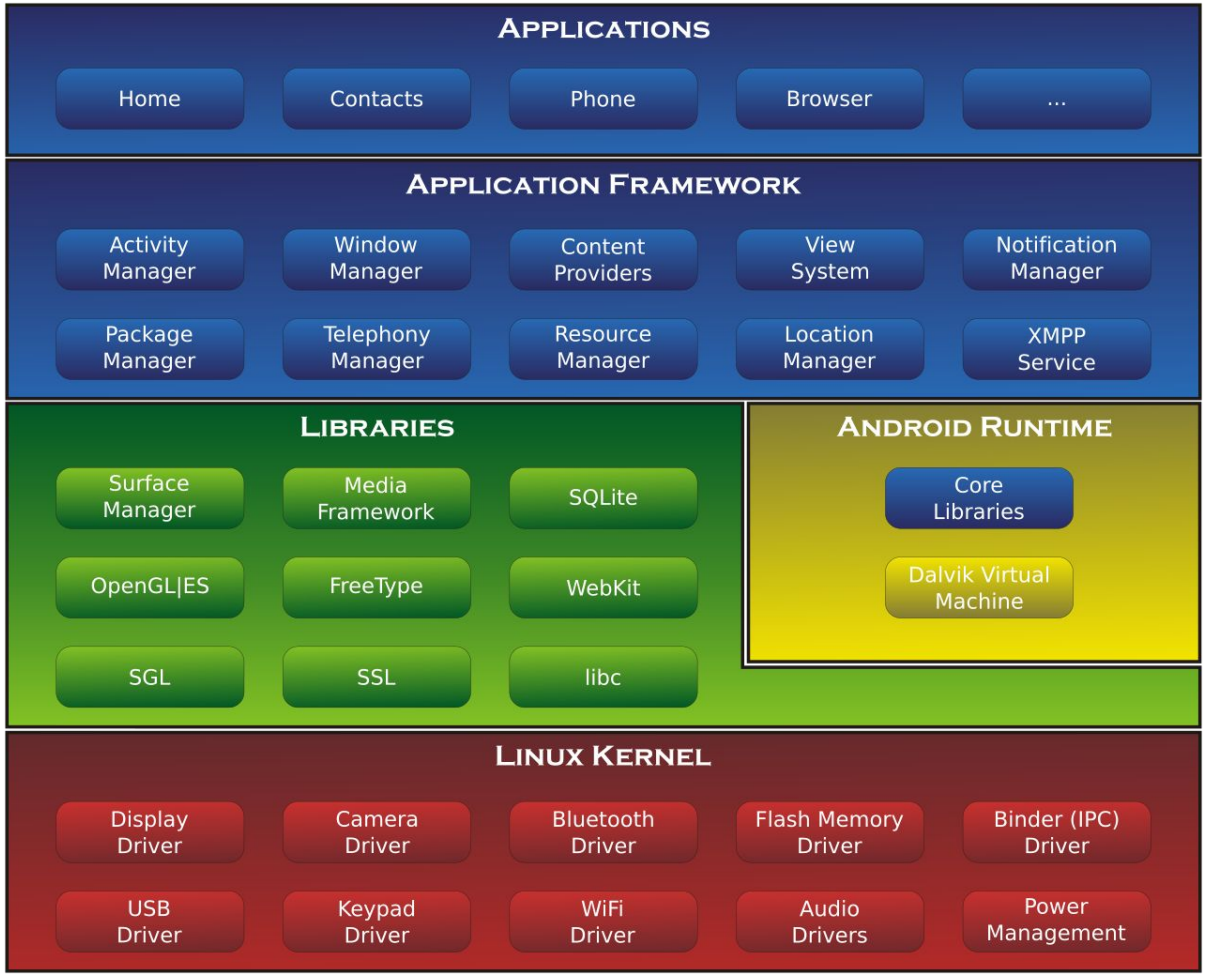
System architecture, versions, SDK, APK, gradle

Android OS

Properties

- Based on Linux kernel (Android is a Linux distribution)
- Open source
- UI for touchscreens
- Used on over 80% of smartphones
- Used on devices: smartphones, TVs, cars …

Source: https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-1-get-started/lesson-1-build-your-first-app/1-2-c-layouts-and-resources-for-the-ui/1-2-c-layouts-and-resources-for-the-ui.html



https://commons.wikimedia.org/wiki/File:Android-System-Architecture.svg#/media/File:Android-System-Architecture.svg

Android version history

Version	Marketing name	Release date	API level	Runtime	Launched with
11	11	February 19, 2020	30	ART	Google Pixel 2, Pixel 2 XL, Pixel 3, Pixel 3 XL, Pixel 3a, Pixel 3a XL, Pixel 4, Pixel 4XL[393]
10	10	September 3, 2019	29	ART	Pixel, Pixel XL, Pixel 2, Pixel 2 XL, Pixel 3, Pixel 3 XL, Pixel 3a, Pixel 3a XL[394]
9	Pie	August 6, 2018	28	ART	Essential Phone, Pixel, Pixel XL, Pixel 2, Pixel 2 XL, Nokia 7 Plus, OnePlus 6, Oppo R15 Pro, Sony Xperia XZ2, Vivo X21UD, Vivo X21, Xiaomi Mi Mix 2S[395]

https://en.wikipedia.org/wiki/Android_(operating_system)

Android SDK

The **Android SDK** is a collection of software **development** tools and libraries required to develop **Android** applications. Every time Google releases a new version of **Android** or an update, a corresponding **SDK** is also released which developers must download and install.

The Android SDK includes the following:

- Required libraries
- Debugger
- An emulator
- Relevant documentation for the Android application program interfaces (APIs)
- Sample source code
- Tutorials for the Android OS

Android apps

- System apps have no special status
- System apps provide key capabilities to app developers
  - your app can use a system app to send an SMS message

App:

- One or more interactive screens (Activity)
- Executed by Android Runtime (ART)
- Bundled in an **APK** (Android Package) = ZIP package

App building blocks

- Resources:
  - layouts, images, strings, colors, XML and media files
- Components:
  - activities, helper classes
- Manifest:
  - information about app for the runtime
- Build configuration:
  - Gradle: build tool

Create your first Android Studio project

Activity template: Empty Activity

Project folders:

- **manifests:** `AndroidManifest.xml`
- **java:** Java and Kotlin source files
- **res:** resources

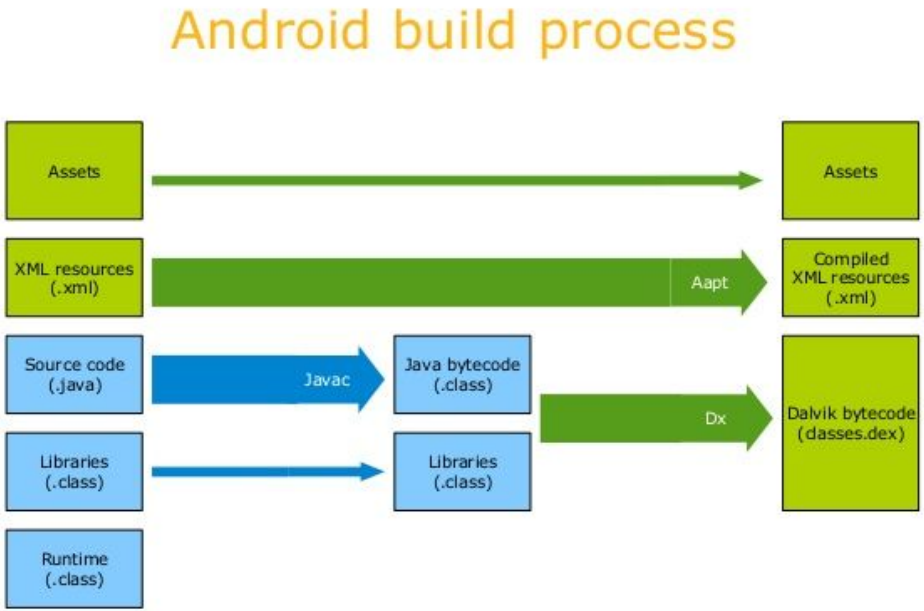
Gradle scripts:

`build.gradle`: Gradle build files for project and modules

Gradle build system

- Modern build system based on plugins
- Other build tools
  - Java: Ant, Maven
  - Linux: make
- How it is used in Android?
  - through the Android plugin for Gradle
- Tasks:
  - dependency management
  - creates the APKs

Android build process



2. Activity

Concept, creation, lifecycle

The concept of activities

Activity

- basic component of an Android app **started by the Android system**
- no main() method - **main activity** → the first screen
- generally implements **one screen** in an app
- minimal dependencies between activities (**loosely bounded**)
- has lifecycle managed by the Android system

Creating activities

1. Create an Activity class (Kotlin, Java)
2. Design the UI - XML
3. Declare the Activity class in AndroidManifest.xml

Creating activities

1. Create an Activity class

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

Creating activities

2. Design the UI - activity_main.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical" >
    <TextView android:id="@+id/text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Hello, I am a Button" />
</LinearLayout>
```

Creating activities

3. Declare the Activity class in AndroidManifest.xml

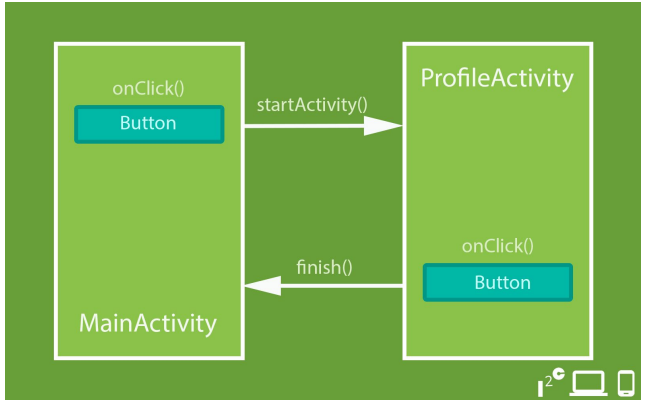
```
<activity android:name=".MainActivity" >

    <intent-filter>

        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />

    </intent-filter>

  </activity>
```

Starting an activity

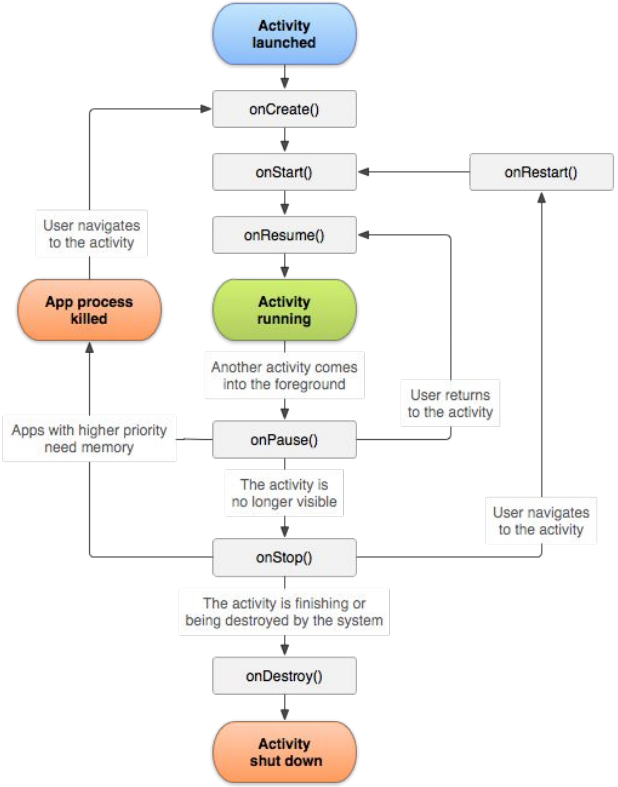
```
const val EXTRA_MESSAGE = "com.example.myfirstapp.MESSAGE"

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    /** Called when the user taps the Send button */
    fun sendMessage(view: View) {
        val editText = findViewById<EditText>(R.id.editText)
        val message = editText.text.toString()
        val intent = Intent(this, DisplayMessageActivity::class.java).apply {
            putExtra(EXTRA_MESSAGE, message)
        }
        startActivity(intent)
    }
}
```

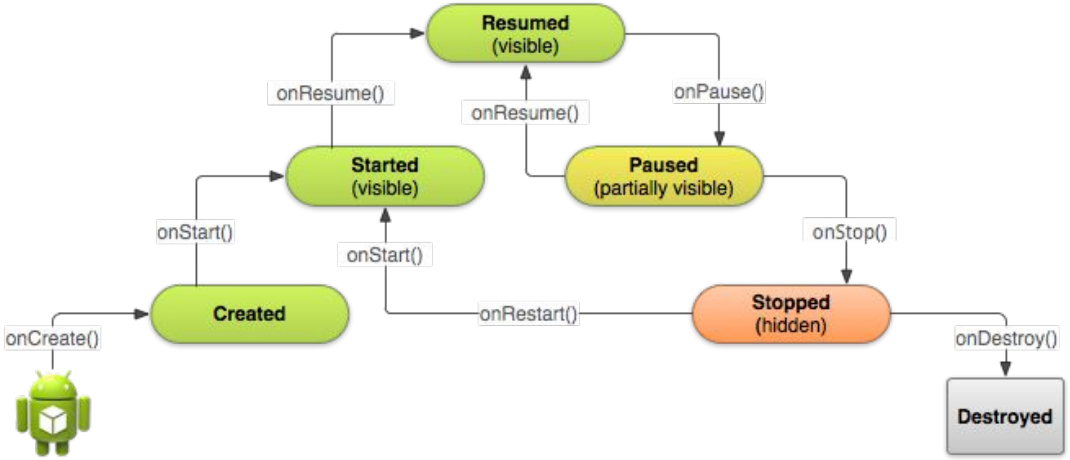
Starting an activity for result

```
//    start SecondActivity
fun startSecondActivity(view: View) {
   val intent = Intent(this, SecondActivity::class.java)
   startActivityForResult(intent, GET_NAME)
}

//   return from SecondActivity
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
   super.onActivityResult(requestCode, resultCode, data)
   if (requestCode == GET_NAME) {
       if (resultCode == Activity.RESULT_OK) {
           val name = data?.getStringExtra(SecondActivity.NAME)
           Toast.makeText(this, "Your name $name ", Toast.LENGTH_SHORT).show()
       }
   }
}
```

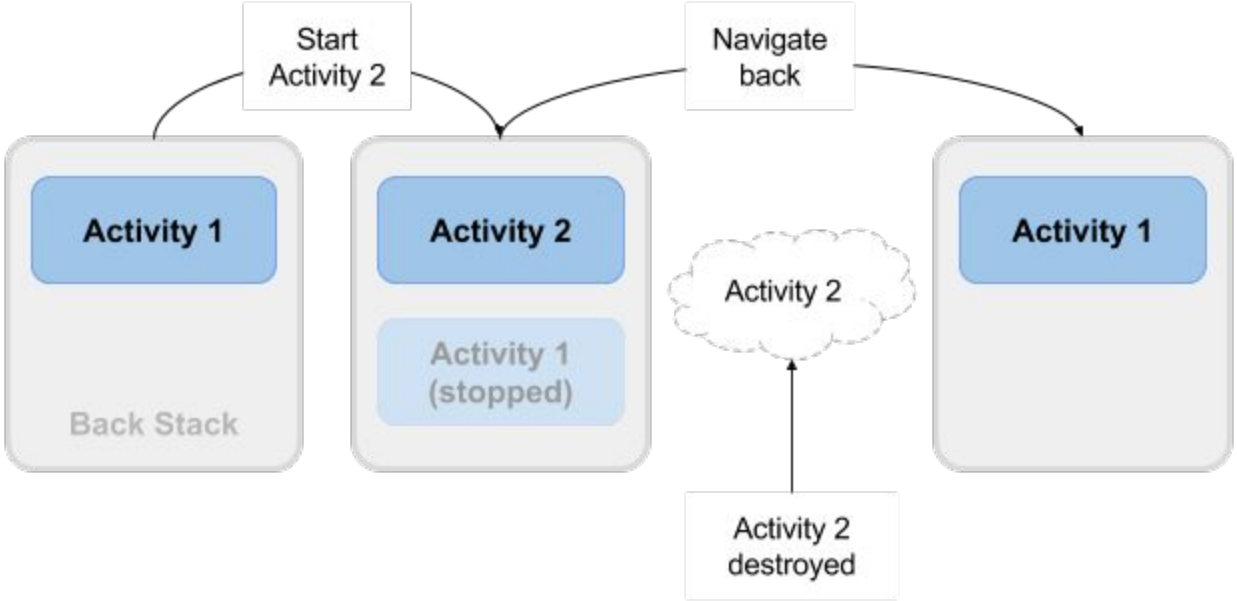


Activity lifecycle

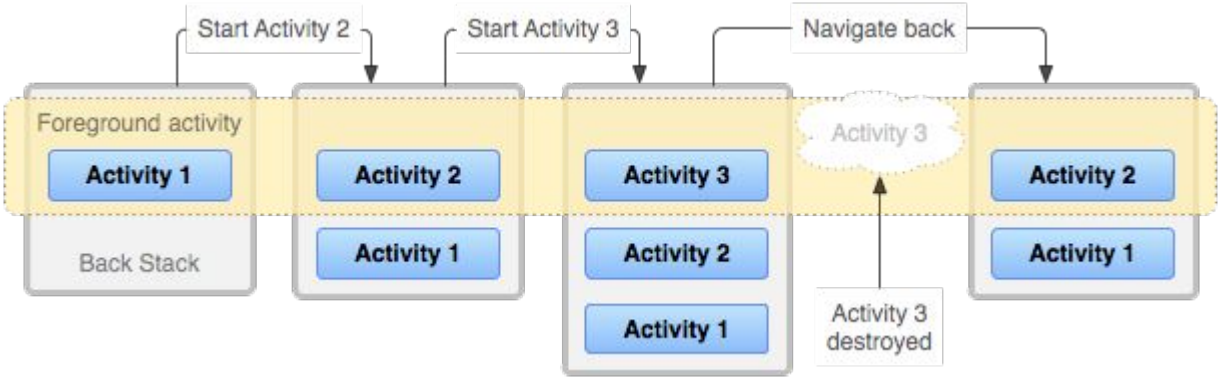




Activity states



Back stack


Start Activity 2
Start Activity 3
Navigate back
Foreground activity
Activity 1
Back Stack
Activity 2
Activity 1
Activity 3
Activity 2
Activity 1
Activity 3
Activity 3 destroyed
Activity 2
Activity 1

3. Layouts

Views

Every element of the screen is a view.

View class - base class for UI classes

A view has:

- a location (x, y)
- width
- height

The unit for location and dimensions is the **device independent pixel** (**dp**).

View groups

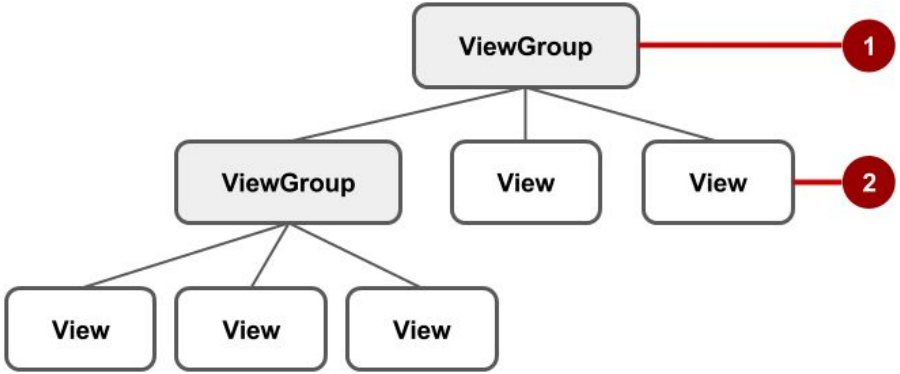
View group = Container for views

Common view groups:

- ScrollView: A group that contains one other child view and enables scrolling the child view.
- RecyclerView: A group that contains a list of other views or view groups and enables scrolling them by adding and removing views dynamically from the screen.

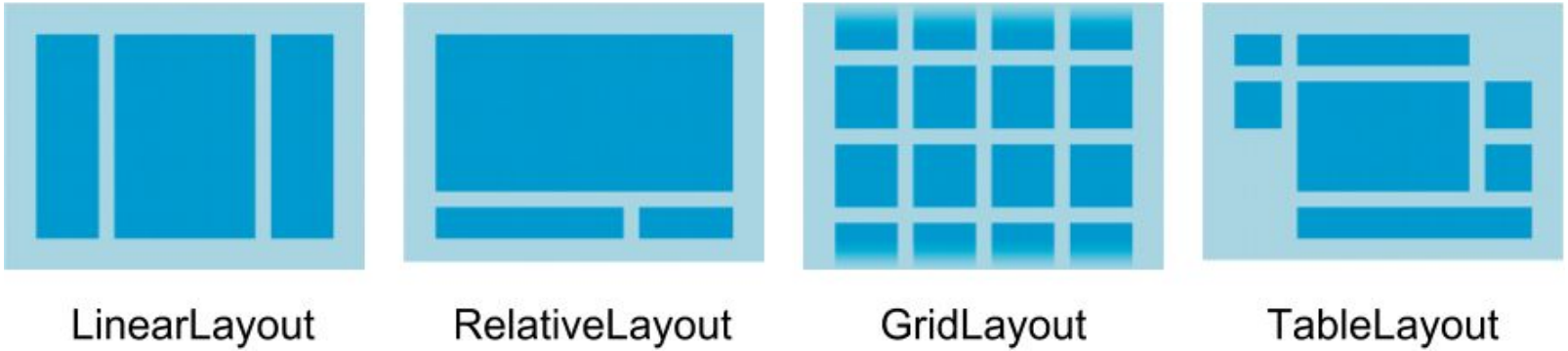
Layout for view groups

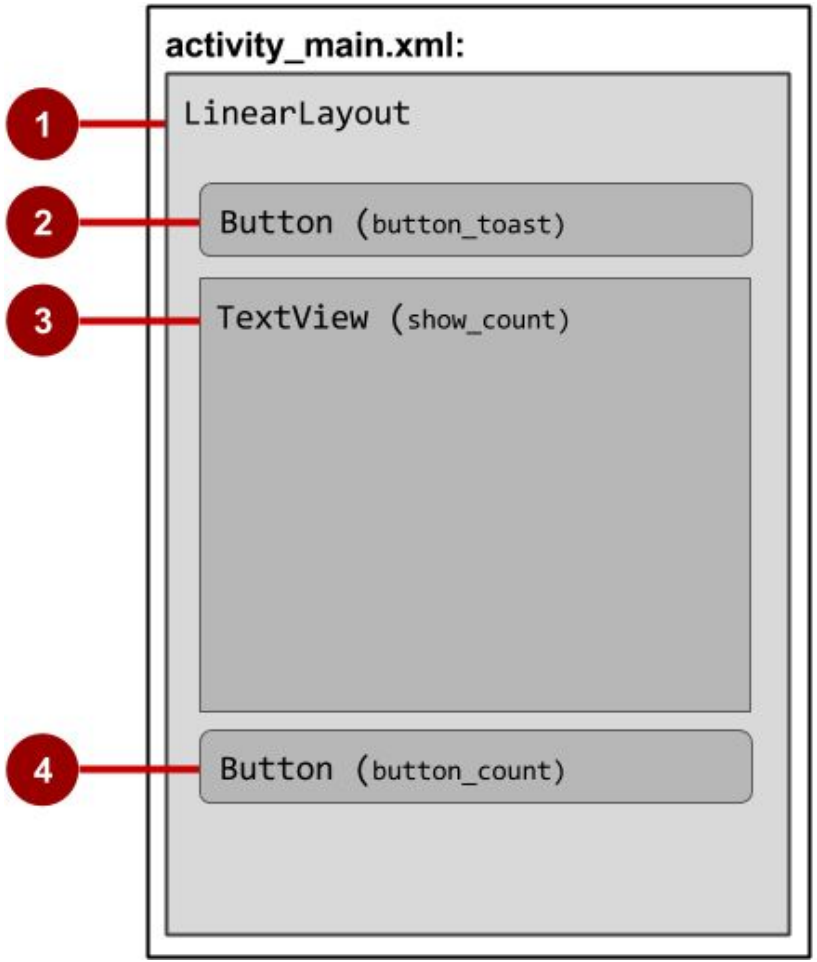
- The views for a screen are organized in a hierarchy.

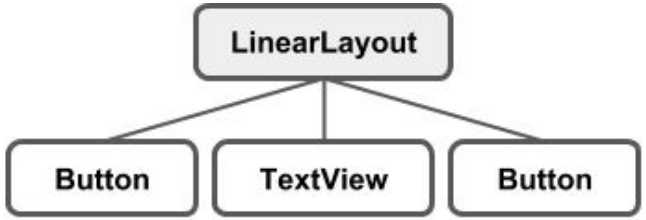


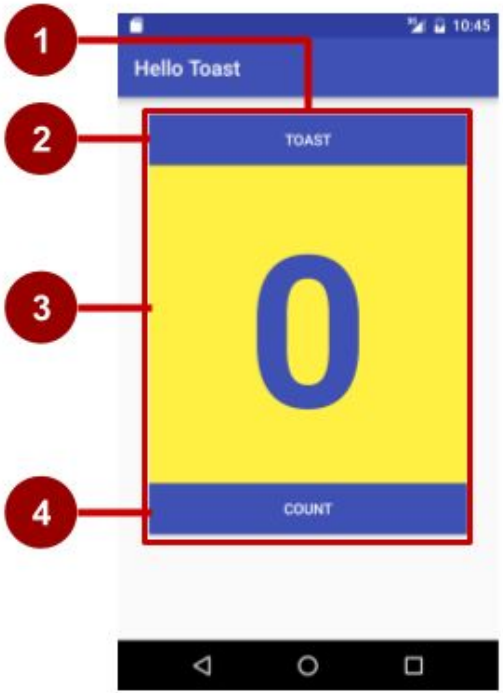
Common layouts

- Some view groups are designated as *layouts* because they organize child views in a specific way and are typically used as the root view group:
  - LinearLayout
  - ConstraintLayout




activity_main.xml:
1
LinearLayout
2
Button (button_toast)
3
TextView (show_count)
4
Button (button_count)


LinearLayout
Button
TextView
Button


1
2
3
4
10:45
Hello Toast
TOAST
0
COUNT

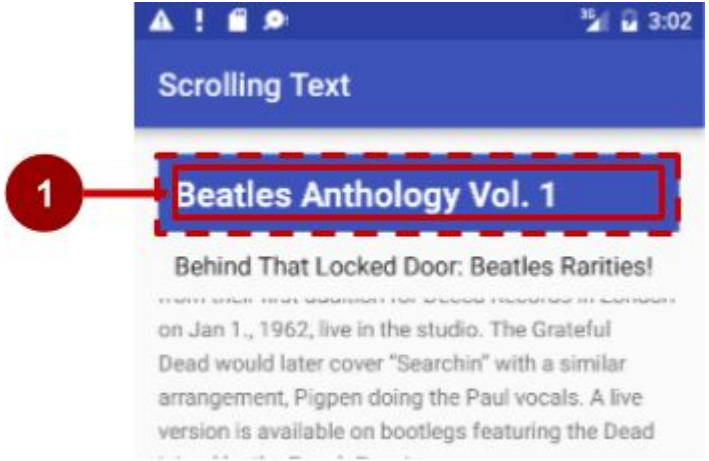
XML attributes (view properties)

```
<TextView

        android:id="@+id/show_count"

        android:layout_width="match_parent"

        android:layout_height="wrap_content"

        android:background="@color/myBackgroundColor"

        android:textStyle="bold"

        android:text="@string/count_initial_value" />
```

- **wrap_content** tells your view to size itself to the dimensions required by its content.
- **match_parent** tells your view to become as big as its parent view group will allow.

View properties

Padding is the space, measured in density-independent pixels, between the edges of the UI element and the element's content.



Resource files (1)

Separating static values from code → You can change the values without modifying the code.

Resource files are stored in folders located in the `res` folder when viewing the Project > Android pane. These folders include:

- `drawable`: For images and icons
- `layout`: For layout resource files
- `menu`: For menu items
- `mipmap`: For pre-calculated, optimized collections of app icons used by the Launcher
- `values`: For colors, dimensions, strings, and styles (theme attributes)

Event handling - XML

```
<Button

    android:id="@+id/button_toast"

    android:onClick="showToast"

    ...
```

```
class MainActivity : AppCompatActivity() {

        override fun onCreate(savedInstanceState: Bundle?) {

                super.onCreate(savedInstanceState)

        }


        private fun showToast(view: View) {
                ...
        }
}
```

Event handling - program code

```
<Button

    android:id="@+id/button_toast"

        ...
```

```
class MainActivity : AppCompatActivity() {

        lateinit var button: Button

        override fun onCreate(savedInstanceState: Bundle?) {

                super.onCreate(savedInstanceState)

                //...

                button = findViewById(R.id.button_toast)

                button.setOnClickListener { showToast(it) }


        }


        private fun showToast(view: View) {
                ...
        }
}
```

Android Architecture Components

Helps to build robust and maintainable apps

- **Data Binding**
- Lifecycles
- LiveData
- Navigation
- Paging
- Room
- ViewModel
- WorkManager

Data Binding

`findViewById()`

- get references to views
- **expensive -** Android traverses the view hierarchy (could be deep - production apps)
- Solutions:
  - Use `ButterKnife` (lightweight library to inject views into Android components)
  - Use **data binding** (part of Android Architecture Components)

Using Data Binding

1. Enable data binding in the android section of the **build.gradle** file:

```
buildFeatures {
    dataBinding true
}
```

2. Use `<layout>` as the root view in your **XML layout**.
3. Define a binding variable: `private lateinit var binding: ActivityMainBinding`
4. Create a binding object in **MainActivity**, replacing setContentView:

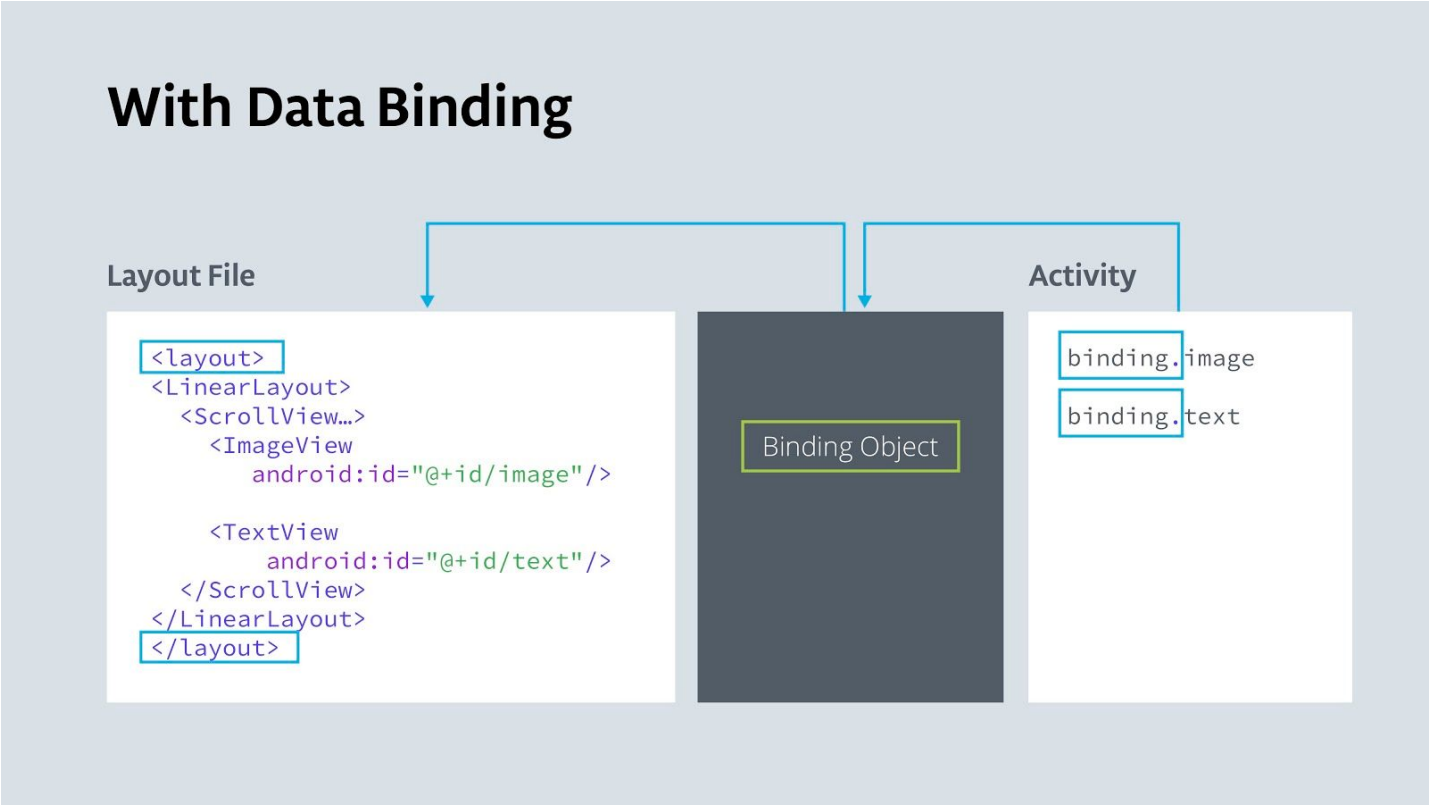
```
binding = DataBindingUtil.setContentView(this, R.layout.activity_main)
```

5. Replace calls to findViewById() with references to the view in the binding object. For example:

```
findViewById<Button>(R.id.done_button)  ⇒ binding.doneButton
```

(In the example, the name of the view is generated camel case from the view's id in the XML.)

Data Binding



Data Binding

Advantages	Disadvantages
● Code is shorter, easier to read, and easier to maintain than code that uses findViewById(). ● Fit to MVVM pattern ● The Android system only traverses the view hierarchy once to get each view, and it happens during app startup, not at runtime when the user is interacting with the app. ● You get type safety for accessing views. (*Type safety* means that the compiler validates types while compiling, and it throws an error if you try to assign the wrong type to a variable.) ● Increases testability	● Increases the build time (code generation!) ● Increases app size

4. Fragments

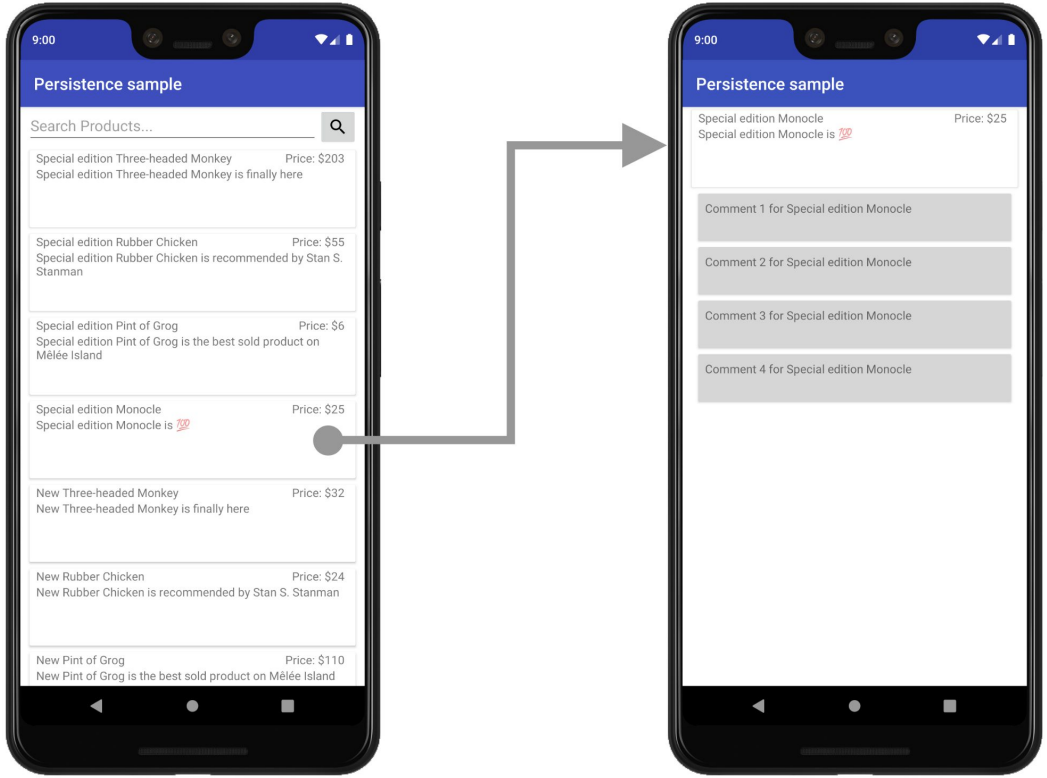
Fragment

A Fragment represents a behavior or a portion of user interface (UI) in an activity. You can combine multiple fragments in a single activity to build a multi-pane UI, and you can reuse a Fragment in multiple activities.

Think of a Fragment as a modular section of an activity, something like a "sub-activity" that you can also use in other activities:

- A Fragment has its own **lifecycle** and receives its own input events.
- You can **add** or **remove** a Fragment while the activity is running.
- A **Fragment** is defined in a **Kotlin class**.
- A **Fragment's UI** is defined in an **XML layout** file.

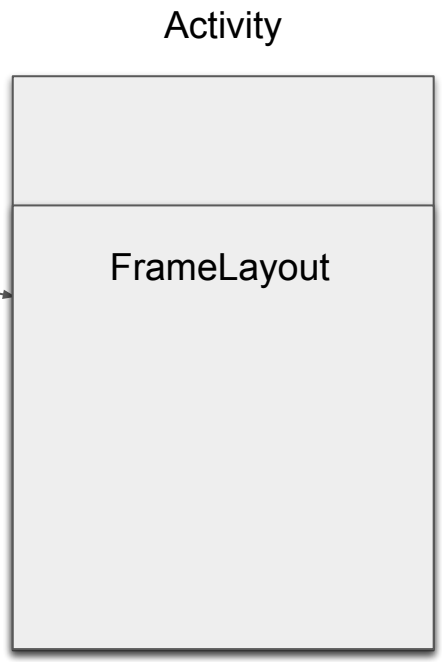
One Activity and two fragments



Layout for fragments

```
//activity_main.xml
<FrameLayout
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/fragment_container"
  android:layout_height="match_parent"
  android:layout_width="match_parent" />
```

```
//MainActivity
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   Log.d(TAG, "Bundle: " + savedInstanceState.toString())
   setContentView(R.layout.activity_main)

   if (savedInstanceState == null) {
       supportFragmentManager
           .beginTransaction()
           .add(R.id.fragment_container, StartFragment.newInstance(), fragmentIDS[0])
           .commit()
   }
}
```



Fragment

```
class StartFragment : Fragment() {

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

    // View initialization logic
   override fun onCreateView(
       inflater: LayoutInflater, container: ViewGroup?,
       savedInstanceState: Bundle?
       ): View? {
       // Inflate the layout for this fragment
       return inflater.inflate(R.layout.fragment_start, container, false)
   }
   companion object {

       @JvmStatic
       fun newInstance() = StartFragment()

   }
}
```

Replacing the fragment

```
button.setOnClickListener{
   this.activity?.supportFragmentManager
       ?.beginTransaction()
       ?.replace(R.id.fragment_container, QuestionFragment.newInstance(), fragmentIDS[1])
       ?.commit()

}
```

5. Navigation

Android Architecture Components

- Data Binding
- Lifecycles
- LiveData
- **Navigation**
- Paging
- Room
- ViewModel
- WorkManager

Navigation component

- Android Architecture Components
- Parts:
  - Navigation graph (resource)
  - NavHostFragment (layout)
  - NavController

Gradle scripts (1)

build.gradle (Project)

```
buildscript {

   ext.navigationVersion ='2.3.0'

   dependencies {
            classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$navigationVersion"
   }
}
```

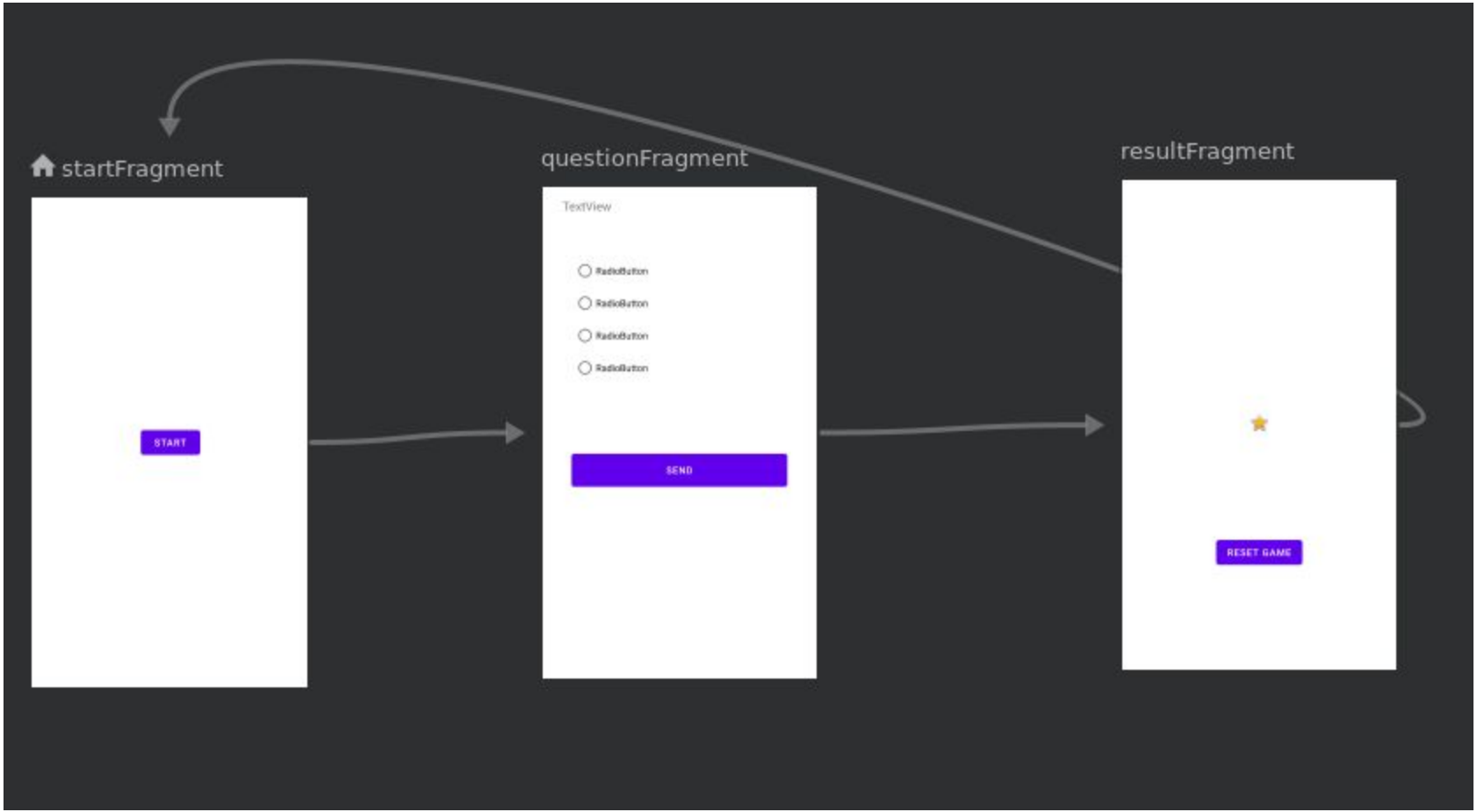
Gradle scripts (2)

build.gradle (Module)

```
apply plugin: 'androidx.navigation.safeargs.kotlin'
```

Responsible for the generation of navigation classes:

<FooFragment>Directions


startFragment
START
questionFragment
TextView
RadioButton
RadioButton
RadioButton
RadioButton
SEND
resultFragment
RESET GAME

Navigation graph (Resource)

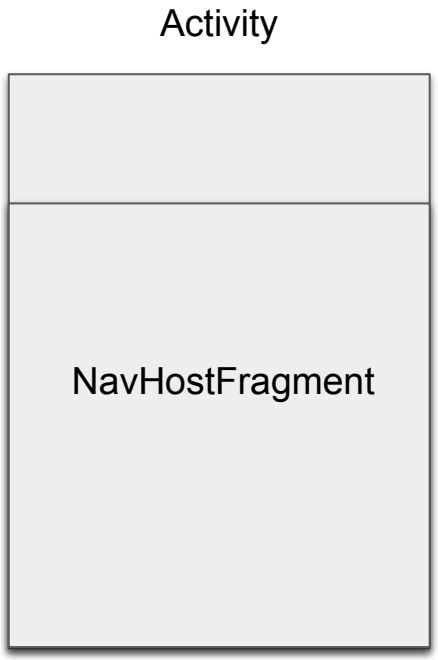
res/navigation/questions_nav_graph.xml

```
<navigation 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:id="@+id/questions_nav_graph"
   app:startDestination="@id/startFragment">

   <fragment
       android:id="@+id/startFragment"
       android:name="com.example.testnavcontroller.fragments.StartFragment"
       android:label="fragment_start"
       tools:layout="@layout/fragment_start" >
       <action
           android:id="@+id/action_startFragment_to_questionFragment"
           app:destination="@id/questionFragment" />
   </fragment>
   ...
</navigation>
```

NavHostFragment (Layout)

MainActivity - activity_main.xml

```
<fragment

    android:id="@+id/myNavHostFragment"

    android:name="androidx.navigation.fragment.NavHostFragment"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    app:navGraph="@navigation/questions_nav_graph"

    app:defaultNavHost="true"

    />
```



NavController

StartFragment:

```
button.setOnClickListener{
        this.findNavController().navigate(
               StartFragmentDirections.actionStartFragmentToQuestionFragment())
}
```

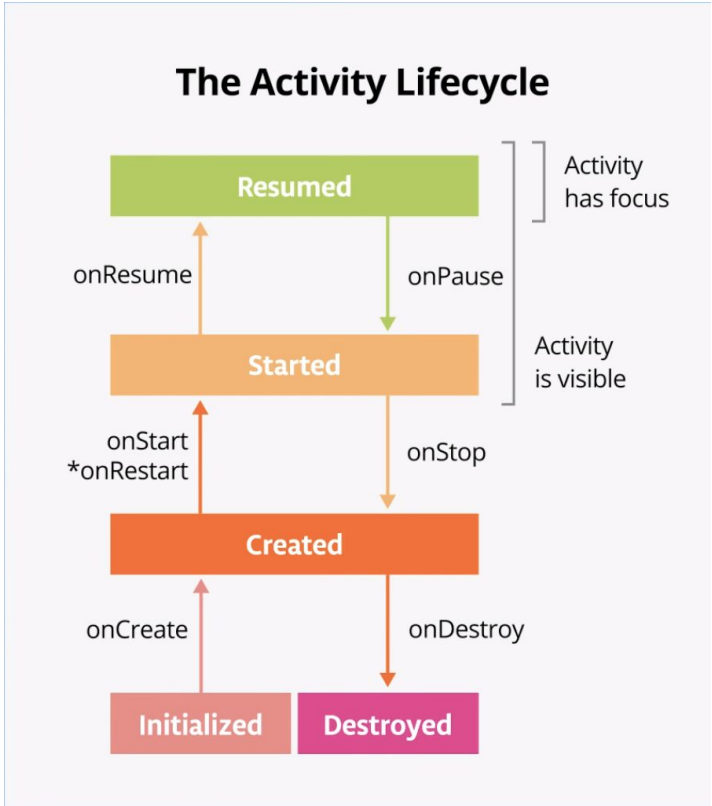
Generated Classes

DataBinding in Fragments

```
class ResultFragment : Fragment() {

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

   override fun onCreateView(
       inflater: LayoutInflater, container: ViewGroup?,
       savedInstanceState: Bundle?
   ): View? {
       // Inflate the layout for this fragment  using data binding
       val binding: FragmentResultBinding = DataBindingUtil.inflate(
           inflater, R.layout.fragment_result, container, false)

       binding.resetGameButton.setOnClickListener { view: View ->
           view.findNavController().navigate(ResultFragmentDirections.actionResultFragmentToStartFragment())
       }
       return binding.root
   }

}
```

6. Activity and fragments lifecycle

The Activity Lifecycle



Demo app: TestLifecycle

```
override fun onStart() {
   super.onStart()
   Timber.i("onStart Called")
}

override fun onRestart() {
   super.onRestart()
   Timber.i("onRestart Called")
}

override fun onResume() {
   super.onResume()
   Timber.i("onResume Called")
}
```

QA - Quiz application

1. Where to store the number of completed quizzes?
   a. Companion object of the ResultFragment
   b. Companion object of the MainActivity
   c. Global application scope
   d. SharedPreferences

2. Where to increment the number of completed quizzes?
   a. onCreateView method of the ResultFragment
   b. QuestionFragment - i. e. process answer

Logging with Timber

`Timber` has several advantages over the built-in Android `Log` class:

- **Generates the log tag** for you based on the **class name**.
- Helps you **avoid showing logs** in a **release version** of your Android app.
- Allows for **integration with crash-reporting libraries**.

Using Timber

Step 1: Add Timber to Gradle (Module: app)

```
dependencies {
    ...
    implementation 'com.jakewharton.timber:timber:4.7.1'
}
```

Using Timber

Step 2: Create an Application class and initialize Timber

Application is a base class that contains global application state for your entire app.

```
class MyApplication : Application() {
   override fun onCreate() {
       super.onCreate()
        Timber.plant(Timber.DebugTree())

   }
}
```

AndroidManifest.xml

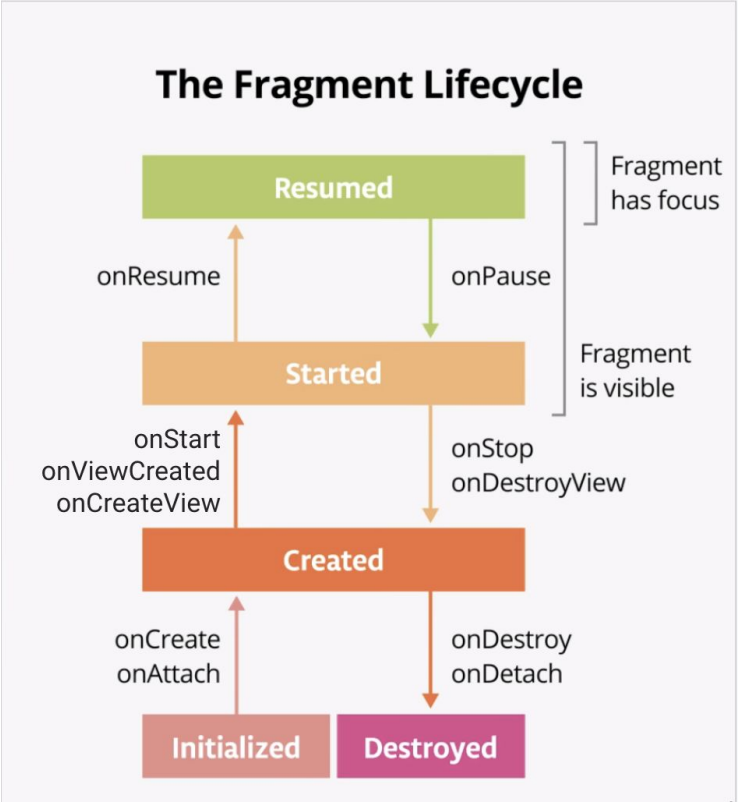
```
<application
      android:name=".MyApplication"
  ...
```

Using Timber

Step 3: Add Timber log statements

```
override fun onStart() {
    super.onStart()
    Timber.i("onStart Called")
}
```

The Fragment Lifecycle



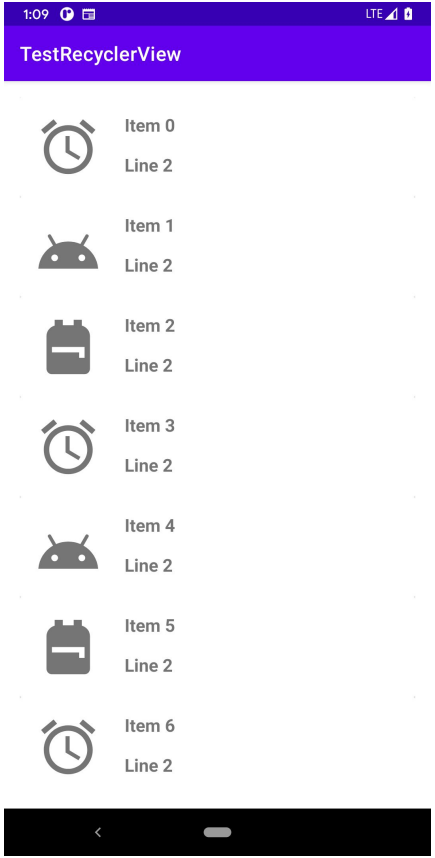
7. RecyclerView

7.1. Drawing UI

When to use

Your app have to display

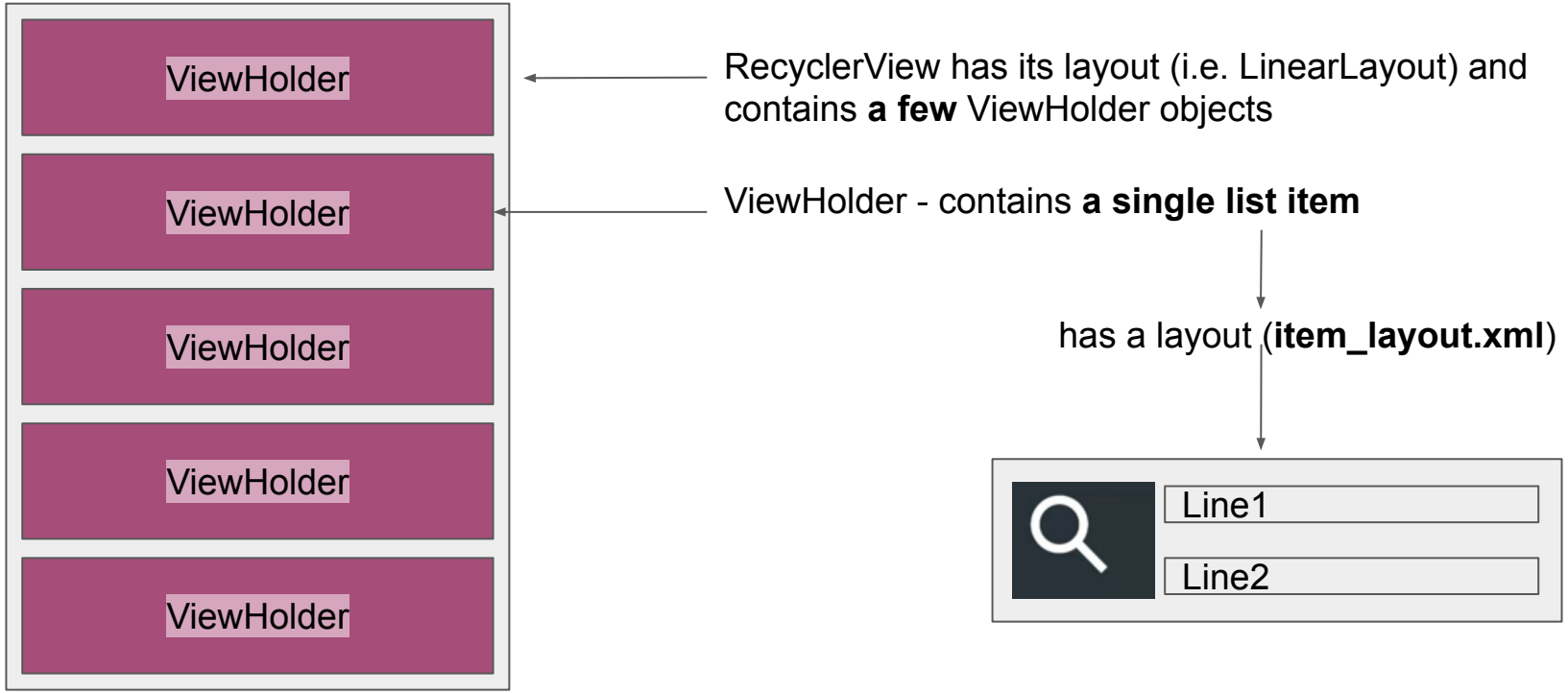
- a **scrolling list** of elements
- based on a **large data set** (or data that frequently changes)



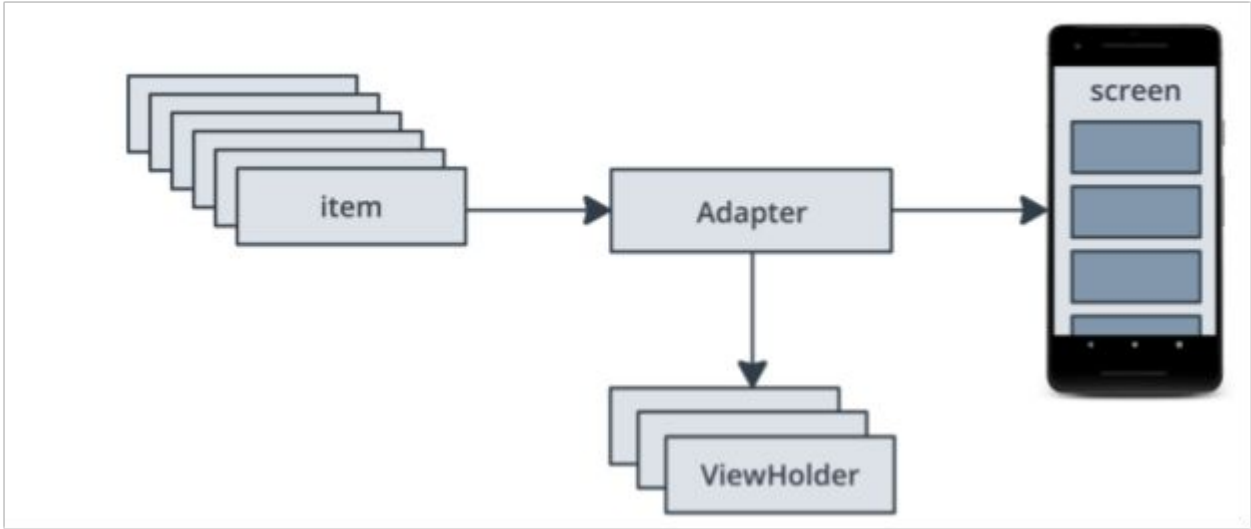
RecyclerView overview

- The `RecyclerView` fills itself with views provided by a *layout manager*.
- The views in the list are represented by ***view holder*** objects. These objects are instances of a class you define by extending `RecyclerView.ViewHolder`. Each view holder is in charge of displaying a **single item with a view**.
- The `RecyclerView` creates only as many view holders as are needed to display the on-screen portion of the dynamic content, plus a few extra. As the user scrolls through the list, the `RecyclerView` takes the off-screen views and rebinds them to the data which is scrolling onto the screen.
- The view holder objects are managed by an ***adapter***, which you create by extending `RecyclerView.Adapter`. The adapter creates view holders as needed. The adapter also binds the view holders to their data. It does this by assigning the view holder to a position, and calling the adapter's `onBindViewHolder()` method. That method uses the view holder's position to determine what the contents should be, based on its list position.

RecyclerView and ViewHolder



RecyclerView.Adapter



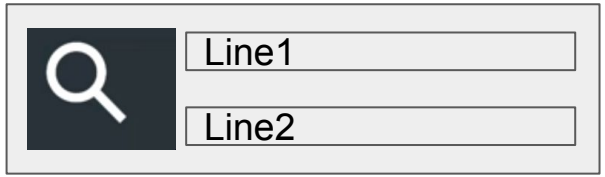
Using RecyclerView - gradle.build (module)

```
dependencies {
    implementation "androidx.recyclerview:recyclerview:1.1.0"
    implementation "androidx.cardview:cardview:1.0.0"
}
```

Used for list item

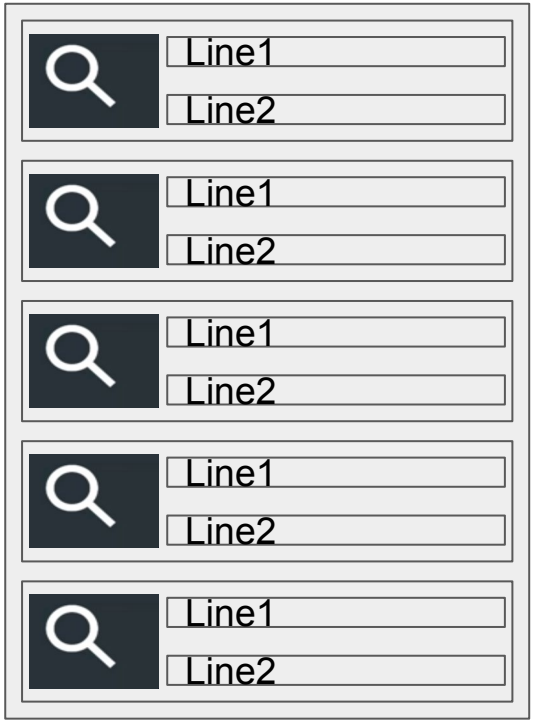
Using RecyclerView - /res/layout/item_layout.xml

```
<androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <ImageView
            android:id="@+id/image_view"
            ...
            />
        <TextView
            android:id="@+id/text_view_1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            ...
             />
        <TextView
            android:id="@+id/text_view_2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            ...
             />
    </androidx.constraintlayout.widget.ConstraintLayout>
```



Using RecyclerView - /res/layout/activity_main.xml

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recycler_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    ...
    tools:itemCount="10"
    tools:listitem="@layout/item_layout" />
```



Item - Kotlin data class

```
data class Item (val imageResource:Int,
                 val text1: String,
                 var text2: String)
```

RecyclerView.Adapter

```
class DataAdapter(private val list: List<Item>) : RecyclerView.Adapter<DataAdapter.DataViewHolder>() {
   // 1. user defined ViewHolder type
   class DataViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {}

   // 2. Called only a few times = number of items on screen + a few more ones
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): DataViewHolder {}

   // 3. Called many times, when we scroll the list
   override fun onBindViewHolder(holder: DataViewHolder, position: Int) {}

   // 4.
   override fun getItemCount() = list.size
}
```

1. RecyclerView.ViewHolder

```
class DataAdapter(private val list: List<Item>) :
      RecyclerView.Adapter<DataAdapter.DataViewHolder>() {

      class DataViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
          val imageView: ImageView = itemView.findViewById(R.id.image_view)
          val textView1: TextView = itemView.findViewById(R.id.text_view_1)
          val textView2: TextView = itemView.findViewById(R.id.text_view_2)
      }

}
```

2. RecyclerView.Adapter - onCreateViewHolder

```
class DataAdapter(private val list: List<Item>) : RecyclerView.Adapter<DataAdapter.DataViewHolder>() {

    // Called only a few times
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): DataViewHolder {
        val itemView =
         LayoutInflater.from(parent.context).inflate(R.layout.item_layout, parent, false)
        return DataViewHolder(itemView)
    }

}
```

3. RecyclerView.Adapter - onBindViewHolder

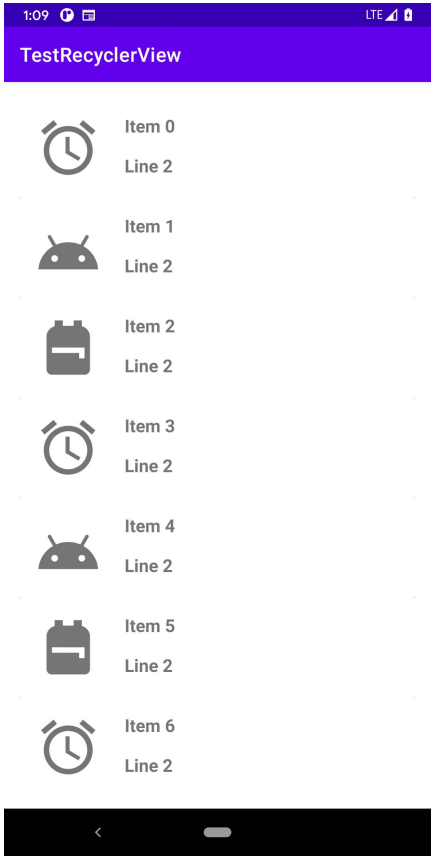
```
class DataAdapter(private val list: List<Item>) : RecyclerView.Adapter<DataAdapter.DataViewHolder>() {

    // Called many times, when we scroll the list
    override fun onBindViewHolder(holder: DataViewHolder, position: Int) {
        val currentItem = list[position]
        holder.imageView.setImageResource(currentItem.imageResource)
        holder.textView1.text = currentItem.text1
        holder.textView2.text = currentItem.text2
    }


}
```

Data generation →List<Item>

```
private fun generateDummyList(size: Int):  List<Item>{
   val list = ArrayList<Item>()
   for (i in 0  until size ){
       val drawable = when (i%3){
           0 -> R.drawable. ic_alarm
           1 -> R.drawable. ic_android
           else -> R.drawable. ic_backpack
       }
       val item = Item(drawable, "Item $i", "Line 2")
       list += item
   }
   return list
}
```

Connect the RecyclerView to the Adapter

```
val list = generateDummyList(500)

val recycler_view : RecyclerView = findViewById(R.id.recycler_view)

recycler_view.adapter = DataAdapter(list)

recycler_view.layoutManager = LinearLayoutManager(this )

recycler_view.setHasFixedSize(true)
```

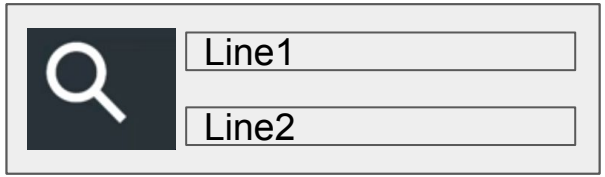


7. RecyclerView

7.2. Handling interactions

Handle click events

```
class DataAdapter(private val list: List<Item>) : RecyclerView.Adapter<DataAdapter.DataViewHolder>() {
   // 1. user defined ViewHolder type
   class DataViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {}

   // 2. Called only a few times = number of items on screen + a few more ones
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): DataViewHolder {}

   // 3. Called many times, when we scroll the list
   override fun onBindViewHolder(holder: DataViewHolder, position: Int) {}

   // 4.
   override fun getItemCount() = list.size
}
```



Where to handle events?

1. `DataAdapter.`**`DataViewHolder - constructor`**
2. `DataAdapter.`**`onCreateViewHolder`**
3. `DataAdapter.`**`onBindViewHolder`**

Where to handle events?

1. DataAdapter.**DataViewHolder - constructor**
2. DataAdapter.**onCreateViewHolder**
3. ~~DataAdapter.**onBindViewHolder**~~

Not efficient

Event handling - Best practice

Adapter tasks:

- creates view holders
- binds view holders to their data

Controller (Activity or Fragment) task:

- event handling

Solution - step 1 (a)

```
class DataAdapter(...){
  inner class DataViewHolder(itemView: View) :
                                        RecyclerView.ViewHolder(itemView),
                                        View.OnClickListener {
    val imageView: ImageView = itemView.findViewById(R.id.image_view)
    val textView1: TextView = itemView.findViewById(R.id.text_view_1)
    val textView2: TextView = itemView.findViewById(R.id.text_view_2)

    // Constructor!!!
    init{
        itemView.setOnClickListener(this)
    }
}
```

Solution - step 1 (b)

```
class DataAdapter(...){
  inner class DataViewHolder(itemView: View) :
                                        RecyclerView.ViewHolder(itemView),
                                        View.OnClickListener {

    //...
    init{
          itemView.setOnClickListener(this)
    }
    override fun onClick(v: View?) {
          //...
    }
}
```

Solution - step 2

```
class DataAdapter(...){

    //...
    interface OnItemClickListener{
        fun onItemClick(position: Int)
    }
}
```

Will be implemented by the controller (Activity or Fragment)

Solution - step 3

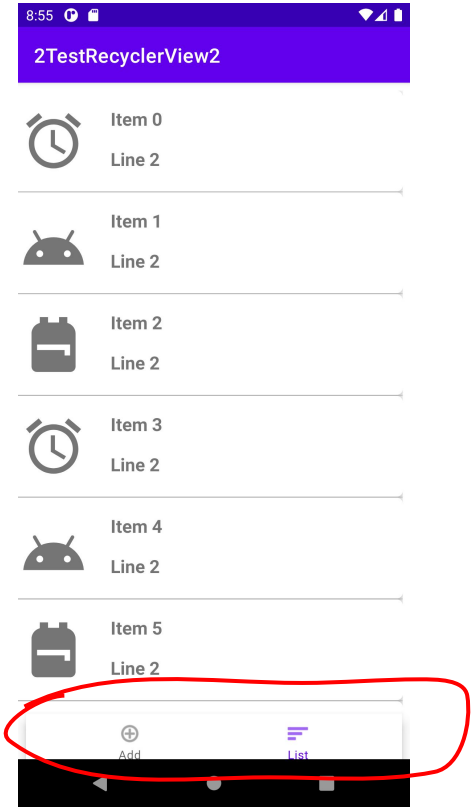
```
class ListFragment : Fragment(), DataAdapter.OnItemClickListener {

    override fun onItemClick(position: Int) {

            val clickedItem : Item = list[position]

            clickedItem.text2 = "Clicked"

            adapter.notifyItemChanged(position)

    }

}
```

Solution - step 4 - Delegation!!!

```
class DataAdapter(
    private val list: List<Item>,
    private val listener: OnItemClickListener
) : RecyclerView.Adapter<DataAdapter.DataViewHolder>() {
    inner class DataViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView),
                                                 View.OnClickListener {
        init{
            itemView.setOnClickListener(this)
        }

        override fun onClick(v: View?) {
            val position: Int = adapterPosition
            if( position != RecyclerView.NO_POSITION) {
                listener.onItemClick(position)
            }
        }
    }
}
```

8. Bottom Navigation Bar

BottomNavigationView



Steps

1. Gradle settings
2. Design navigation (Fragments + `list_navigation.xml`)
3. Design menu (`bottom_nav_menu.xml`)
4. `Design screen (activity_main.xml)`
5. MainActivity

Step 1 Dependencies

build.gradle (Project)

```
buildscript {
   ext.kotlin_version = "1.4.10"
   ext.navigationVersion='2.3.0'
}
```

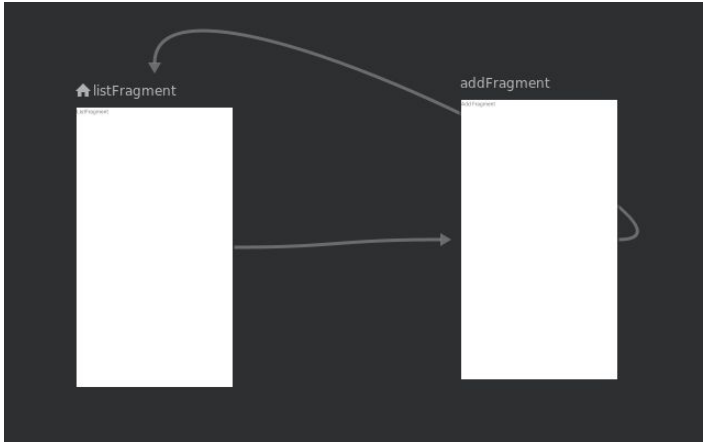
build.gradle (Module)

```
android{
   //...
   buildFeatures{
       dataBinding true
   }
}
dependencies{
      //...
      implementation 'androidx.navigation:navigation-fragment-ktx:2.2.2'
      implementation 'androidx.navigation:navigation-ui-ktx:2.2.2'
}
```

Step 2. Design Navigation

/res/layout/list_navigation.xml

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    //...
    android:id="@+id/list_navigation"
    app:startDestination="@id/listFragment">

    <fragment
        android:id="@+id/listFragment"
        android:name="com.example.testrecyclerview2.fragments.ListFragment"
        //...
    </fragment>

    <fragment
        android:id="@+id/addFragment"
        android:name="com.example.testrecyclerview2.fragments.AddFragment"
        //...
    </fragment>
</navigation>
```



Step 3. Bottom navigation menu

/res/menu/bottom_nav_menu.xml

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto">
   <item
       android:id="@+id/addFragment"
       android:icon="@android:drawable/ic_menu_add"
       android:orderInCategory="1"
       android:title="Add"
       app:showAsAction="always" />
   <item
       android:id="@+id/listFragment"
       android:icon="@android:drawable/ic_menu_sort_by_size"
       android:orderInCategory="2"
       android:title="List"
       app:showAsAction="always" />

</menu>
```

Navigation destination

Navigation destination

Step 4. Design activity layout + data binding

/res/layout/activity_main.xml

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    <androidx.constraintlayout.widget.ConstraintLayout
        <fragment
            android:id="@+id/myNavHostFragment"
            android:name="androidx.navigation.fragment.NavHostFragment"
            //...
        </fragment>
        <com.google.android.material.bottomnavigation.BottomNavigationView
            android:id="@+id/bottom_nav_view"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            //...
            app:menu="@menu/bottom_nav_menu">
        </com.google.android.material.bottomnavigation.BottomNavigationView>
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Data binding

Menu resource file

Step 5. Set up bottom navigation

MainActivity

```
class MainActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       // inflate layout
       val binding = DataBindingUtil.setContentView<ActivityMainBinding>(this, R.layout.activity_main)

       //  set up  BottomNavigation
       val navController = findNavController(R.id.myNavHostFragment)
       val bottomNav = binding.bottomNavView
       bottomNav?.setupWithNavController(navController)
   }
}
```

```
<fragment
    android:id="@+id/myNavHostFragment"
```

```
<com.google.android.material.bottomnavigation.BottomNavigationView
        android:id="@+id/bottom_nav_view"
```

9. ViewModel

Sharing data between fragments

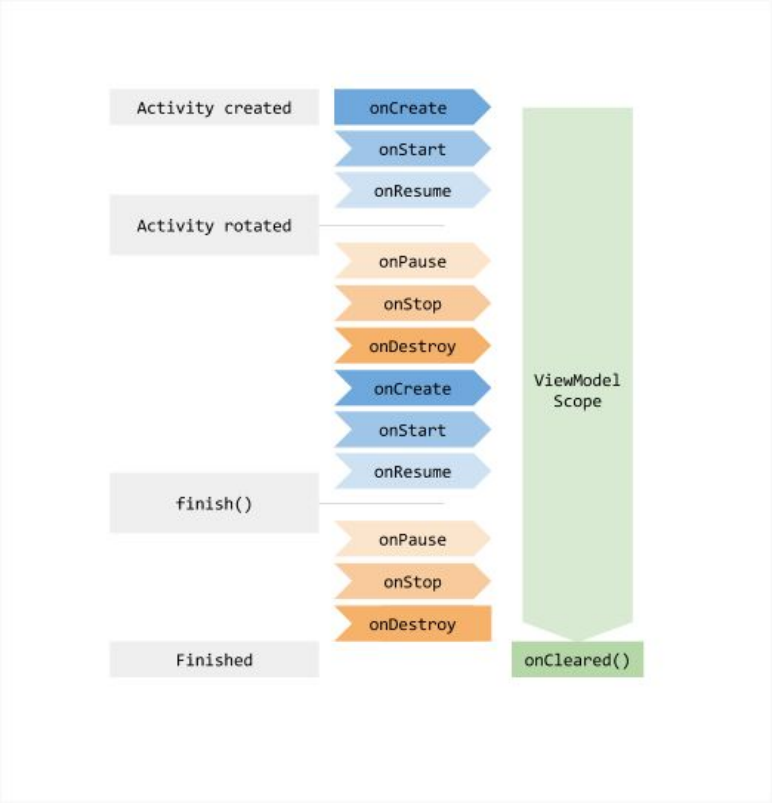
Android Architecture Components

- Data Binding
- Lifecycles
- LiveData
- Navigation
- Paging
- Room
- **ViewModel**
- WorkManager

ViewModel

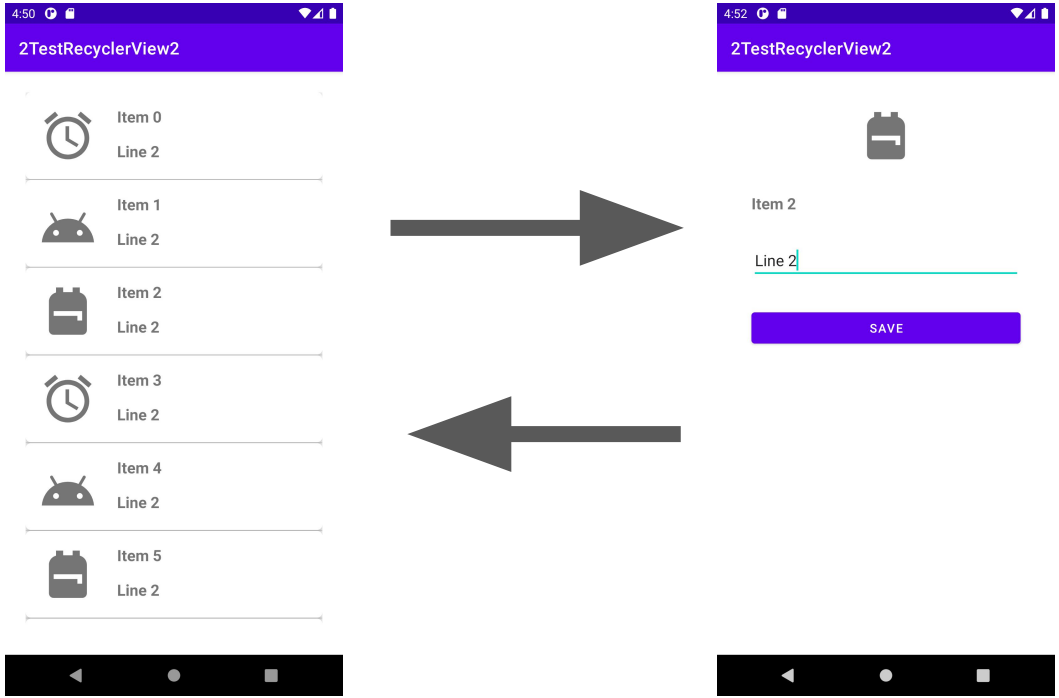
- The ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way. The ViewModel class allows data to survive configuration changes such as screen rotations.

ViewModel lifecycle

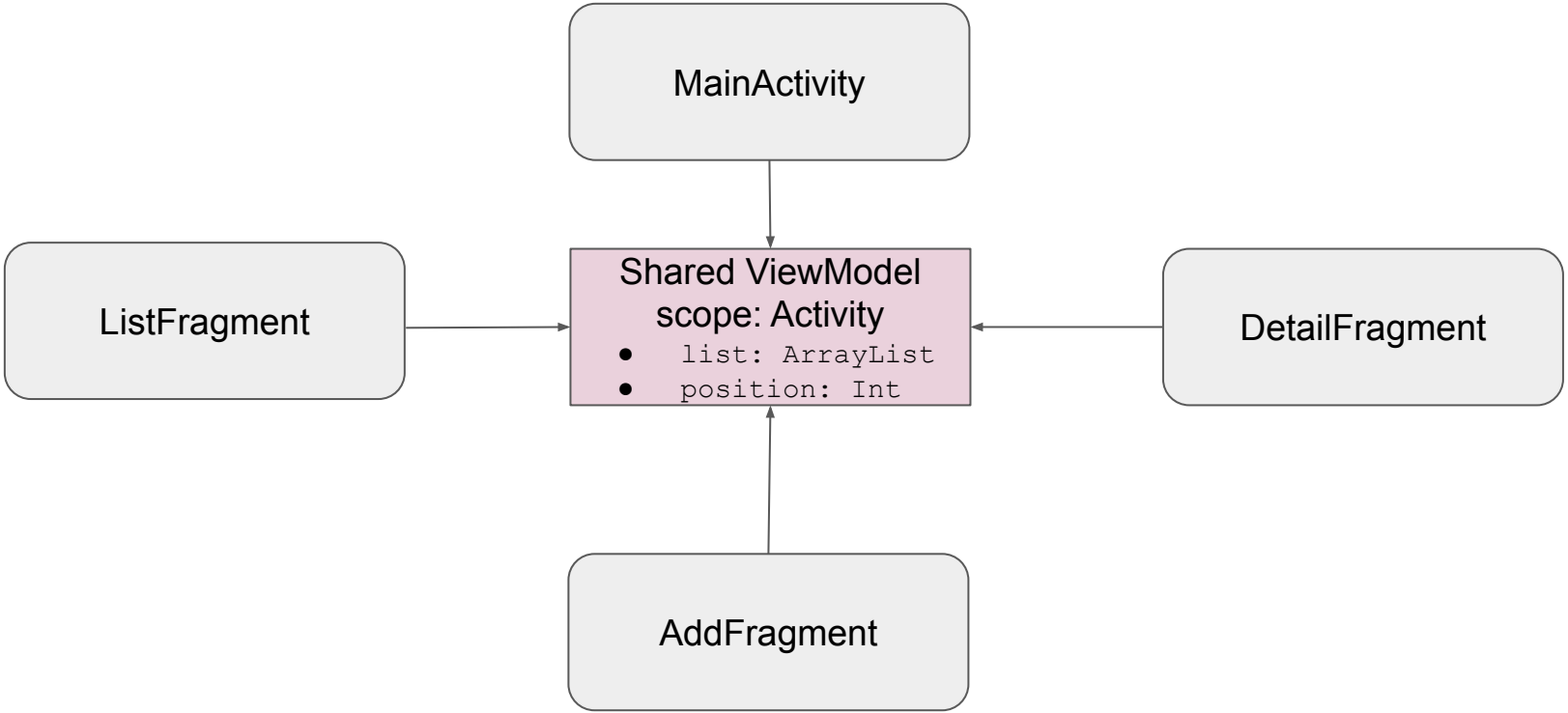


ListFragment → DetailFragment

https://developer.android.com/topic/libraries/architecture/viewmodel



ViewModel - sharing data between fragments



SharedViewModel

```
class SharedViewModel : ViewModel() {
   var list: ArrayList<Item> = generateDummyList(10)
   var position: Int = 0

   private fun generateDummyList(size: Int): ArrayList<Item> {
       val list = ArrayList<Item>()
       for (i in 0 until size) {
           val drawable = when (i % 3) {
               0 -> R.drawable.ic_alarm
               1 -> R.drawable.ic_android
               else -> R.drawable.ic_backpack
           }
           val item = Item(drawable, "Item $i", "Line 2")
           list += item
       }
       return list
   }

}
```

ListFragment

```
class ListFragment : Fragment(), DataAdapter.OnItemClickListener{
   lateinit var adapter: DataAdapter
   lateinit var viewModel: SharedViewModel

   override fun onCreateView(
       inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? {
       val layout = inflater.inflate(R.layout.fragment_list, container, false)
       // Access activity's ViewModel
       viewModel = activity?.run{
          ViewModelProvider(this).get(SharedViewModel::class.java)
       }!!
       //...
   }

   override fun onItemClick(position: Int) {
       viewModel.position = position
       findNavController().navigate(ListFragmentDirections.actionListFragmentToDetailFragment())
   }
}
```

DetailFragment

```
class DetailFragment : Fragment() {
   lateinit var viewModel: SharedViewModel

   override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,savedInstanceState: Bundle?): View? {
       // Inflate the layout for this fragment
       val layout = inflater.inflate(R.layout.fragment_detail, container, false)

       // Access activity's ViewModel
       viewModel = activity?.run{
           ViewModelProvider(this).get(SharedViewModel::class.java)
       }!!

       val imageResource: Int = viewModel.list[viewModel.position].imageResource
       val line1 = viewModel.list[viewModel.position].text1
       val line2 = viewModel.list[viewModel.position].text2
       // ..
       return layout
   }
}
```

10. LiveData

LiveData + ViewModel

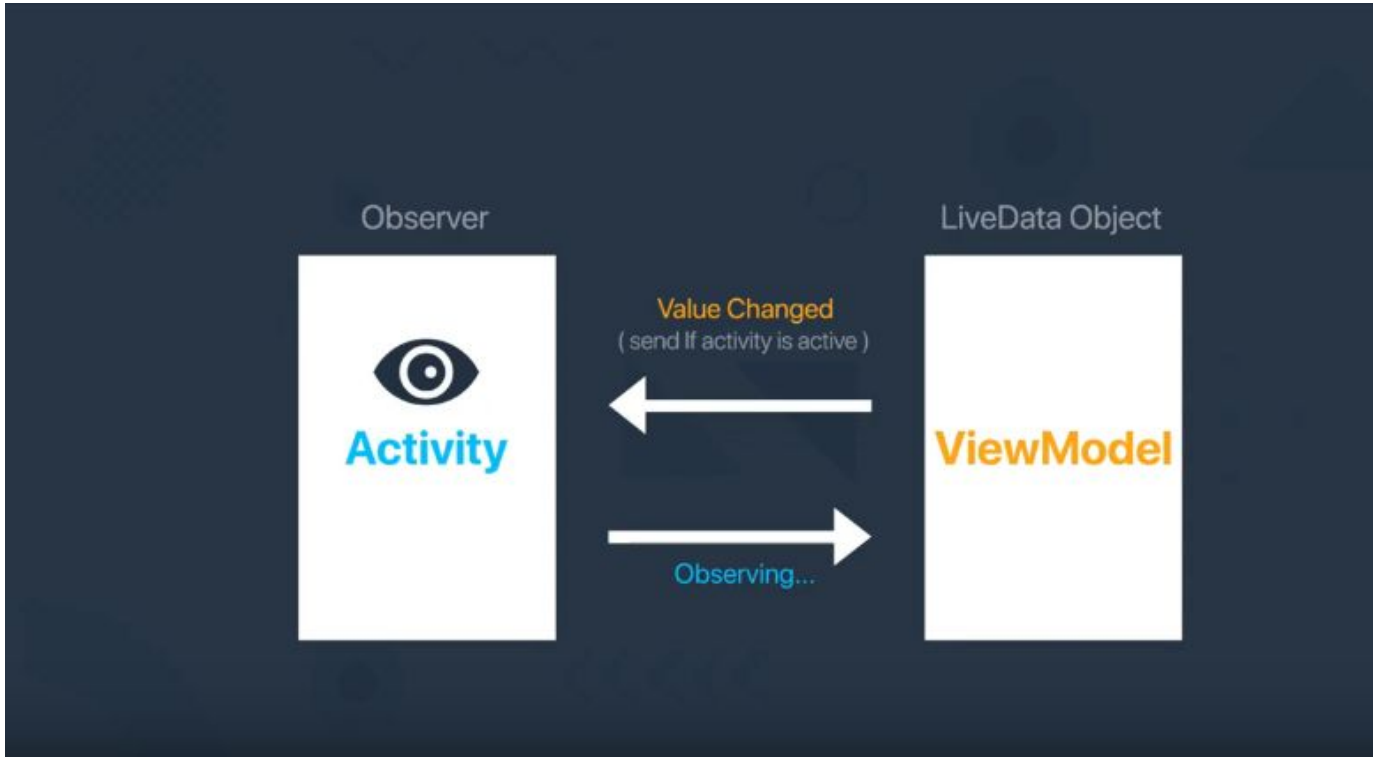
Android Architecture Components

- Data Binding
- Lifecycles
- **LiveData**
- Navigation
- Paging
- Room
- ViewModel
- WorkManager

LiveData

- Simple
- Lifecycle-aware
- **Observable**
- Data holder

LiveData - Observer Design Pattern



https://www.youtube.com/watch?v=suC0OM5gGAA

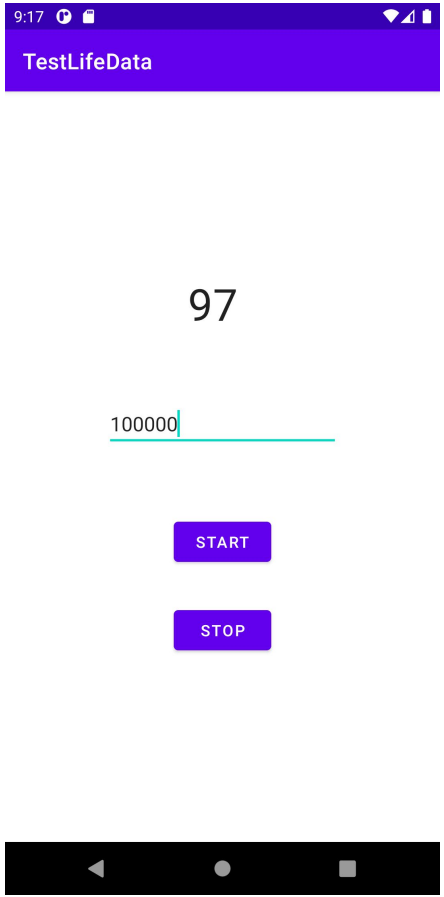
LiveData - Observer Design Pattern

https://www.youtube.com/watch?v=suC0OM5gGAA

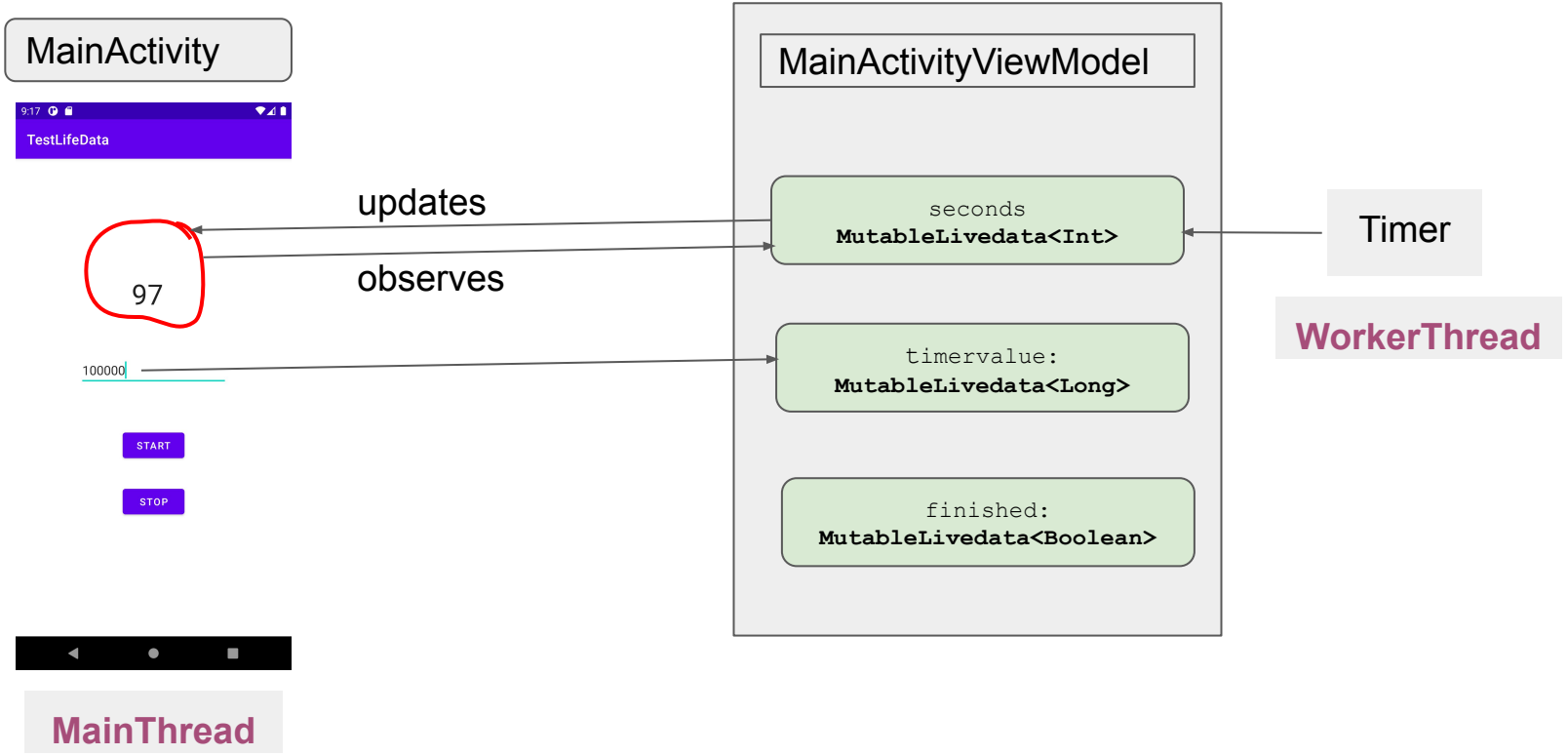
LiveData

- sends updates
  - only to active observers
  - only when data changes

https://www.youtube.com/watch?v=suC0OM5gGAA



Activity - ViewModel - LiveData



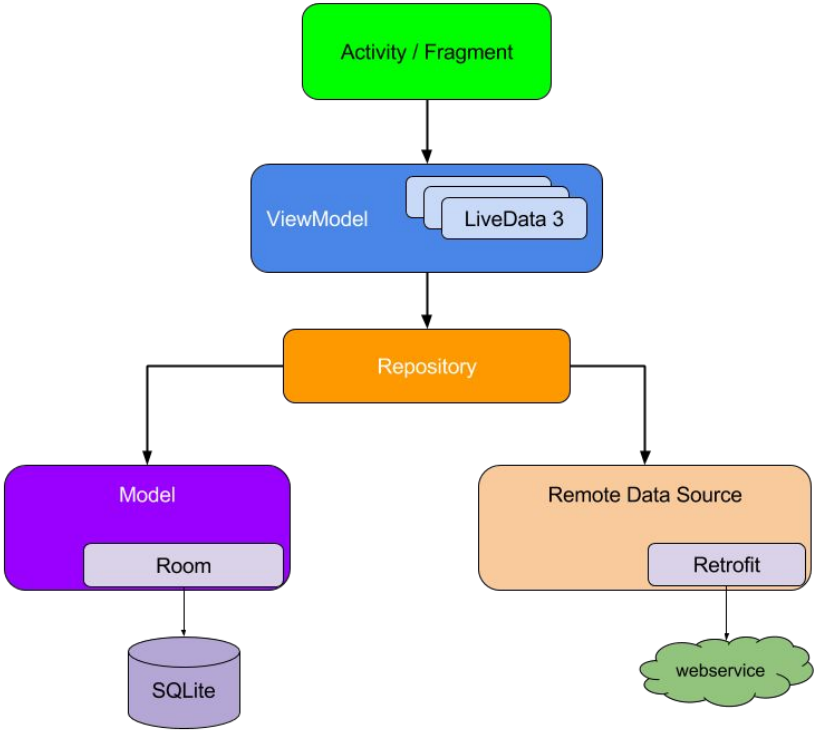
11. Room

Persistence

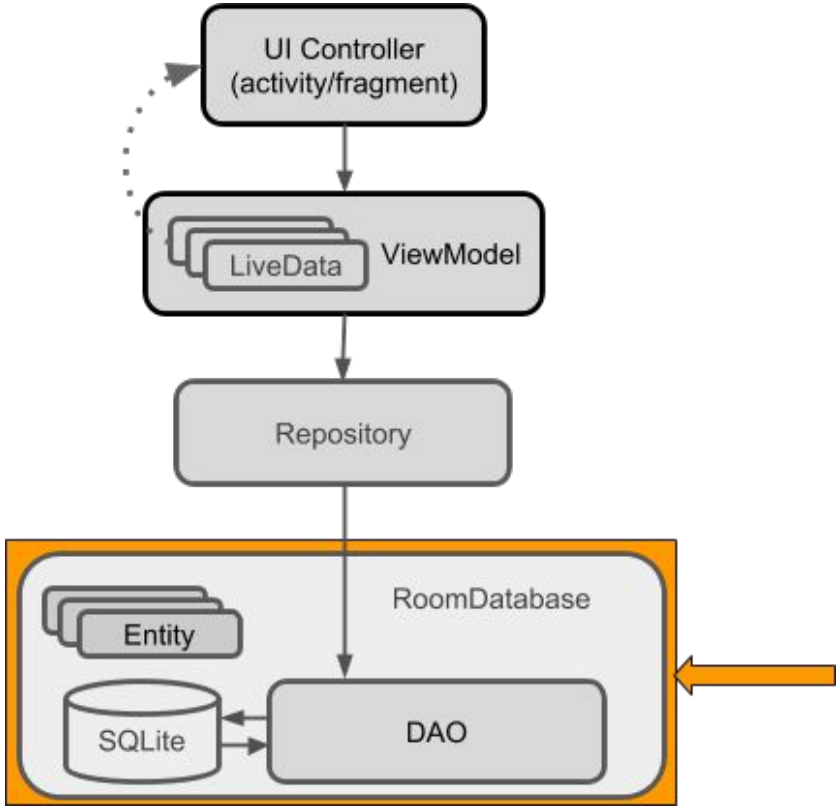
Android Architecture Components

- Data Binding
- Lifecycles
- LiveData
- Navigation
- Paging
- **Room**
- ViewModel
- WorkManager

Recommended app architecture



Room: Entity, DAO, Database



Room tutorial with example

Room = Object Relational Mapping (ORM) library

Official:

- https://developer.android.com/training/data-storage/room

YouTube:

- https://www.youtube.com/watch?v=5rfBU75sguk&list=TLPQMDMxMTIwMjAJ2zLIGtW8WA&index=1
- https://www.youtube.com/watch?v=5rfBU75sguk&list=TLPQMDMxMTIwMjAJ2zLIGtW8WA&index=2
- https://www.youtube.com/watch?v=3USvr1Lz8g8&list=TLPQMDMxMTIwMjAJ2zLIGtW8WA&index=3
- https://www.youtube.com/watch?v=5rfBU75sguk&list=TLPQMDMxMTIwMjAJ2zLIGtW8WA&index=4

Entity

Represents a table within the database

```
@Entity(tableName = "user_table")
data class User(
    @PrimaryKey(autoGenerate = true) val id: Int,
    val firstName: String,
    val lastName: String,
    val birthYear: Int
)
```

DAO - Data Access Object

Contains the methods used for accessing the database.

```
@Dao
interface UserDao {
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun addUser(user: User)

    @Query("SELECT * FROM user_table ORDER BY id ASC")
    fun readAllData(): LiveData<List<User>>
}
```

Database

- Contains the database holder and serves as the main access point for the underlying connection to your app's persisted, relational data.

```
@Database(entities =[User::class], version = 1, exportSchema = false)
abstract class UserDatabase: RoomDatabase() {
    abstract fun userDao(): UserDao
    companion object{
        @Volatile
        private var INSTANCE: UserDatabase? = null

        fun getDatabase(context: Context): UserDatabase{
        // ...
        }
    }
}
```

Database - SINGLETON!

```
fun getDatabase(context: Context): UserDatabase{
   val tempInstance = INSTANCE
   if( tempInstance != null ){
       return tempInstance;
   }
   synchronized(this){
       val instance = Room.databaseBuilder(
           context.applicationContext,
           UserDatabase::class.java,
           "user_database"
       ).build()
       INSTANCE = instance
       return instance
   }
}
```

1. Which Design Pattern?
2. Which category?
   a. Behavioural
   b. Structural
   c. Creational

Repository

```
class UserRepository(private val userDao: UserDao) {

   val readAllData: LiveData<List<User>> = userDao.readAllData()

   suspend fun addUser(user: User){
       userDao.addUser(user)
   }
}
```

12. Coroutines

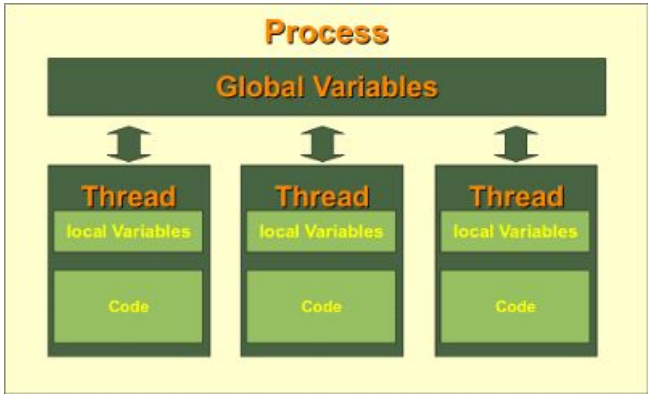
Asynchronous programming

Outline

1. Threads (OS, Java, Kotlin)
2. Android Threads
3. Kotlin Coroutines
4. Kotlin Coroutines on Android

12. 1. Threads

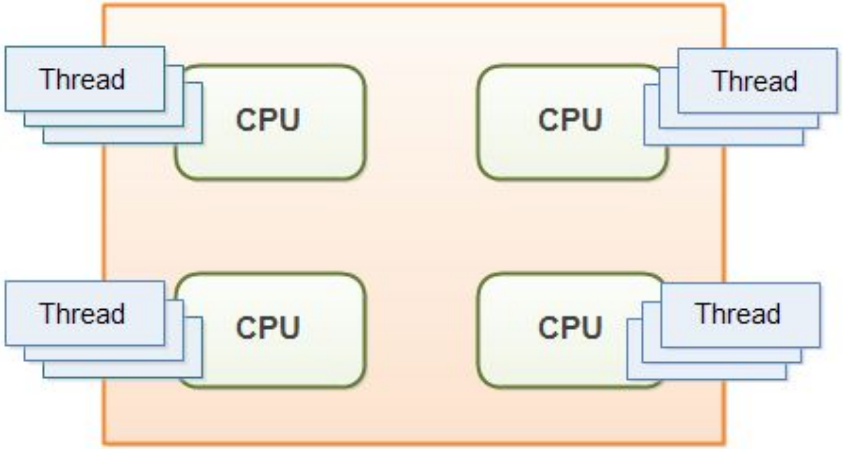
Threads



- **Concurrent** code execution
- Threads **share** the same **address space**
- Each thread has its own
  - **Stack**
  - **Instruction pointer**
- The **same code** may be executed **concurrently** by multiple threads

Thread = Lightweight **Process**

Concurrency



Concurrency

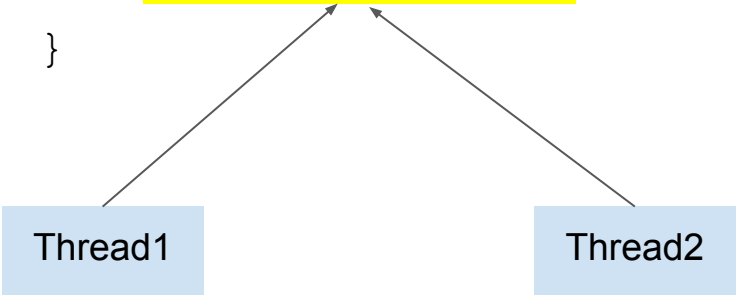
- **Real** - hardware resources
- **Simulated** by the JVM throuh threads scheduler

Synchronizing Threads

```
public class Counter{
    private int value = 0;

    public int getNextValue(){
        return value++;
     }
}
```

Synchronizing Threads

```
public class Counter{
    private int value = 0;

    public int getNextValue(){
        return value++;
    }
}
```



Synchronizing Threads

```
public static final int N = 1000000;
public static void main(String[] args) {
   Counter counter = new Counter();
   Runnable runnable = new Runnable(){
       @Override
       public void run() {
           for(int i=0; i<N;++i) {
               System.out.println(Thread.currentThread().getName() + ": " + counter.getNextValue());
           }
       }
   };
   Thread t1 = new Thread( runnable);t1.start();
   Thread t2 = new Thread( runnable);t2.start();
   try {
       t1.join();
       t2.join();
   } catch (InterruptedException e) {
       e.printStackTrace();
   }
   System.out.println(Thread.currentThread().getName() + ": " + counter.getNextValue());
}
```

Synchronizing Threads

```
value++;
```

1. **read** variable value from memory
2. **add** 1 to the value
3. **write** it back

The ++ operation is not executed **atomically** – **can be interrupted**

Synchronizing Threads

```
public class Counter{
    private int value = 0;
    public int getNextValue(){
        synchronized(this){
            return value++;
        }
    }
}
```

Synchronized blocks:

- Every **object** contains a **single lock**
- A **lock is taken** when a thread **enters** in a **synchronized** section
- If the **lock is unavailbale**, threads enter in a **waiting queue**
- If the **lock becomes available** each thread is **resumed**

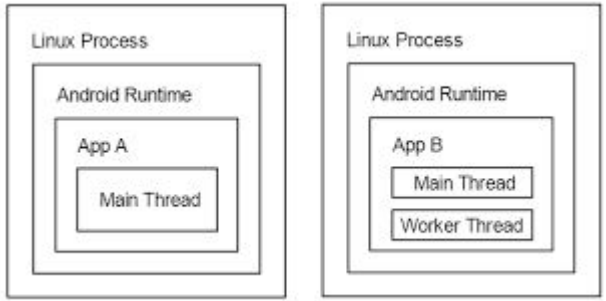
Kotlin Threads

```
class Counter{
   private var value = 0
   fun nextValue() = value++
}

val N = 1000000
fun main() {
   val counter = Counter()
   val runnable = Runnable{
       for( i in 1..N) {
           print("${Thread.currentThread().name} Counter: ${counter.nextValue()}\n")
       }
   }
   val t1 = Thread(runnable)
   val t2 = Thread(runnable)
   t1.start()
   t2.start()
   t1.join()
   t2.join()
   print("${Thread.currentThread().name} Counter: ${counter.nextValue()}\n")
}
```

12. 2. Android Threads

Android processes and threads

- By default, all components of the same application run in the **same process**



- When an application is launched, the system creates a thread of execution for the application, called "main."
  - usually **Main-Thread == UI Thread**
  - **UI Thread: every 16 ms onDraw (refresh screen)**

Android Threads - Rules

- Do not block the UI thread
  - Time consuming operations → run on a separate thread

- Do not access the Android UI toolkit from outside the UI thread
  - UI refresh → only on the Main Thread

Typical long lasting operations

- Network data communication
  - HTTP REST Request
  - SOAP Service Access
  - File Upload or Backup
- Reading or writing of files to the filesystem
  - Shared Preferences Files
  - File Cache Access
- Internal Database reading or writing
- Camera, Image, Video, Binary file processing.

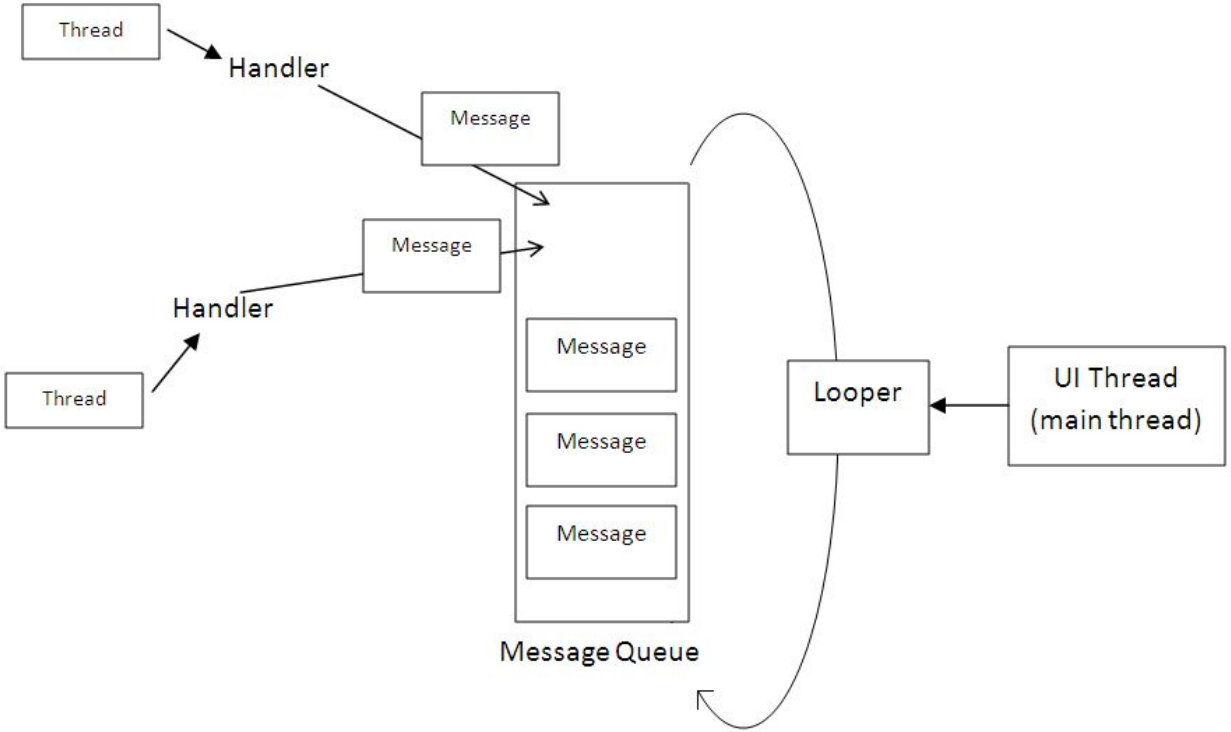
Synchronous execution - blocking

```
fun onClick(v: View) {
   val bitmap = processBitMap("image.png")
   imageView.setImageBitmap(bitmap)
}
```

Asynchronous execution

```
fun onClick(v: View) {
    Thread(Runnable {
        val bitmap = processBitMap("image.png")
        imageView.post {
            imageView.setImageBitmap(bitmap)
        }
    }).start()
}
```

Communication with the UI Thread



12. 3. Kotlin coroutines

Coroutine

- Simplifies **asynchronous programming**
  - Asynchronicity expressed as **sequential code** that is easy to read
- Light-weight **threads**
  - **Runnables** with super powers
  - Takes a block of code to run in a thread
  - Solves exception handling and cancellation

Understand Kotlin Coroutines on Android (Google I/O'19)

Suspendable functions

```
suspend fun loadData(){

    val data = networkRequest()

    show(data)

}
```

Can be:
- suspended
- resumed

Can be called:
- from a coroutine
- another suspening function

Coroutines have:
- states

Continuation

Kotlin compiler: transforms the code

```
fun loadData(cont: Continuation){
    val data = networkRequest(cont)
    show(data)
}
```

Code execution

```
suspend fun loadData(){
    val data = networkRequest()
    show(data)
}

suspend fun networkRequest(): Data =
    withContext(Dispatchers.IO){
    }
}
```

Blocking function blocks the **UI thread**

Blocking function blocks the **IO thread**

Coroutine Dispatchers

Determines **what thread** or thread pool the coroutine uses for execution.

Common dispatchers:

- **.Main** - UI/Non-blocking
- **.Default** - used for CPU-intensive computations
- **.IO** - Network & Disk

Coroutine builders

- To start a new coroutine you have to use a **Coroutine builder**
- A coroutine builder
  - takes some code
  - wrap it in a coroutine
  - pass it to the system for execution
- Coroutine builders
  - **launch**
  - **async**

Main coroutine builder: launch

```
import kotlinx.coroutines.*
fun main(){
    print("${Thread.currentThread().name}: ")
    println("Start")

    // Start a coroutine
    GlobalScope.launch {
        delay(1000)
        print("${Thread.currentThread().name}: ")
        println("Hello")
    }

    Thread.sleep(2000) // wait for 2 seconds
    print("${Thread.currentThread().name}: ")
    println("Stop")
}
```

Call a suspendable function

- Must be called from a coroutine

```
fun onButtonClicked(){
    launch{
        loadData()
    }
}
```

```
suspend fun loadData(){
    val data = networkRequest()
    show(data)
}
```

Questions

1. Who can cancel the execution?
2. Does it follow a particular lifecycle?
3. Who gets exceptions if it fails?

Scopes

Coroutine scope:

- Keep track of coroutines
- Ability to cancel them
- Is notified of failures

Using scope

```
//MyViewModel.kt

val scope = CoroutineScope(Dispatchers.Main)

fun onButtonClicked(){

    scope.launch{

        loadData()

    }

}
```

Parent

Child

Using scope

```
//MyViewModel.kt
val scope = CoroutineScope(Dispatchers.Main)
fun onButtonClicked(){
    scope.launch{
        loadData()
    }
}
fun onCleared(){
    scope.cancel()
}
```

Parent

Child

Cancels all the children

Function execution

Runs in a scope

```
suspend fun loadData(){
    val data = networkRequest()
    show(data)
}
```

Coroutine builders: `launch` vs `async`

launch	async
Creates a **new coroutine**	Creates a **new coroutine**
Fire and Forget	**Returns a value**
Takes a **dispatcher**	Takes a **dispatcher**
Executed in a **scope**	Executed in a **scope**
Re-throws exceptions	Holds on exceptions until await is called

Coroutine builders: `launch` vs `async`

launch	async

`scope.launch(Dispatchers.IO){`
`  loggingService.upload(logs)`
`}`
	`suspend fun getUser(userId: String): User=`
	`  coroutineScope{`
	`  val deferred = async(Dispatchers.IO){`
	`  userService.getUser(userId)`
	`  }`
	`  deferred.await()`
	`  }`
	`}`

Exception handling: `launch` vs `async`

launch	async

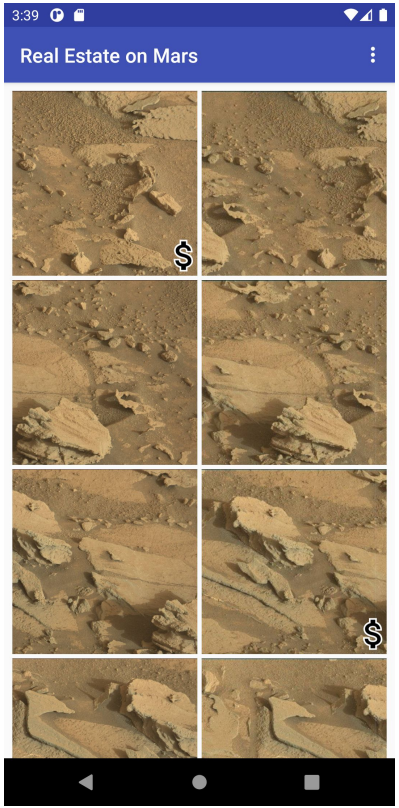
`scope.launch(Dispatchers.IO){`
`  try{`
`  loggingService.upload(logs)`
`  }catch(e: Exception){`
`  // handle Exception`
`  }`
`}`
	`suspend fun getUser(userId: String): User=`
	`  coroutineScope{`
	`  val deferred = async(Dispatchers.IO){`
	`  userService.getUser(userId)`
	`  }`
	`  try{`
	`  deferred.await()`
	`  } catch(e: Exception){`
	`  // handle exception`
	`  }`
	`  }`
	`}`

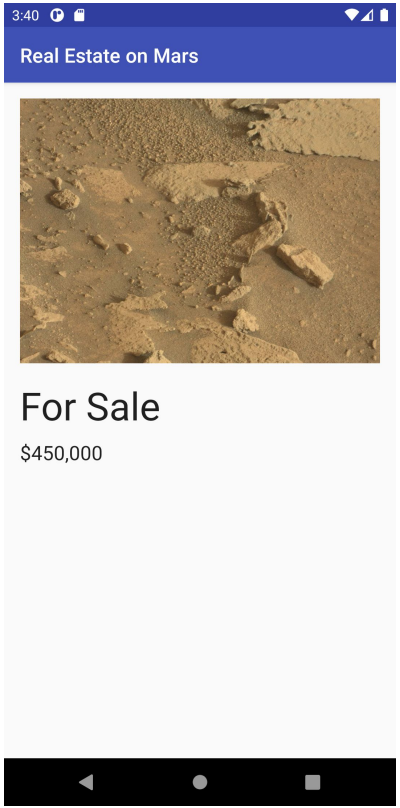
12. 4. Android coroutines

Coroutines dependencies

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.9'

implementation
'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9'
```

Android CodeLab: MarsRealEstate





[Android Kotlin Fundamentals: 8.1 Getting data from the internet]