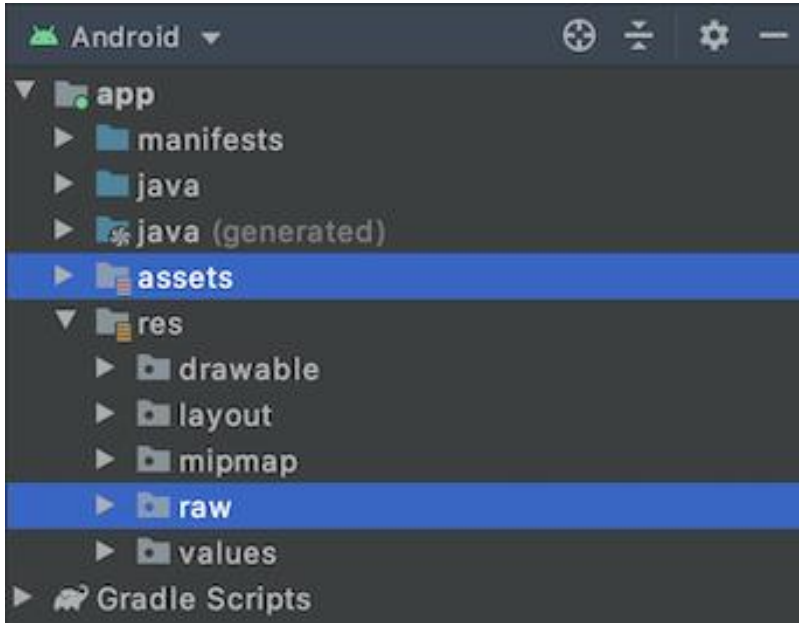
Completing List View

Add Asset

- It can be noticed that unlike **Eclipse ADT (App Development Tools), Android Studio** doesn't contain an **Assets** folder in which we usually use to keep the web files like **HTML**. Assets provide a way to add arbitrary files like text, XML, HTML, fonts, music, and video in the application. If one tries to add these files as "**resources**", Android will treat them into its resource system and you will be unable to get the raw data. If one wants to access data untouched, Assets are one way to do it. But the question arises is **why in the asset folder? We can do the same things by creating a Resource Raw Folder.**

How the asset folder is different from the Resource Raw folder?

- In Android one can store the raw asset file like JSON, Text, mp3, HTML, pdf, etc in **two** possible locations:
- **assets**
- **res/raw folder**

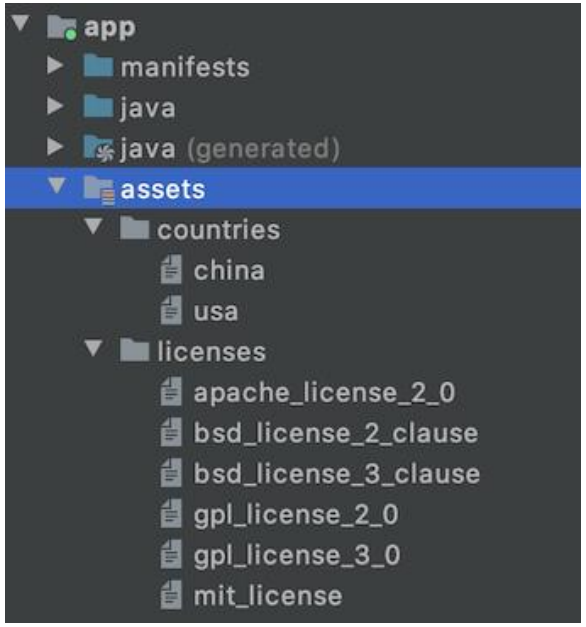


Both of them appears to be the same, as they can read the file and generate InputStream as below

```
// From assets

assets.open(assetPathFilename)

// From res/raw

resources.openRawResource(resourceRawFilename)
```

But when to use which folder?

- Below is some guidance that might be helpful to choose
- **1. Flexible File Name: (assets is better)**
- **assets:** The developer can name the file name in any way, like having capital letters (fileName) or having space (file name).
- **res/raw:** In this case, the name of the file is restricted. File-based resource names must contain only lowercase **a-z, 0-9, or underscore.**
- **2. Store in subdirectory: (possible in assets)**
- **assets:** If the developer wants to categories the files into subfolders, then he/she can do it in assets like below.



- **res/raw:** In this case, files can only be in the root folder.

3. Compile-time checking: (possible in res/raw)

- **assets:** Here, the way to read it into InputStream is given below. If the filename doesn't exist, then we need to catch it.

```
assets.open("filename")
```

- res/raw folder: Here, the way to read it into InputStream is:

```
resources.openRawResource(R.raw.filename)
```

So putting a file in the res/raw folder will provide ensure the correct file-name during compile time check.

4. List filenames at runtime: (possible in assets)

•**assets:** If the developer wants to list all the files in the assets folder, he/she has used the **list()** function and provide the folder name or **” “** on the root folder as given below.

- res/raw folder: Here, the way to read it into InputStream is:

```
resources.openRawResource(R.raw.filename)
```

So putting a file in the res/raw folder will provide ensure the correct file-name during compile time check.

4. List filenames at runtime: (possible in assets)

•**assets:** If the developer wants to list all the files in the assets folder, he/she has used the **list()** function and provide the folder name or **” “** on the root folder as given below.

```
assets.list(FOLDER_NAME)?.forEach {
  println(it)
}
```

res/raw: This is not possible in this folder. The developer has to know the filename during development, and not runtime.

So, in assets, one can read the filename during runtime, list them, and use them dynamically. In res/raw, one needs to code them ready, perhaps in the string resources file.

5. Filename accessible from XML: (possible in res/raw)

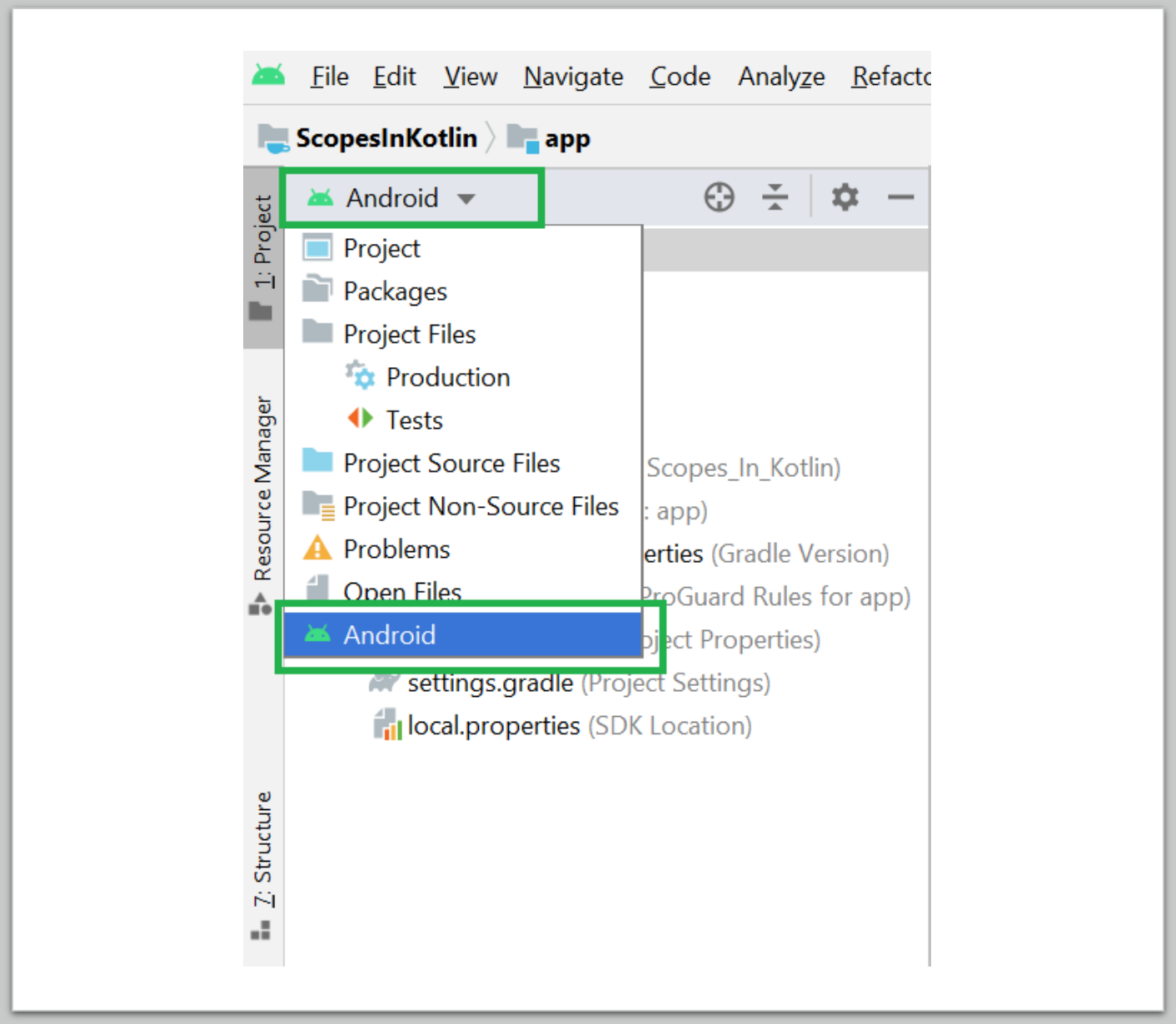
- **assets:** No simple way the developer can arrange an XML file (e.g. AndroidManifest.xml) to point to the file in the assets folder.
- **res/raw:** In any XML files like in Java, the developer can access the file in res/raw using **@raw/filename** easily.

So if you need to access your file in any XML, put it in the res/raw folder. Let's make a table to remember the whole scenario easily.

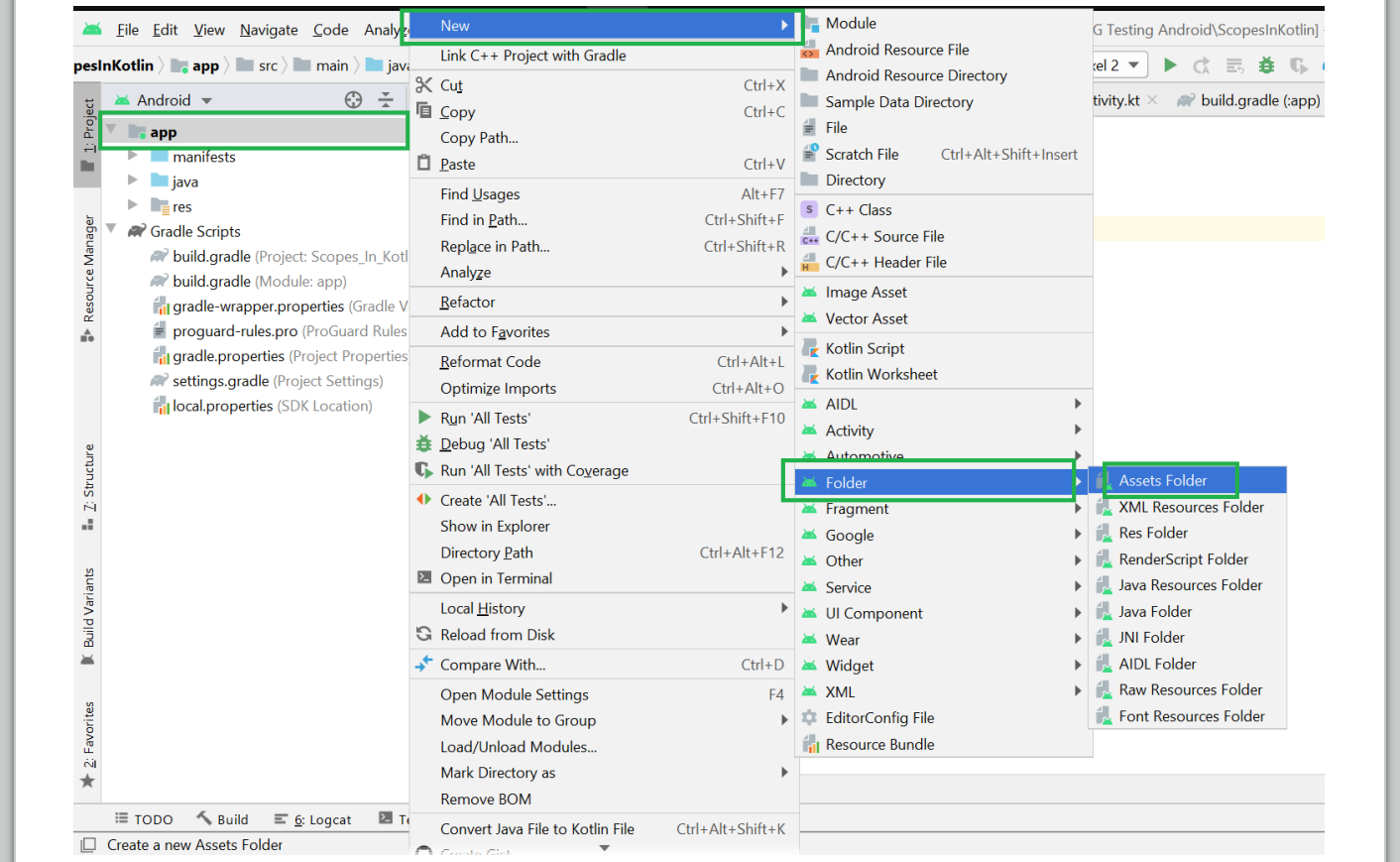
Scenario	Assets Folder	Res/Raw Folder
Flexible File Name	YES	NO
Store in subdirectory	YES	NO
Compile-time checking	NO	YES
List filenames at runtime	YES	NO
Filename accessible from XML	NO	YES

How to Create Assets Folder in Android Studio?

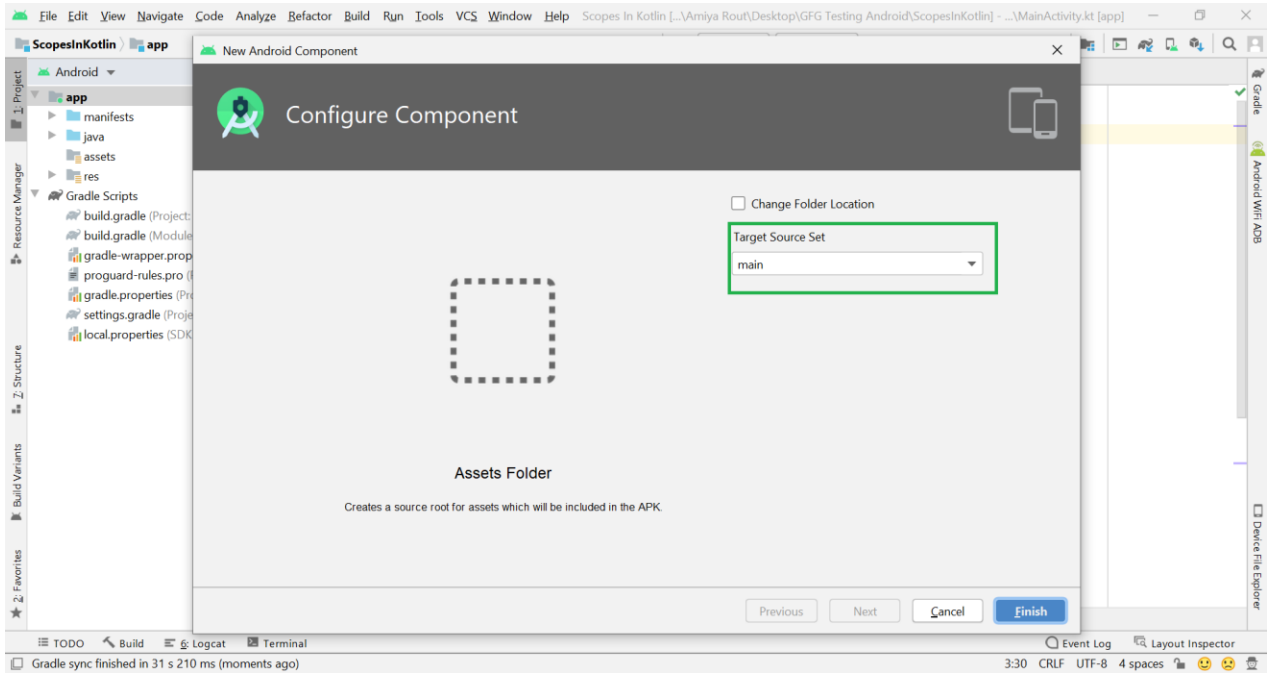
- how to create an assets folder in the android studio. Below is the step-by-step process to create an assets folder in Android studio.
- **Step**
  1: To create an **asset** folder in Android studio open your project in Android mode first as shown in the below image.



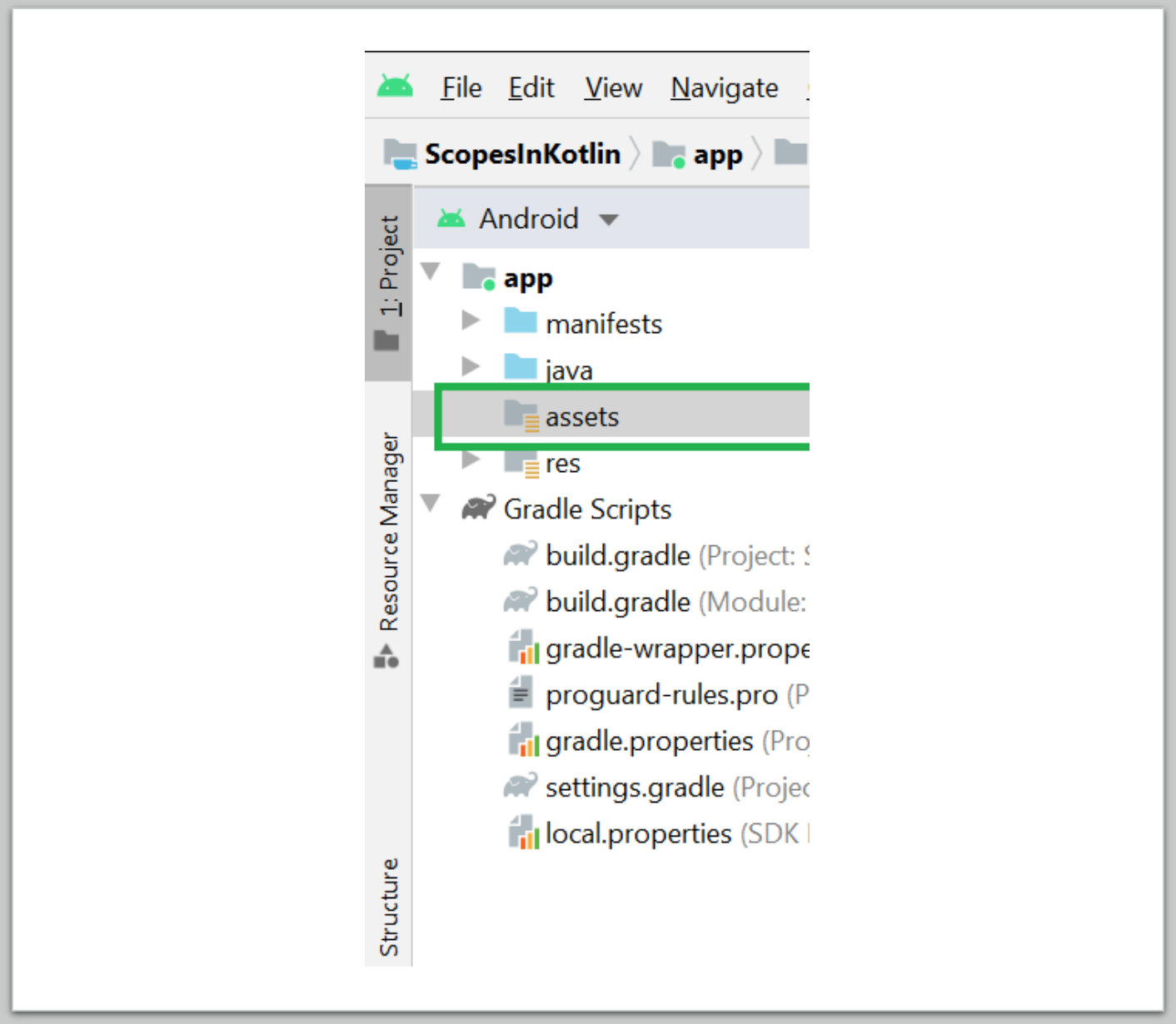
- **Step 2:** Go to the **app > right-click > New > Folder > Asset Folder** and create the **asset** folder.



- **Step 3:** Android Studio will open a dialog box. Keep all the settings default. Under the target source set, option **main** should be selected. and click on the **finish button**.



- **Step 4:** Now open the app folder and you will find the assets folder by the name of “**assets**” as shown in the below image.



findViewById in Kotlin:

Or how and why we have two ways to avoid it

In Android, if you want to change the properties of views from your layout, like changing the text of a TextView or adding an onClickListener , you would have to first assign that view's id to a variable by calling findViewById . For example, if you only had one TextView with an id of textView, you'd need to write the following Kotlin code:

```
private lateinit var textView : TextView


//in onCreate()
textView = findViewById(R.id.textView)
```

As you can imagine, this gets pretty verbose for a large number of views. With Kotlin, we have two separate ways to avoid declaring variables and writing findViewById code:

- Kotlin Android Extensions (JetBrains' approach)
- Data Binding (Google's approach)

1. Kotlin Android Extensions

To solve the issue of writing useless code, JetBrains has created the Kotlin Android Extensions which have a number of features, but exist primarily to avoid the need for findViewById code. Using them is straightforward, you simply need to import kotlinx.android.synthetic.main.activity_main.* and then you simply reference views by their id from the layout file, for example:

```
textView.text = "Hello world!"
```

This is super useful, because it makes all of the code we wrote initially unnecessary. The way it works is basically it calls findViewById itself and it creates a cache of views it references any time you want to access that view again.

2. Data Binding

To solve the same issue, as well as add some extra functionality (for example observing data changes), Google has created the Data Binding Library. Using this is a bit more involved than Kotlin Android Extensions, but in theory it allows for more complex manipulation of the data and, most importantly, it avoids findViewById calls entirely by creating an object that already contains references to all the views that have ids.
To use it, you first need to enable data binding in your app's build.gradle

```
android {
    //...

    dataBinding {
        enabled = true
    }
}
```

Then, you need to wrap each layout xml file with a <layout> tag.

Finally, need to create a variable to reference the object holding all the view ids:

```
private lateinit var binding: ActivityMainBinding

//in onCreate() replace setContentView() with:
binding = DataBindingUtil.setContentView(this, R.layout.activity_main)
```

After that, we can access all views using their ids like so:

```
binding.textView.text = "Hello world!"

//or if you have multiple properties or views using the with keyword
with(binding) {
      textView.text = "Hello world"
}
```

Both approaches offer multiple benefits and expanded functionality so I encourage digging deeper into them if you're curious, I just wanted to show you how each is used strictly in the context of avoiding writing boilerplate code. If you have any questions you're welcome to ask them in the comments section below.

Model View Controller

The Model-View-Controller pattern

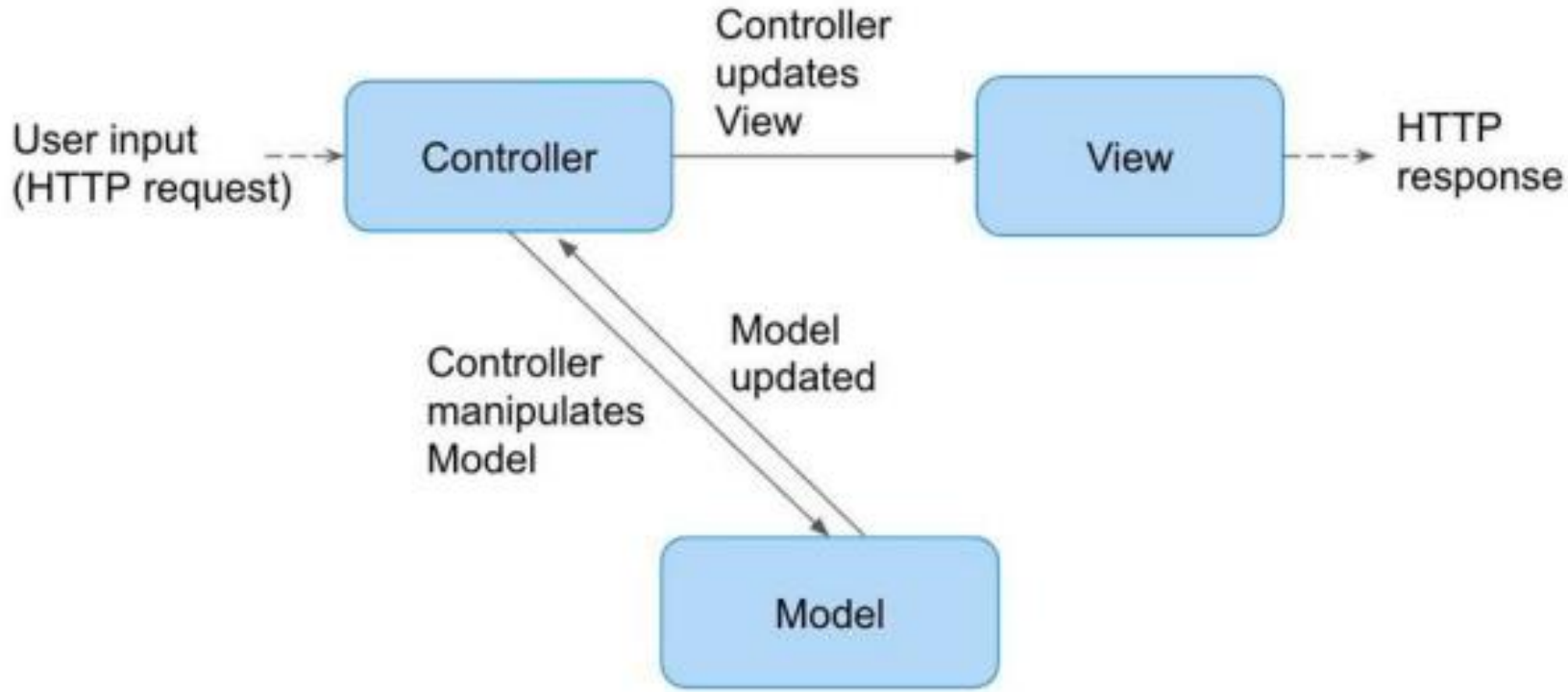
The Model View Controller pattern is a pattern that separates components of a software system, based on responsibilities. In this pattern, three components with distinct responsibilities are defined as follows:

- **Model** is the data layer. This includes the data objects, database classes, and other business logic, concerned with storing the data, retrieving the data, updating the data, etc.
- **View** renders the data using the Model in a way suitable for user interface. It is what gets displayed to the user.
- **Controller** is the brains of the system. It encapsulates the logic of the system, and it controls both the Model and View. The user interacts with the system through this controller.

The general idea is that the Model should not be concerned with how it's ultimately displayed to the user. On the flip side, a View should not be concerned with the values of the actual data it is displaying, only that it needs to be displayed. The Controller then acts as the glue between these two components, orchestrating how the data should be displayed

Organizing a system this way allows for two advantages: separation of concerns and unit testability. Because components are isolated from each other, each focused on only one responsibility, the system is more flexible and modular. Each component is able to be unit tested on its own and could be swapped out for another version without affecting the other.

For example, the View could be changed out while the underlying data and logic remained the same.

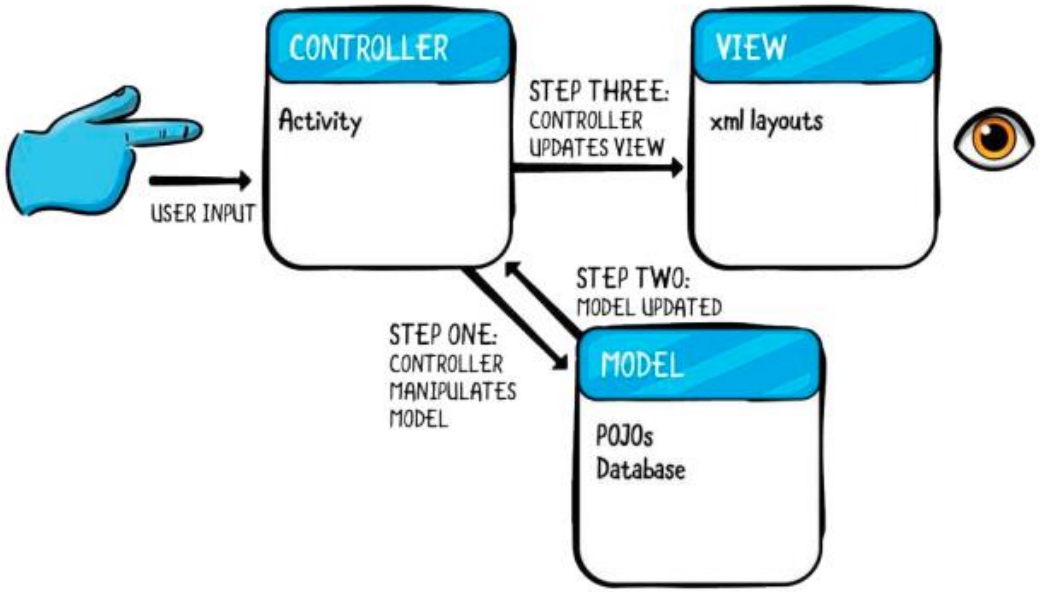


Typical MVC flow in a web application.

- The user inputs are HTTP requests.
- The Controller handles those requests by updating the data in the Model.
- The View returned to the user is a rendered HTML page, displaying the data.

Applying MVC to Android

Seeing that this pattern worked well in organizing other software systems, developers naturally tried to apply the MVC pattern to the development of Android apps as well when Android was introduced. Here is one way in which the MVC pattern was adapted for Android:



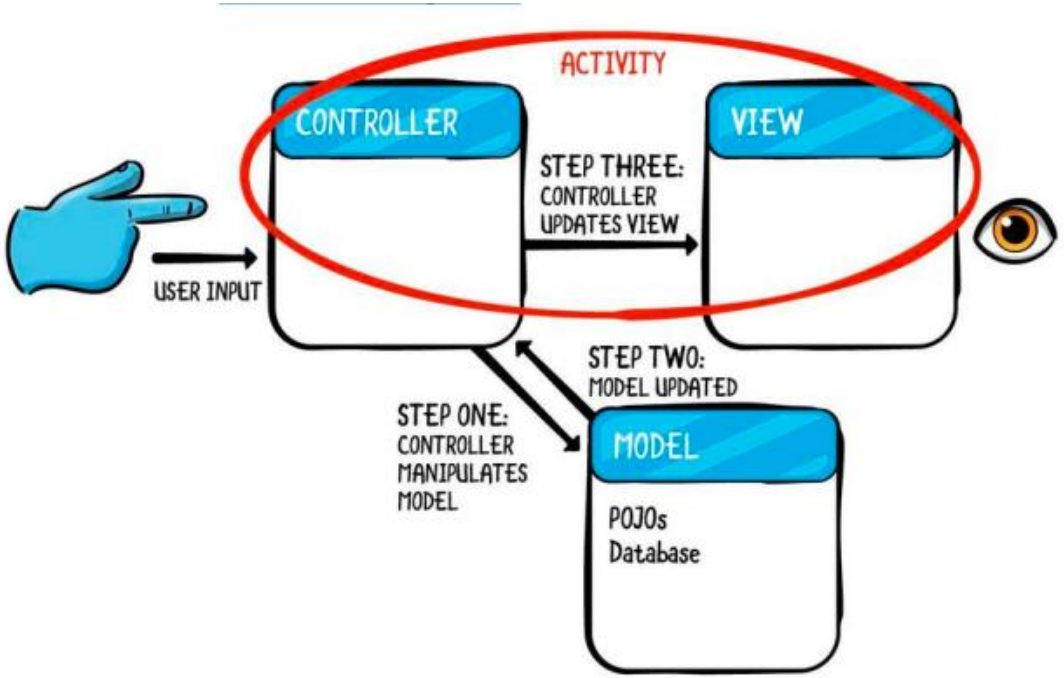
Attempt to apply MVC to Android.

In MVC, the main entry point to the app is through the Controller, so it makes sense to give the role of the Controller to the Android **Activity** where it can take in user inputs, such as a button click, and respond accordingly. The Model consists of the data objects, which, in Android, are just regular Kotlin data classes, as well as the classes to handle data locally and remotely. The **View** consists of the layout files in an Android app.

However, there is a problem with this oversimplified adaptation of MVC to Android.

When the Controller updates the View, the View cannot update if it is merely a static .xml layout. Instead, the Activity must almost always contain some **view logic**, such as showing or hiding views, displaying a progress bar or updating the text on screen, in response to user input. Furthermore, not all layouts are inflated through .xml; what if your Activity dynamically loaded your layouts?

If the Activity must hold references to views and logic for changing them as well as all the logic for its Controller responsibilities, then the Activity effectively serves as both the Controller and the View in this pattern



More realistic application of MVC to Android.

Having the Activity act as both the Controller and the View is problematic for two reasons. First, it defeats the goal of MVC, which was to separate responsibilities among three distinct components of a software system. Second, having the Activity as the Controller in MVC poses a problem for unit testing, which you will read more about in the next chapter.

MVP

- although the MVC architecture pattern theoretically allows an app to achieve separation of concerns and unit testability, in practice, MVC didn’t quite work on Android.

- The problem was that the Android Activity, unfortunately, served the role of both View and Controller. Ideally, you would want to somehow move the Controller out of the Activity into its own class so that the Controller can be unit testable.

- One architecture pattern that achieves this goal is MVP, which stands for Model View Presenter, and is made up of the following parts:

- •Model is the data layer, responsible for the business logic.
- •View displays the UI and listens to user actions. This is typically an Activity (or Fragment).
- •Presenter talks to both Model and View and handles presentation logic.

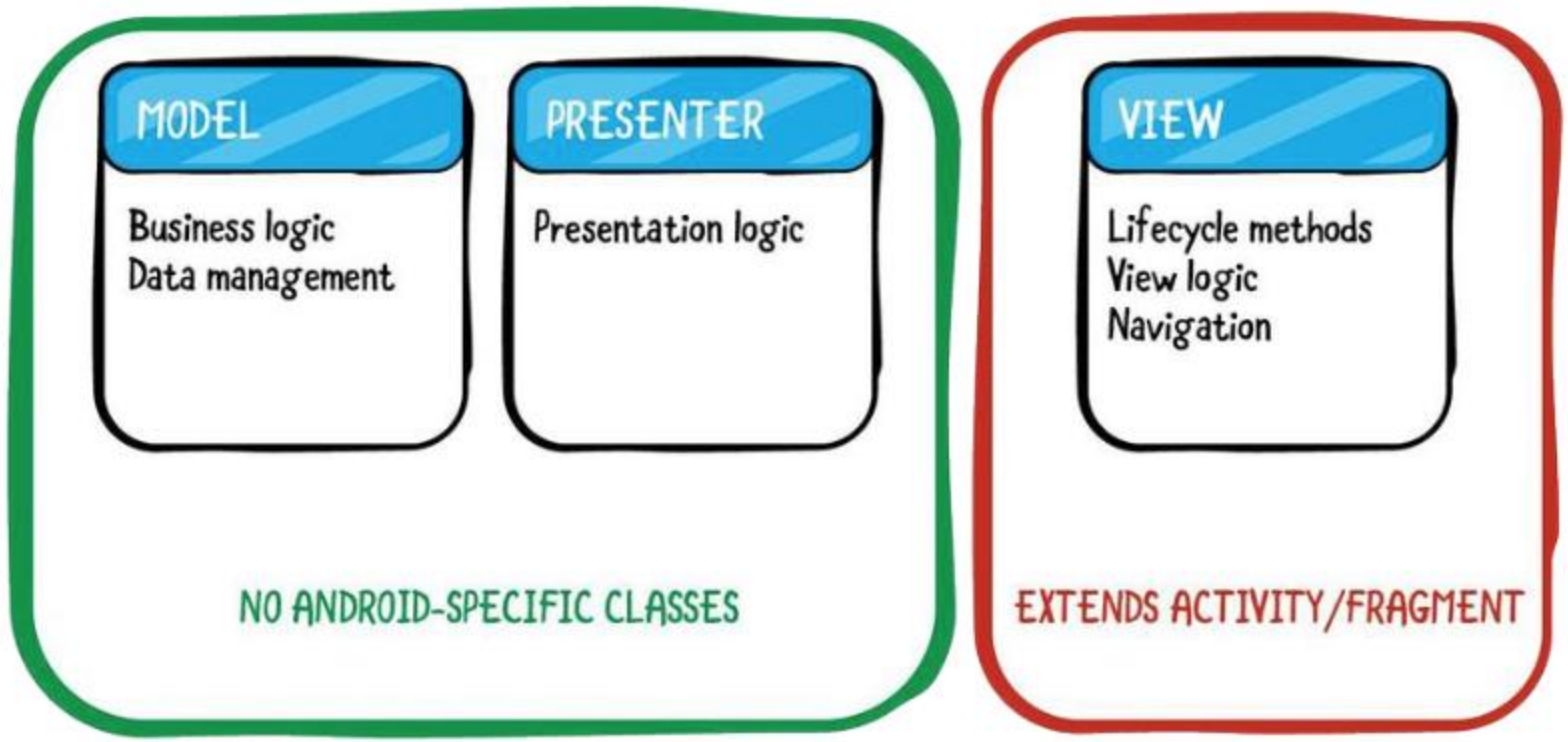
The Model-View-Presenter pattern

- With MVP, the Model remains the same as with the MVC architecture. The Model is the data layer, and is responsible for the business logic: retrieving the data, storing the data and changing the data.

- The View is also the same as with the MVC architecture, responsible for displaying the UI, except that, with MVP, this role is specifically designated to an Activity or Fragment.
- The View will hide and display views, handle navigating to other Activities through intents, and listen for operating system interactions and user input.The Presenter is the glue class that talks to both the Model and View. Any code that does not directly handle UI or other Android framework-specific logic should be moved out of the View and into the Presenter class.
- For example, while the View will listen for button clicks, the presentation logic of what happens after a user clicks a button should go into the Presenter class. Similarly, when the Model has updated, it is not the responsibility of the Model to know how the View ultimately wants to display the data.

- It is the Presenter’s job to do any additional mapping or formatting of the data before handing it to the View to display it. This kind of logic is known as presentation logic, and it is handled by the aptly named Presenter.
- While the View extends from an Activity or Fragment, the Model and Presenter do not extend Android framework-specific classes, and, for the most part, they should not contain Android framework-specific classes. In other words, there should be no references to the com.android.* package in the Model or Presenter. As you will read later, this rule allows for both the Model and the Presenter to be unit tested with this pattern.

The Presenter should not contain Android framework-specific classes

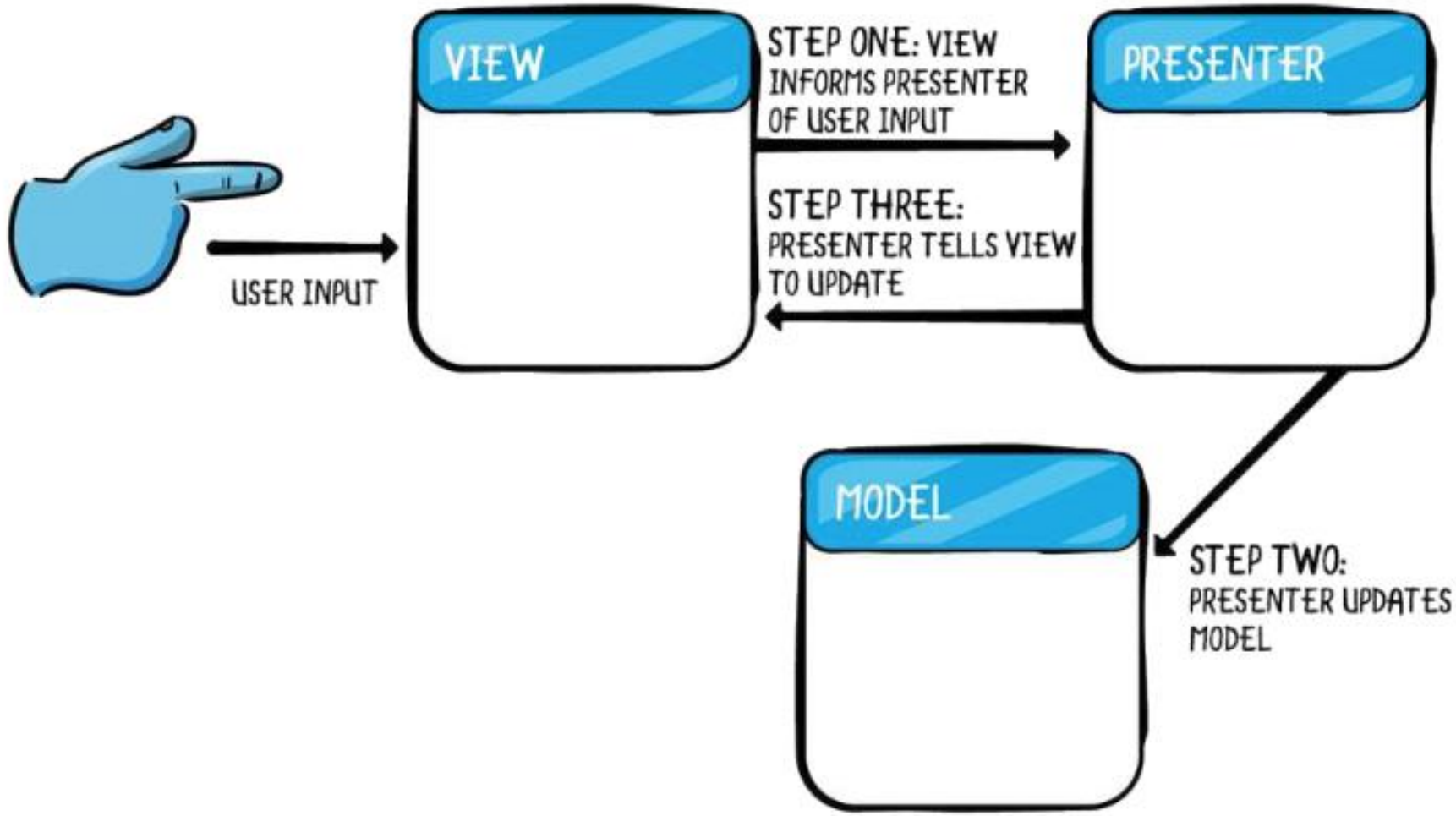


- In summary, with the MVP pattern, the roles of the Model and the View are equivalent to those in MVC, except that now the Activity or Fragment is explicitly designated the role of the View. The Controller has now been replaced with a Presenter class, which, based on the definition above, seems to serve a similar role as the theoretical Controller.

- However, there are several key characteristics of the MVP pattern that differentiate it from MVC and allow for it to work in Android: the order in which events occur, the breaking of the ties between View and Model and the use of interfaces.

MVP Flow

- To use MVP, you'll need to reimagine the order in which events occur in the flow of an Android app. Unlike in MVC wherein the Controller is supposed to be the main entry point for the app, the main entry point is the View.
- If the View serves as the entry point for user input — such as a button tap — that implies that the View must be an Activity or Fragment, as these Android classes are able to handle user input.
- The View can then inform the Presenter of the user input, and the Presenter can handle it by updating the Model. Once the Model has been updated, the Presenter can then update the View with the most up-to-date data.


VIEW
PRESENTER
MODEL
USER INPUT
STEP ONE: VIEW INFORMS PRESENTER OF USER INPUT
STEP THREE: PRESENTER TELLS VIEW TO UPDATE
STEP TWO: PRESENTER UPDATES MODEL

MVC flow

Separation of concerns and unit testing

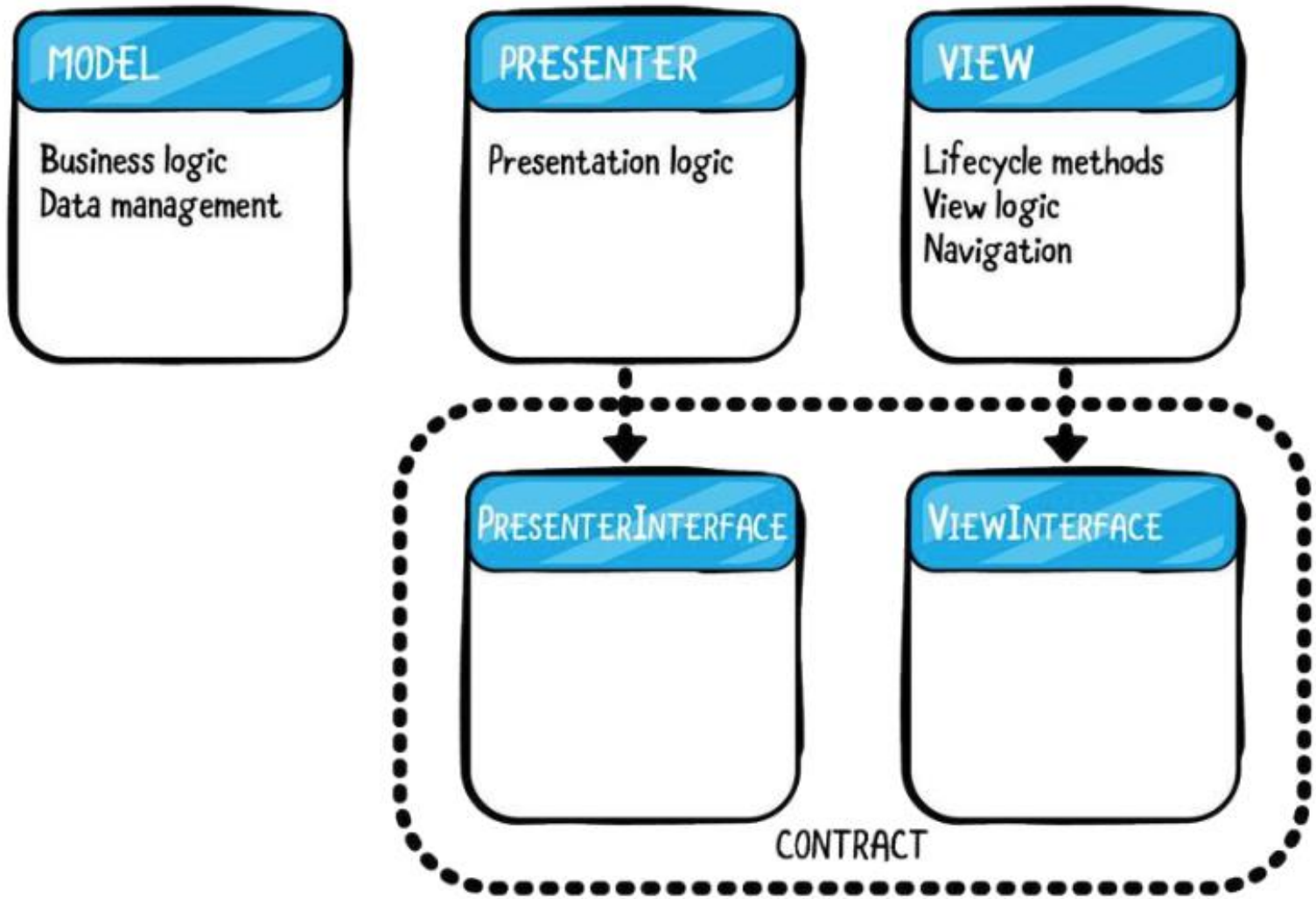
- As the example above demonstrates, the Presenter serves as a middleman between the Model and the View, shuttling information and updates between the two classes. Because the Presenter handles these responsibilities, the Model and the View do not need to be aware of each other. By breaking the direct tie between the Model and the View, MVP allows for better separation of concerns.
- Furthermore, with this pattern, the Presenter can now be unit tested. Because it does not extend any class specific to the Android framework, the Presenter can be instantiated through a constructor like var presenter = Presenter(), allowing us to call methods on an instance of the Presenter like presenter.onDeleteTapped(moviesToDelete) to test that it behaves as expected in unit tests.

Using Interfaces

- With the exception of Android Architecture Component classes, which are allowable in Presenters, the Presenter should not contain references to any Android framework-specific classes, such as Context, View, or Intent. This rule allows you to write regular JUnit tests on the Presenter.
- However, the Presenter, of course, needs to talk to the View, which means it needs a reference to the Activity, an Android framework-specific class. How can you get around this reference to an Android class when you write your unit tests?

- The way to resolve this problem is to create interfaces for the Presenter and View, keeping the interfaces in a single class called the Contract class. The Presenter class implements the PresenterInterface and the Activity implements the ViewInterface.

- The Presenter class will then hold a reference to an instance of the ViewInterface interface, rather than a reference directly to the Activity. This way, the Presenter and View interact with each other through interfaces rather than actual implementations, which is, in general, a good software engineering principle that allows decoupling of the two classes.
- In this case, it also has the additional benefits of removing the Android-specific framework class Activity from the Presenter and allows you to mock the ViewInterface in the Presenter for unit testing.



Create interfaces for the Presenter and View

- While it is necessary to create an interface for the View, strictly speaking, the same is not true of the Presenter interface. It would be perfectly fine to allow the View to interact with the actual implementation of the Presenter, especially since it is unlikely to have changing implementations of the Presenter.

- However, you choose to have a Presenter interface in this project so that the Contract class neatly documents the relationship between the Presenter and the View, clearly outlining the interactions.

MVP advantages and concerns

- By dividing an Activity into separate Model, View, and Presenter classes with interfaces, you are able to achieve separation of concerns as well as unit-testable Models and Presenters. These are certainly considerable achievements, even if it means having to navigate through interfaces, which can be confusing at first. In Android Studio, CMD + OPTION + B on Mac or CTRL + ALT + B on PC is the keyboard shortcut for finding your way to actual implementations of interface methods.

MVP advantages and concerns

- Additionally, it is important to give some thought to what happens when the operating system destroys the Activity. When your Activity is destroyed, you must be sure to have your Presenter destroy any subscriptions or AsyncTasks or else that will cause problems when the task completes and the Activity is no longer there.
- You'll also need to think about what happens to the Presenter upon destruction of the Activity. If you want the memory held by the Presenter to be freed and garbage-collected when the Activity is destroyed, you must make sure there are no references to the Presenter in any other classes that will continue to live in memory.
- If you do want your presenter survive Activity destruction, you can try to save some state through the onSaveInstanceState, or use loaders, which survive configuration changes. In our example project, we will allow the Presenter to be destroyed when the Activity is destroyed.

MVVM

- MVVM stands for Model-View-ViewModel. MVVM is an architectural pattern whose main purpose is to achieve separation of concerns through a clear distinction between the roles of each of its layers:
-
  •View displays the UI and informs the other layers about user actions.
- •ViewModel exposes information to the View.
- •Model retrieves information from your datasource and exposes it to the ViewModels.

- At first glance, MVVM looks a lot like the MVP and MVC architecture patterns
- The main difference between MVVM and those patterns is that there is a strong emphasis that the ViewModel should not contain any references to Views.

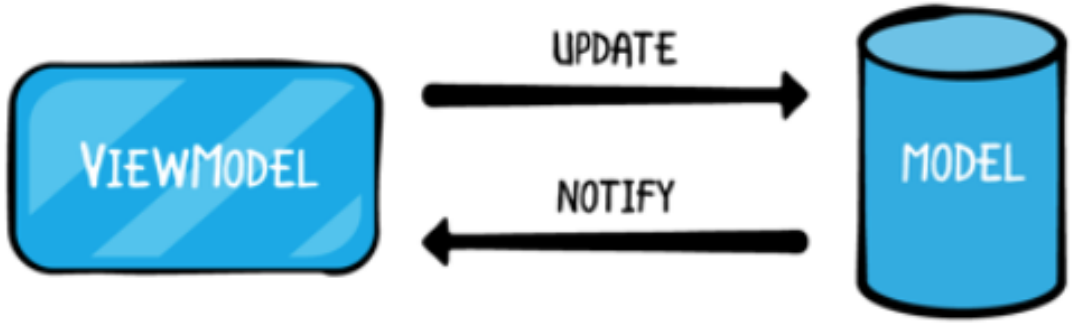
- The ViewModel only provides information and it is not interested in what consumes it.

- This makes it easy to create a one-to-many relationship wherein your Views can request information from any ViewModel they need.

- Also of note regarding the MVVM architecture pattern is that the ViewModel is also responsible for exposing events that the Views can observe.
- Those events can be as simple as a new user in your Database or even an update to a whole list of a movie catalog. Now, you will explore how each of the MVVM layers work one by one.

The Model

- The Model, better known as DataModel, is in charge of exposing relevant data to your ViewModels in a way that is easy to consume. It should also receive any events from the ViewModel that it needs to create, read, update or delete any necessary data from the backend.



- In Android, you usually create Models as Kotlin data classes that represent the information that you obtain from your data source, such as an API or a database. For example, say you have an app that displays information about the latest movies. You would surely create a Movie class that contains data such as the title, description, time and release date of the movie.
- When following this architecture pattern, you should strive to stick to the single-responsibility principle of software design, creating a Model for each logical object in your domain. This will make it much easier for you to create the necessary ViewModels later on.

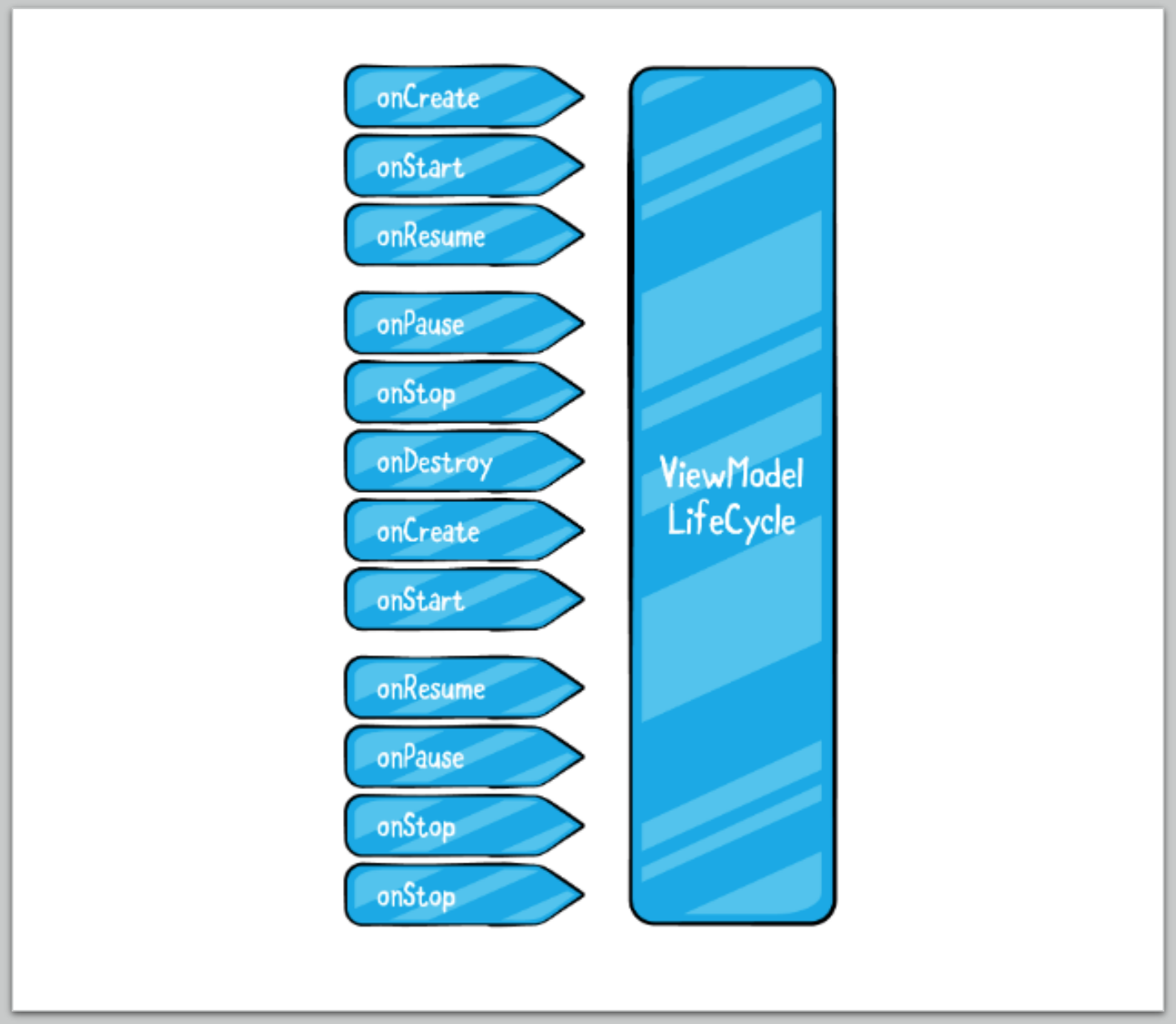
The ViewModel

- The ViewModel retrieves the necessary information from the Model, applies the necessary operations and exposes any relevant data for the Views.
- The Android platform is responsible for managing the lifecycle events of the classes that handle the UI, such as activities and fragments. The operating system can destroy or re-create your activities at any time in response to certain user actions or events.
- The problem is that, if Android destroys or re-creates an activity or fragment, all data contained within those components is lost. For example, your app may include a list of movies in one of its activities. If the activity is destroyed and re-created, the list of movies will have to be retrieved again. This may slow down your app if the list is housed in an external database or API.

- The most common solution to this problem is to save your data in your onSaveInstanceState() bundle and restore it later in your onCreate() method. But this approach only works for primitive data such as Integers or simple classes that can be serialized and deserialized.

- Thanks to Google's new Architecture Components, you now have a special class to build your ViewModels called ViewModel.The ViewModel class is specially designed to manage and store information in a lifecycle-aware manner.
- This means that the data stored inside it can survive configuration/lifecycle changes like screen rotations.

- The ViewModel remains in memory until the lifecycle object to which it belongs has completely terminated. This behavior applies to activities when they finish and in fragments when they are detached.
- In the next illustration, you can see how the ViewModel remains active and retains information through the whole lifecycle of an activity, even when it is destroyed:



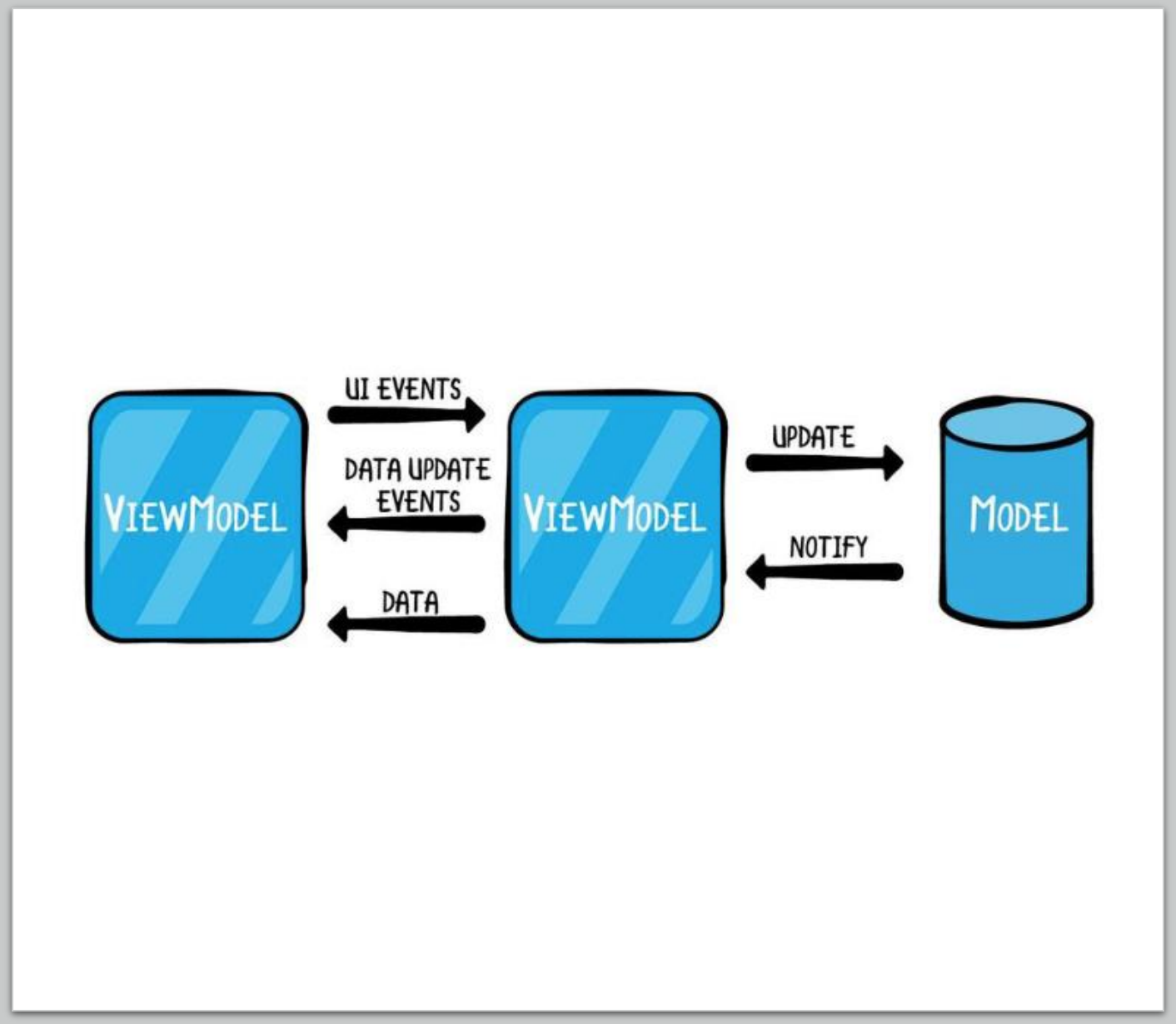
- To communicate changes in the data, ViewModels can expose events that the Views can observe and react accordingly. Those events can be as simple as a new user having been created in the database or an update to an entire movies catalog.
- This way, ViewModels don't need to have any reference to Activities, Fragments or Adapters.

The View

- The View is what most of us are already familiar with, and it is the only component that the end user really interacts with. The View is responsible for displaying the interface, and it is usually represented in Android as Activities or Fragments. Its main role in the MVVM pattern is to observe one or more ViewModels to obtain the necessary information it needs and update the UI accordingly.
- The View also informs ViewModels about user actions. This makes it easy for the View to communicate to more than one Model. Views can have a reference to one or more ViewModels, but ViewModels can never have any information about the Views.

- In Android, you will usually communicate the data between the Views and the ViewModels with Observables, using libraries such as RxJava, LiveData or DataBinding.

- You can see how the interaction between each layer works, below:



MVVM advantages and concerns

- One problem that the MVC architecture patterns have in common is that the Controllers and the Presenters are sometimes very hard to test due to their close relationship with the View layer. By handling all data manipulation to ViewModels, unit testing becomes very easy since they don't have any reference to the Views.
- One problem present in some architectures, MVC in particular, is that the business logic is quite difficult to test due to a lack of separation from the View logic. By confining all data manipulation to the ViewModel, and by keeping it free of any View code, the business logic becomes unit testable, as it can be executed without requiring the Android runtime.

- Another problem with the MVC pattern is that there is usually confusion as to which code goes where. Sometimes, when code doesn't fit in the Model or the View, it is put in the Controller. This often leads to a common problem known as fat controllers, whereby the controller classes become overly large and difficult to maintain.

- MVVM solves the fat controller issue by providing a better separation of concerns. Adding ViewModels, whose main purpose is to be completely separated from the Views, reduces the risk of having too much code in the other layers.

MVVM vs. MVC vs. MVP

- You might be wondering why you would want to use MVVM over MVC or MVP. After all, MVC and MVP are among the most common Android architecture patterns and are both very easy to understand. There has been endless debate on which approach is best, but the answer largely boils down to personal preference.

- As we usually say in the development world, there is no silver bullet to solve every software design issue. And although MVVM is a very useful development pattern, it also has some disadvantages.

- The main disadvantage of this architecture pattern is that it can be too complex for applications whose UI is rather simple. Adding as much level of abstraction in such apps can result in boiler plate code that only makes the underlying logic more complicated.

MVI

- MVI stands for Model-View-Intent. MVI is one of the newest architecture patterns for Android. The architecture was inspired by the unidirectional and cyclical nature of the Cycle.js framework and brought to the Android world by Hannes Dorfaman.
- MVI works in a very different way compared to its distant relatives such as MVC, MVP or MVVM. The role of each of its components goes as follows:

•Model: Represents a state. Models in MVI should be immutable to ensure a unidirectional data flow between them and the other layers in your architecture.

•Intent: Represents an intention or a desire to perform an action by the user. For every user action, an intent will be received by the View, which will be observed by the Presenter and translated into a new state in your Models.

•View: Just like in MVP they are represented by Interfaces, which are then implemented in one or more Activities or Fragments.

Model

- In other architecture patterns such as MVVM or MVP, the Models act as a rather simple layer to hold your data and act as a bridge to the backend of your app such as your databases or your APIs.

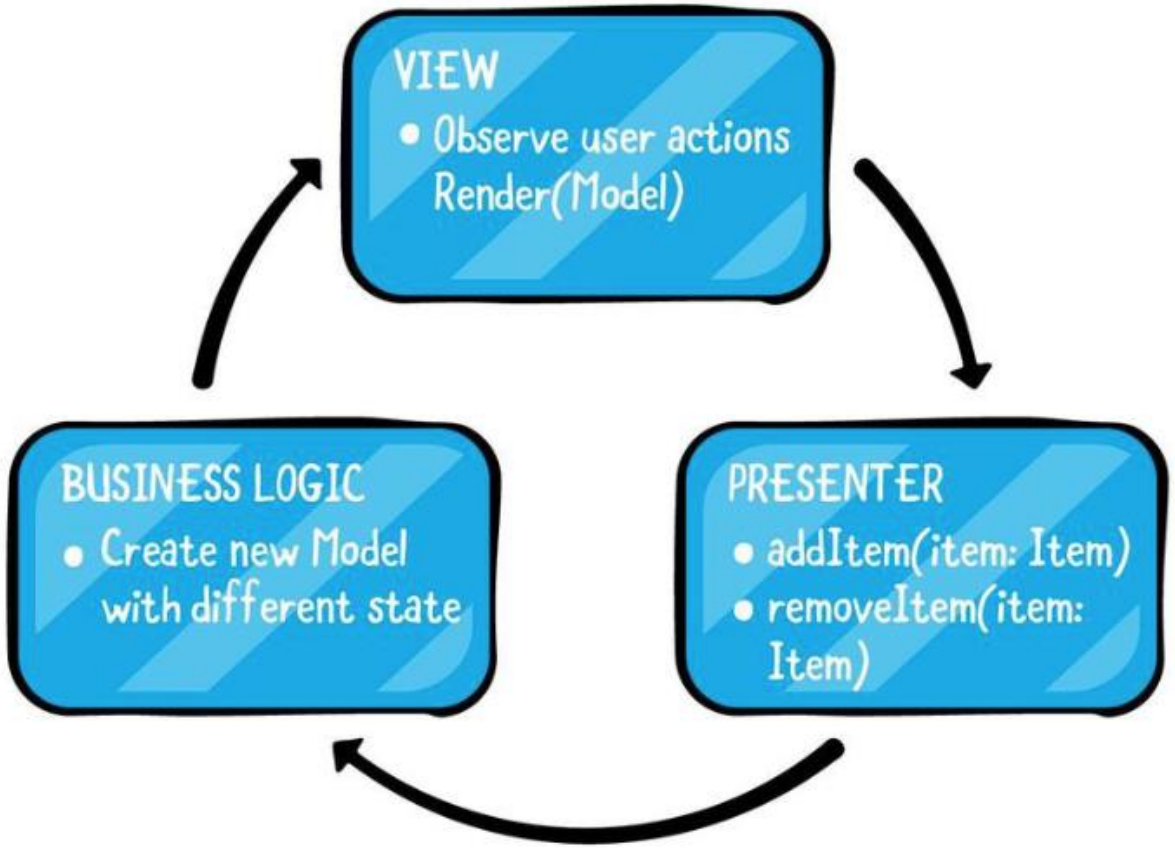
- However, in MVI, Models have a much more important role; they not only hold data, but also represent the state of your app.

But... What is the state of your app?

- reactive programming is a paradigm in which you react to a change, such as the value of a variable or a button click in your UI. Well, when your apps react to this change, they enter into a new state. The new state is usually, but not always, represented as a UI change with something like a progress bar, a new list of movies or a completely different screen.

- To make things clearer, imagine that you want to add a new item to a list of todo items. This is how a typical implementation of MVI will work under the hood:
- 1.An Observable will send a notification to its subscribers about a new item being added to the list. Usually, this means adding a new record in a local and/or remote database.
- 2.A method in your Presenter, such as presenter.addNewItem(item), will be called.
- 3.Your business logic will use your current Model to create a new Model which contains the new set of items. It is very important that you remember the immutability of your Models at this point since you can't just use the current Model to add the new item, you need to create a new one.
- 4.Your Presenter will then be notified about a new state in your app with an Observer.
- 5.Your Presenter will then call a render() method in your View and pass the new Model with the updated information as an argument.
- 6.Your View will display the new list of items based on the Model received from the Presenter.

- The interaction between the different layers can be illustrated by the following diagram:



Do you notice something in particular about this diagram? If you said cyclical flow, you are correct

- Thanks to the immutability of your Models, and the cyclical flow of your layers, you get other benefits:

- •Single State: Since immutable data structures are very easy to handle and can only be managed in one place, you can be sure there will only be a single state between all the layers in your app.
- •Thread Safety: This is specially useful while working with reactive apps that make use of libraries such as RxJava or LiveData. Since no methods can modify your Models they will always need to be recreated and kept in a single place, with this you make sure that there will be no other side effects such as different objects modifying your Models from different threads.

Views & Intents

- In MVI, just like in in MVP, the Views are defined with the help of an Interface that acts as a contract which is implemented by a Fragment or an activity. The difference lies in the fact that Views in MVI tend to have a single render() method that accepts a state to render to the screen and different intent() methods as Observables that respond to user actions.
- The intents in MVI don't represent the usual android.content.Intent class that is used for things like starting a new class. Intents in MVI represent an action to be performed that is translated to a change of the state in your app. For this simple example you only have one intent, the getItemsIntent(), but you can have any number of intents in your Views depending on the number of actions.

State Reducers

- With your usual mutable Models it is very easy to change the state of your app. Whenever you need to add, remove or update some underlying data you just need to call a method in your Models such as this:

```
myModel.addItems(newItems)
```

- But you already learned that, since your Models are immutable, they need to be recreated each time the state of your app changes. If you just want new data to be displayed you can just create a new Model, but what do you do when you need information from a previous state?

- This is where State Reducers come to save the day.State Reducers are a concept derived from the reducer functions in reactive programming.

- Reducer functions, to put it simply, are functions that provide steps to properly merge things into a single component called the accumulator.

MVI Advantages and Concerns

- Just like the previous patterns, Model-View-Intent is just an additional tool that you have at your disposal to create maintainable and scalable apps.

The main advantages of MVI are:

- •A unidirectional and cyclical data flow for your app.
- •Consistent state during the whole lifecycle of your Views.
- •Immutable Models that provide a reliable behavior and thread safety on big apps.

- Probably the only downside of using MVI rather than other architecture patterns for Android is that the learning curve for this pattern tends to be a bit higher since you need to have a decent amount of knowledge of other intermediate/advanced topics such as reactive programming, multi-threading and RxJava.

- Therefore, other architecture patterns such as MVC or MVP might be easier to grasp for beginner Android developers.

Android Unidirectional Data Flow with Live Data

The Unidirectional Data Flow (UDF) pattern has improved the usability and performance of Coinverse since the first beta launched in February. Coinverse is the first app creating audiocasts covering technology and news in cryptocurrency. Upgrades using UDF include more efficient newsfeed creation, removal of adjacent native ads, and faster audiocast loading.

UDF Strengths

Linear flow of app logic

Control UI & async events

Pre-defined test structure

Faster debugging

Structure

View state — Final persisted data of a screen displayed to a user

View events — State changes initiated by the user or Android lifecycle

View effects — One time UI occurrences that don’t persist like navigation or dialogs

View **ViewModel** **Repository**

1 Creates *events* → 2 Requests data → 3 Returns data

5 Observes view *state*/*effects* ← 4 Creates + updates view *state*/*effects*

Improvements

View events — Create hard contract between the view and ViewModel.

View effects — Enable multiple + simultaneous effects with LiveData attributes and simplify code with custom extension functions.

Repository — Improve data transformations and flatten request callbacks with Coroutines.

ViewModel — Simplify data requests with Kotlin Coroutines Flow.

Create Contract Between the View and ViewModel 1.0
With the stream of events, if an event is not processed in the ViewModel, the app still compiles and the code will not perform as expected.

1.Pass events stream to ViewModel.

```
private val viewEvent: LiveData<Event<ViewEvent>> get() = _viewEvent
private val _viewEvent = MutableLiveData<Event<ViewEvent>>()

override fun onCreate(savedInstanceState: Bundle?) {
    ...
    if (savedInstanceState == null)
      _viewEvent.value = Event(FeedLoad(...))
}

override fun onResume() {
  super.onResume()
  viewEvent.observe(viewLifecycleOwner, EventObserver { event ->
    contentViewModel.processEvent(event)
  })
}
```

2. Process events in View Model with process Event.

```
val viewState: LiveData<ViewState> get() = _viewState
val viewEffect: LiveData<Event<ViewEffect>> get() = _viewEffect

private val _viewState = MutableLiveData<ViewState>()
private val _viewEffect = MutableLiveData<Event<ViewEffect>>()

fun processEvent(event: ViewEvent) {
    when (event) {
        is ViewEvent.FeedLoad -> {
          // Populate view state based on network request response.
          _viewState.value = ContentViewState(getMainFeed(...),...)
          _viewEffect.value = Event(UpdateAds())
        }
        ...
}
```

2.0

Instead of loosely processing the View events in the ViewModel with a LiveData steam, a contract can be defined with an Interface between the View and ViewModel. This ensures all expected events are processed. If the ViewModel does not override an event from the Interface a Lint error will throw and the code will not compile.

1.Define view events.

```
interface ContentViewEvents {
    fun feedLoad(event: FeedLoad)
    fun feedLoadComplete(event: FeedLoadComplete)
    ...
}
```

2. Send events to View Model with interface.

```
private lateinit var viewEvents: ContentViewEvents

// Builder for view events interface
fun initEvents(viewEvents: ViewEvents) {
    this.viewEvents = viewEvents
}

override fun onCreate(savedInstanceState: Bundle?) {
    contentViewModel.attachEvents(this)
    viewEvents.feedLoad(FeedLoad(...))
}
```

3. Implement events interface in View Model.

```
class ViewModel : ViewModel(), ContentViewEvents {
  // Immutable state attributes
  val viewState: LiveData<State> get() = _viewState
  val viewEffect: LiveData<Event<Effects>> get() = _viewEffect

  private val _viewState = MutableLiveData<State>()
  private val _viewEffect = MutableLiveData<Event<Effects>>()

  fun attachEvents(fragment: Fragment) {
    if (fragment is ContentFragment)
      fragment.initEvents(this)
    }

  override fun feedLoad(event: FeedLoad) {
    _viewState.value = ViewState(getContentList(...))
    _viewEffect.value = ViewEffects(updateAds = liveData {emit(Event(UpdateAdsEffect()))})
  }
}
```

3. Implement events interface in View Model.

```
class ViewModel : ViewModel(), ContentViewEvents {
  // Immutable state attributes
  val viewState: LiveData<State> get() = _viewState
  val viewEffect: LiveData<Event<Effects>> get() = _viewEffect

  private val _viewState = MutableLiveData<State>()
  private val _viewEffect = MutableLiveData<Event<Effects>>()

  fun attachEvents(fragment: Fragment) {
    if (fragment is ContentFragment)
      fragment.initEvents(this)
    }

  override fun feedLoad(event: FeedLoad) {
    _viewState.value = ViewState(getContentList(...))
    _viewEffect.value = ViewEffects(updateAds = liveData {emit(Event(UpdateAdsEffect()))})
  }
}
```

Enable Multiple + Simultaneous Effects

Another point from Yigit regarding 1.0 is the ViewEffect is only able to send one event at a time from the ViewModel to the UI. For most cases this may work. However, with more features added in the future, numerous effects may need to process simultaneously.

1.0

1.Create view effect types in order to use parent Sealed class effect to pass state change from the ViewModel to the view.

```
// Business logic sends to UI.
sealed class ViewEffect {
  data class ContentSwipedEffect(
    val feedType: FeedType,
    val actionType: UserActionType,
    val position: Int) : ContentEffectType()
  class UpdateAds : ViewEffect()
  ...
}
```

2. Send view effect to the view by updating the state of the view effect Live Data value.

```
val viewState: LiveData<ViewState> get() = _viewState
val viewEffect: LiveData<Event<ViewEffect>> get() = _viewEffect

private val _viewState = MutableLiveData<ViewState>()
private val _viewEffect = MutableLiveData<Event<ViewEffect>>()

fun processEvent(event: ViewEvent) {
    when (event) {
        is ViewEvent.FeedLoad -> {
          // Populate view state based on network request response.
          _viewState.value = ContentViewState(getMainFeed(...),...)
          _viewEffect.value = Event(UpdateAds())
        }
        ...
}
```

2.0

In order to handle multiple + simultaneous effects,
the ViewEffect model is refactored to have LiveData attributes for each effect type. The same as before, the effect change is updated in the ViewModel as a LiveData variable.

1. Create view effect with LiveData attributes for each type.

```
/** View state effects for content feeds */
data class ContentEffects(
        val contentSwiped: LiveData<Event<ContentSwipedEffect>> = liveData {},
        val updateAds: LiveData<Event<UpdateAdsEffect>> = liveData {},
        ...)

sealed class ContentEffectType {
        data class ContentSwipedEffect(
                val feedType: FeedType,
                val actionType: UserActionType,
                val position: Int) : ContentEffectType()
        class UpdateAdsEffect : ContentEffectType()
}
```

2. Send the view effect to the view by creating the view effect state, ContentEffects, and saving the LiveData value to the view effect attribute updateAds. This ensures the state change will be observed in the view and update the ads accordingly. Because each attribute is of LiveData type, multiple effects can be saved at the same time.

```
override fun feedLoad(event: FeedLoad) {
  ...
  _viewEffect.value = ContentEffects(updateAds = liveData { emit(Event(UpdateAdsEffect())) }
}
```

Use Custom Extension Functions

Once the view effect state ContentEffects is initialized in the ViewModel, in order to update the existing attributes we use Kotlin’s copy extension function to preserve the current state of the ContentEffects' attributes.

To improve the View Model readability a custom send extension function has been created to handle this process for each view effect attribute.

```
/**
 * Updates [ContentEffects] effect state.
 *
 * @receiver MutableLiveData<ContentEffects> view effect state
 * @param effect ContentEffectType
 */
fun MutableLiveData<ContentEffects>.send(effect: ContentEffectType) {
    this.value = when (effect) {
        is ContentSwipedEffect ->
            this.value?.copy(contentSwiped = liveData { emit(Event(effect)) })
        is UpdateAdsEffect ->
            this.value?.copy(updateAds = liveData { emit(Event(effect)) })
        ...
    }
}
```

2. Then use the send extension to pass the Update Ads Effect to the view. We are using the send extension because the view effect state object has already been initialized, and we just need to update it's attribute.

```
override fun updateAds(event: UpdateAds) {
  _viewEffect.send(UpdateAdsEffect())
}
```

In 1.0 the Repository returns LiveData which is processed in the ViewModel, then updated in the final persisted view. UsingLiveData between the ViewModel and view is great because it is tied to the view lifecycle and the view data is observed when the state updates.

LiveData Can be Problematic in the Repository

- Data processing is complicated because LiveData must be observed in the view vs. Coroutine Flows which can be returned anywhere.
- Transforming LiveData requires either aswitchMap or using MediatorLiveData.
- LiveData creates unnecessary syntax in the ViewModel, which we'll explore in the last section.

Improve Data Transformations with Flow

Using Kotlin's Coroutine Flow we can return data directly in the Repository, and apply transformations if needed before returning the final data to the ViewModel.

1.0

Make a data request for the main feed, parse the response, and return LiveData to the ViewModel.

```
fun getMainFeedList(...) = MutableLiveData<Lce<Result.PagedListResult>>().also { lce ->
lce.value = Lce.Loading()
contentEnCollection.orderBy(TIMESTAMP, DESCENDING)
.whereGreaterThanOrEqualTo(TIMESTAMP, timeframe).get()
.addOnCompleteListener { change ->
val contentList = arrayListOf<>()<Content>()
change.result?.documentChanges?.map { doc ->
contentList.add(doc.document.toObject(Content::class.java))
}
database.contentDao().insertContentList(contentList)
lce.value = Lce.Content(ContentResult.PagedListResult(queryMainContentList(timeframe), ""))
}.addOnFailureListener {
lce.value = Lce.Error(ContentResult.PagedListResult(...))
}
}
```

2.0

To properly handle the lifecycle of the Flow, it’s important to launch the original call from the ViewModel to make sure it’s tied to the ViewModel lifecycle.

```
fun getMainFeedList(isRealtime: Boolean, timeframe: Timestamp) = flow<Lce<PagedListResult>> {
emit(Loading())
try {
val contentList = contentEnCollection.orderBy(TIMESTAMP, DESCENDING)
.whereGreaterThanOrEqualTo(TIMESTAMP, timeframe).get().await()
.documentChanges
?.map { change -> change.document.toObject(Content::class.java) }
?.filter { content -> !labeledSet.contains(content.id) }
database.contentDao().insertContentList(contentList)
lce.emit(Lce.Content(PagedListResult(queryMainContentList(timeframe), "")))
} catch (error: FirebaseFirestoreException) {
lce.emit(Error(PagedListResult(
null,
CONTENT_LOGGED_IN_NON_REALTIME_ERROR + "${error.localizedMessage}")))
}
}
```

Flatten Request Callbacks

Notice above how 2.0's Repository is more readable without the multiple layers of nesting required when using callbacks. In order to remove callbacks we can convert these requests to Coroutines. There are two types of network requests to convert.

- One-time requests
- Realtime updates

One-time requests

One time requests are Rest API calls that return a single Json response.
Firebase Firestore's get() method is an example of a one time request.
we can use await as seen above to cleanly return the asynchronous result.
As Rosário links to, the coroutines-android and coroutines-play-services libraries are required to implement theawait extension function to remove the callback nesting.

Realtime updates

Realtime requests involve creating a constant stream of data with a Websocket in order receive data as it is transmitted from the network.
Firebase Firestore's addSnapshotListener is an example of a realtime request.

```
private suspend fun syncLabeledContent(...) {
  val response = user.document(COLLECTIONS_DOCUMENT)
    .collection(collection)
    .orderBy(TIMESTAMP, DESCENDING)
    .whereGreaterThanOrEqualTo(TIMESTAMP, timeframe)
    .awaitRealtime()
  if (response.error == null) {
    val contentList = response.packet?.documentChanges?.map { doc ->
      doc.document.toObject(Content::class.java)
    }
    database.contentDao().insertContentList(contentList)
  } else
    lce.emit(Error(PagedListResult(
      null,
      "Error retrieving user save_collection: ${response.error?.localizedMessage}")))
}
```

ViewModel

Simplify Data Requests with Kotlin Coroutines' Flow 1.0

```
private fun getContentList(...) = Transformations.switchMap(
repository.getMainFeedList(...)) {
lce -> when (lce) {
// SwitchMap must be observed for data to be emitted in ViewModel.
is Lce.Loading -> Transformations.switchMap(queryMainContentList(timeframe)) {
pagedList -> MutableLiveData<PagedList<Content>>().apply {
this.value = pagedList
}
}
is Lce.Content -> Transformations.switchMap(lce.packet.pagedList!!) {
pagedList -> MutableLiveData<PagedList<Content>>().apply {
this.value = pagedList
}
}
is Lce.Error -> {
// Handle error and emit pre-existing data.
Transformations.switchMap(queryMainContentList(timeframe)) {
pagedList -> MutableLiveData<PagedList<Content>>().apply {
this.value = pagedList
}
}
}
}
```

2.0

```
private fun getContentList(...) = liveData {
  repository.getMainFeedList(isRealtime, timeframe).collect { lce ->
    when (lce) {
      is Loading -> { emitSource(queryMainContentList(timeframe)) }
      is Lce.Content -> { emitSource(lce.packet.pagedList!!) }
      // Handle error and emit pre-existing data.
      is Error -> { emitSource(queryMainContentList(timeframe))}
    }
  }
}
```

- There is no longer the need to create switchMaps and MutableLiveData instances in order to process and return the data in the ViewModel.
- The liveData Coroutine is used to return LiveData to update the view state with less boilerplate code.

Ways to write to File in Kotlin

Here are several ways to write to a file in Kotlin:

- BufferedWriter and use() function
- PrintWriter and use() function
- File.writetext() function
- File.writeBytes() function
- Files.write() function

Using BufferedWriter

java.io.File provides bufferedWriter() method that returns a new BufferedWriter object. We're gonna call use method to write to file by executes the block function and closes it.

```
File(name).bufferedWriter().use { out ->
  out.write(text)
  out.write(nextText)
}
```

```
package com.trail.kotlin.writefileZWSP
import java.io.FileZWSP
fun main(args: Array<String>) {ZWSP
    val outStr1: String = "trail\nProgramming Tutorials"ZWSP
    val outStr2: String = "Becoming trail..."ZWSP
    File( pathname: "trail.txt").bufferedWriter().use { out ->ZWSP
        out.write(outStr1)ZWSP
        out.write( str: "\n")ZWSP
        out.write(outStr2)ZWSP
    }ZWSP
```

Trail
Programming Tutorials
Becoming Trail...

Using PrintWriter

You can use java.io.File printWriter() method that returns a new PrintWriter object. PrintWriter prints formatted representations of objects to a text-output stream.
We also need to call use method for executing the block function.
With PrintWriter, we can use println() function so that it's helpful to write line by line.

```
    File(name).printWriter().use { out ->
    out.println(line1)
    out.println(line2)
}
```

```
package com.bezkoder.kotlin.writefile
import java.io.File
fun main(args: Array<String>) {
    val outStr1: String = "trail\nProgramming Tutorials"
    val outStr2: String = "Becoming trail..."
    File( pathname: "trail.txt").printWriter().use { out ->
        out.println(outStr1)
        out.println(outStr2)
    }
```

Trail
Programming Tutorials
Becoming Trail...

Using File writeText function

java.io.File has writeText() that sets the content of file as text (UTF-8) or you can specify charset in the second parameter. Using writeText() will override the existing file.
To append text to existing file, use appendText().

```
val myFile = File(name)
myFile.writeText(text)
myFile.appendText(nextText)
```

```
package com.trail.kotlin.writefile
import java.io.File
fun main(args: Array<String>) {
    val outStr1: String = "trail\nProgramming Tutorials"
    val outStr2: String = "Becoming trail..."
    val myFile = File( pathname: "trail.txt")
    myFile.writeText(outStr1)
    myFile.appendText( text: "\n" + outStr2)
```

Trail
Programming Tutorials
Becoming Trail...

Using File writeBytes function

Instead of write with Text, we can write array of bytes to a file using writeBytes(). Kotlin also provides appendBytes() function to append array of bytes to the content of existing file.

```
val myFile = File(name)
myFile.writeBytes(bytesArray)
myFile.appendBytes(nextBytesArray)
```

```
package com.trail.kotlin.writefile
import java.io.File
fun main(args: Array<String>) {
    val outStr1: String = "trail\nProgramming Tutorials"
    val outStr2: String = "Becoming trail..."
    val file2 = File( pathname: "trail.txt")
    file2.writeBytes(outStr1.toByteArray())
    file2.appendBytes(("\n" + outStr2).toByteArray())
}
```

Trail
Programming Tutorials
Becoming Trail...

Using Files write function

java.nio.file.Files provides static method write() that Writes bytes to a file.
There is options parameter to specify how the file is created or opened.

```
Files.write(path, bytes, StandardOpenOption.CREATE_NEW);
Files.write(path, bytes, StandardOpenOption.APPEND);
```

```
package com.trail.kotlin.writefile
import java.io.File
import java.nio.file.Files
import java.nio.file.StandardOpenOption
fun main(args: Array<String>) {
    val outStr1: String = "trail\nProgramming Tutorials"
    val outStr2: String = "Becoming trail..."
    val myfile = File( pathname: "trail.txt")
    Files.write(myfile.toPath(), outStr1.toByteArray(), StandardOpenOption.CREATE_NEW)
    Files.write(myfile.toPath(), ("\n" + outStr2).toByteArray(), StandardOpenOption.APPEND)
}
```

Trail
Programming Tutorials
Becoming Trail...

Kotlin write JSON to file

Now we have a Tutorial class like this:

```kotlin
package com.trail.kotlin.writefile
class Tutorial(
    val title: String,
    val author: String,
    val categories: List<String>
) {
    override fun toString(): String {
        return "Category [title: ${this.title}, author: ${this.author}, categories: ${this.categories}]"
    }
}
```

What if we want to write List<Tutorial> to JSON file?
There are two steps:

- use Gson to convert Kotlin object to JSON string
- write the string to JSON file

If you want to make pretty JSON string, GsonBuilder will help you.
Let’s do it right now.

```
package com.trail.kotlin.writefile
import java.io.File
import com.google.gson.Gson
import com.google.gson.GsonBuilder
fun main(args: Array<String>) {
    val tutsList: List<Tutorial> = listOf(
        Tutorial("Tut #1", "trail", listOf("cat1", "cat2")),
        Tutorial("Tut #2", "trail, listOf("cat3", "cat4"))
    );
    val gson = Gson()
    val jsonTutsList: String = gson.toJson(tutsList)
    File( pathname: "trail.json").writeText(jsonTutsList)
    val gsonPretty = GsonBuilder().setPrettyPrinting().create()
    val jsonTutsListPretty: String = gsonPretty.toJson(tutsList)
    File( pathname: "trail.json").writeText(jsonTutsListPretty)
}
```

[{"title":"Tut #1","author":"bezkoder","categories":["cat1","cat2"]},{"title":"Tut #2","author":"zkoder","categories":["cat3","cat4"]}]

Trail Json file

```
[
  {
    "title": "Tut #1",
    "author": "bezkoder",
    "categories": [
      "cat1",
      "cat2"
    ]
  },
  {
    "title": "Tut #2",
    "author": "zkoder",
    "categories": [
      "cat3",
      "cat4"
    ]
  }
]
```