ViewModel & LiveData

Download the full App created in this guide

https://drive.google.com/file/d/19hr4KqbyGgZvhjkx1M4rBZIZcxzJ5wbm/view?usp=sharing

Download the Navigation View Model app from this guide
https://drive.google.com/file/d/144foCUlTej6Amnw4M9Zpa1o2dXIdePr_/view?usp=sharing

Steps:

1. Create a new project use the Empty Activity template

2. Make sure in your project structure dependencies that the view model and live data are included as you can see in the photo attached. If not Search for ViewModel and copy the latest dependencies into your app Gradle file and sync. Copy also the LiveData dependencies we will need it later on. Here is s link to https://developer.android.com/jetpack/androidx/releases/lifecycle#declaring_dependencies

```
androidx.lifecycle:lifecycle-common:2.3.1
androidx.lifecycle:lifecycle-livedata:2.0.0
androidx.lifecycle:lifecycle-livedata-core:2.3.1
androidx.lifecycle:lifecycle-runtime:2.3.1
androidx.lifecycle:lifecycle-viewmodel:2.3.1
androidx.lifecycle:lifecycle-viewmodel-savedstate:2.3.1
```

3. In the Activity layout file select the existing TextView widget and use the Attributes tool window to change the id property to result_text. Drag a Number (Decimal) view from the palette and position it above the existing TextView. With the view selected in the layout refer to the Attributes tool window and change the id to dollar_text. Drag a Button widget onto the layout so that it is positioned below the TextView, double-click on it and to edit the text and change it to read “Convert”. With the button still selected, change the id property to convert_btn. Click on the Infer constraints button to add any missing layout constraints. If you don’t want to bother yourself with ui now, you can copy the activity_main xml file located in our starter files.
4. Now add you Kotlin code to preform the conversion and show it in the result Text View.
5. When all is done execute your program and check it.
6. Well done - now please rotate your screen - What Happened??

Our complete code so far should look like this:

```
class MainActivity : AppCompatActivity() {

    companion object {
        const val EURO_DOLLAR_RATE = 1.17
    }

    private lateinit var binding : ActivityMainBinding

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


        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.button.setOnClickListener {  it: View!

            binding.resultText.text = "${binding.dollarText.text.toString().toDouble().times(EURO_DOLLAR_RATE)}"

        }
```

Configuration changes

What Happened is that screen orientation changes is a **configuration change**(like language, resolution, text size and much more) in order to load the more specific resources for that configuration the system automatically sends a kill signal to all of the activities and fragments displayed on the screen and creates a new instances - when the new instance is created all the specific resources are loaded by default. That is why all the information is gone.
In our case the data is simple but think of cases where we work with REST API or with other remote databases - this requires refetching our data again and again and can affect the user experience.

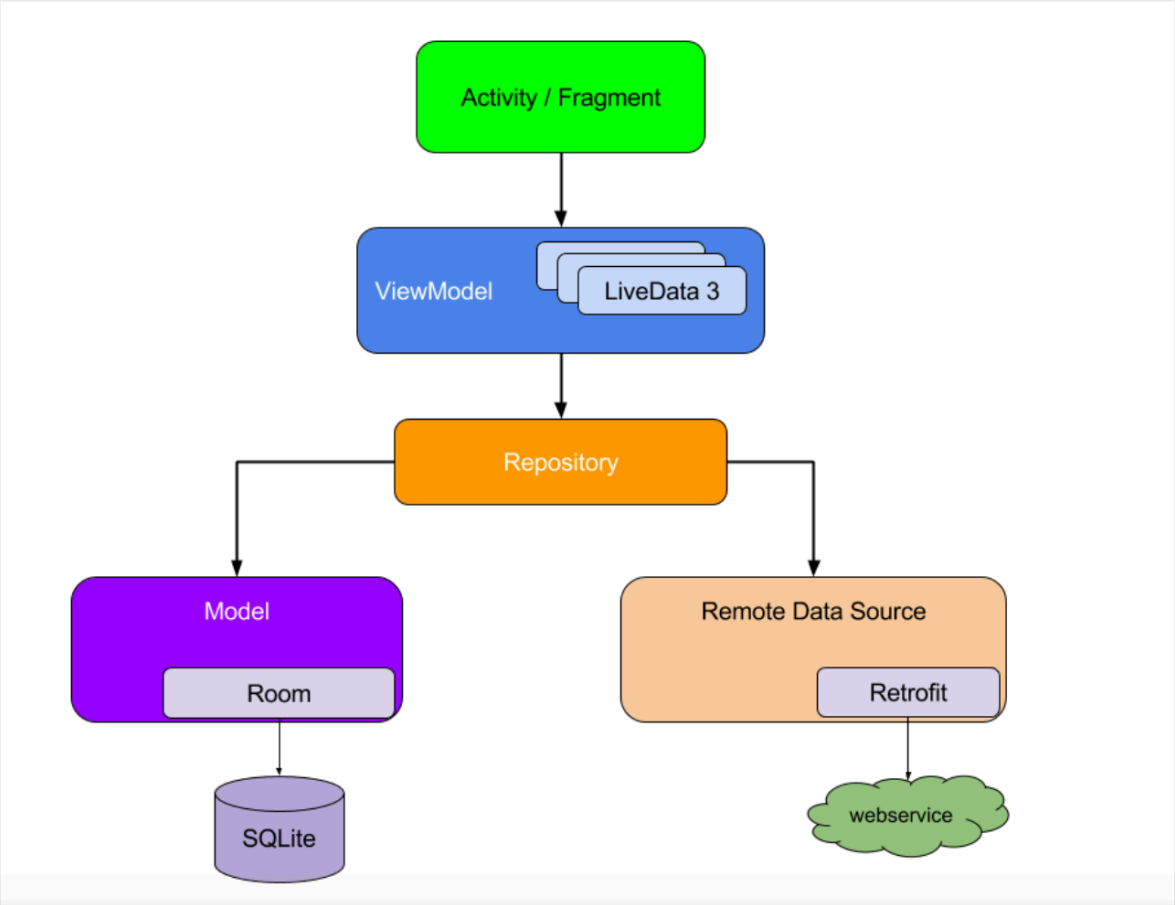
Old solution

Besides preventing the screen orientation changes in the manifest (by forcing only one orientation for each activity) or telling the system we want to handle this specific change ourselves (also in the manifest - android:configChanges="orientation" for each activity and overriding the onConfigurationChange methods inside the activities) - This are all ways to bypass the default system behavior. If we we leave the default system behavior that kills the activity, we should have overridden the **onSaveInstaceState** and **onRestoreInstanceState** in the activity and pass the information manually with the outgoing and incoming Bundle.

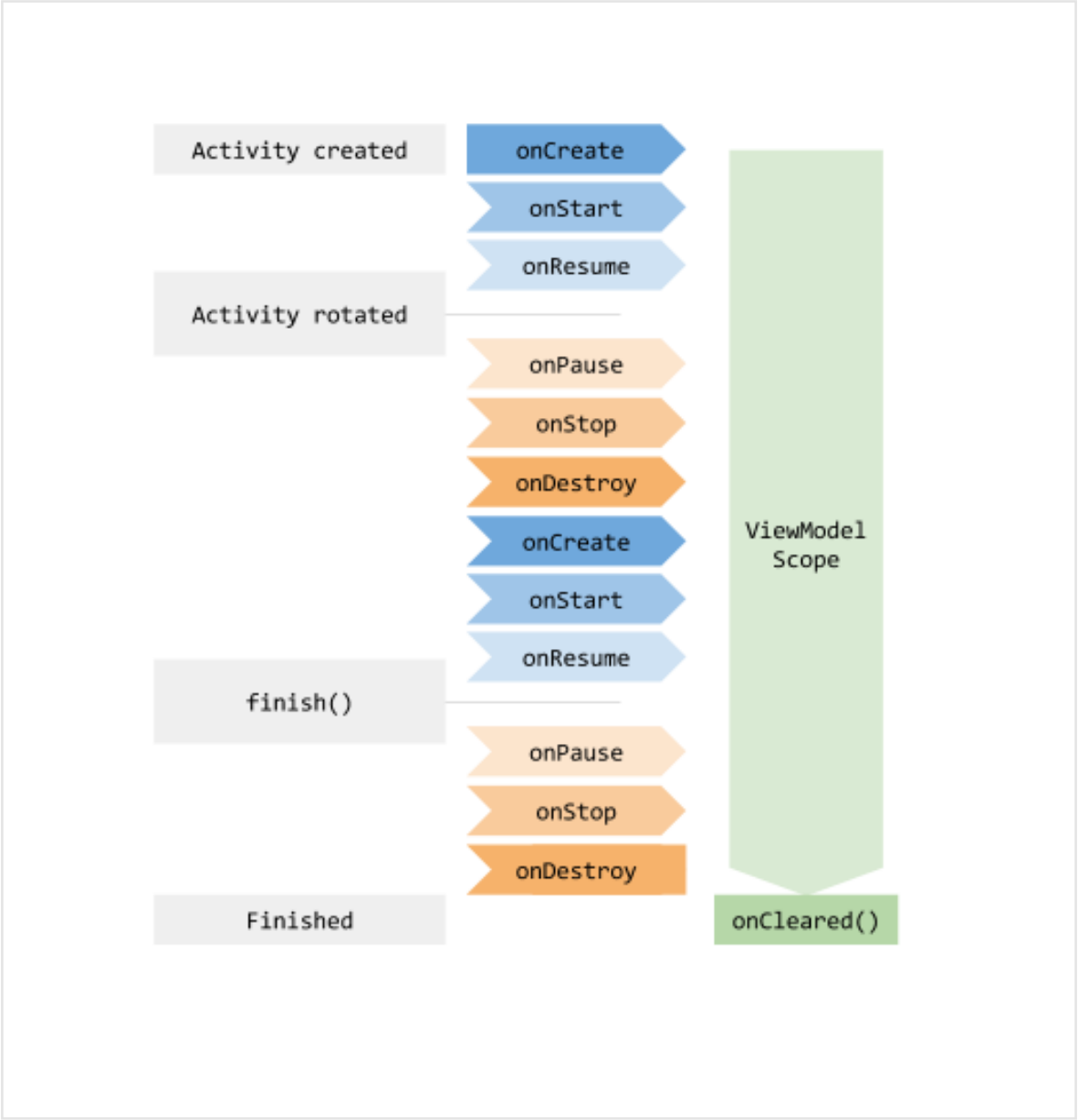
```
override fun onSaveInstanceState(outState: Bundle, outPersistentState: PersistableBundle) {
    super.onSaveInstanceState(outState, outPersistentState)
}

override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
}
```

New Solution MVVM - Model - View - ViewModel



Please look at the attached photo from the android developers:



As you can see a ViewModel is a special class that is aware of our lifecycle and can outlive our views (Activities and Fragments). Because of this, it can stay alive during configuration changes. This is the place where we save all of our data. Meaning the activity and fragments will serve only for UI purposes and will not hold any data. Instead, they will have a reference to their specific ViewModel and it will save the data for them and outlives their configuration changes. Only when the activity onDestroy() is called without onCreate() immediately after it then the viewModel onCleared() function is called and the instance holding our data is deallocated.

Our ViewModel is aware of our lifecycle when we pass LifeCycleOwner as us and with the help of the LiveData it will update us automatically on any change in it or when our new lifecycle event requires it.

Let's implement it:

1. Create a new class and inherit from the **ViewModel** class. Please note

you can also inherit from the **AndroidViewModel** in cases where the Context is needed for example when working with databases.

2. Now we remove all the data and the data related functions to that ViewModel: create a var property called result of type Double and initialize it to 0.0, add a custom setter that receive the value multiply it by the conversion rate and save the result in the field

```
class MainViewModel : ViewModel() {

    companion object {
        const val EURO_DOLLAR_RATE = 1.17
    }

    var result : Double = 0.0
        set(value) {
            field = value * EURO_DOLLAR_RATE
        }
}
```

3. Fragments and Activities needs to obtain a reference to the ViewModel in order to be able to access the model and observe data changes(later on). A Fragment or Activity maintains references to the ViewModels on which it relies for data using an instance of ViewModelProvider class. A ViewModelProvider instance is created via a call to the ViewModelProviders(owner) method from within the Fragment or Activity and pass the current Fragment or Activity as the lifecycle owners. It returns a ViewModelProvider instance. Once the ViewModelProvider instance has been created, the get() method can be called on that instance passing through the class of specific ViewModel that is required - the reflection class file but Use 'java' property to get Java class corresponding to this Kotlin class. The provider will then either create a new instance of that ViewModel class, or return an existing instance.
4. In the button click set the result field in your viewModel and read the updated value to the TextView. Please note that you also need to read it on the onCreate() in case of configuration change (don't worry, we will remove all of this when we use LiveData).
5. Run the app and rotate the screen - The amount saved in the view model and the activity is reading it in any new instance created!

Our complete code should look like this:

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding : ActivityMainBinding

    private  lateinit var viewModel : MainViewModel

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

        viewModel = ViewModelProvider( owner: this)[MainViewModel::class.java]

        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.button.setOnClickListener { it: View!

            viewModel.result = binding.dollarText.text.toString().toDouble()
            binding.resultText.text = viewModel.result.toString()
        }

        binding.resultText.text = viewModel.result.toString()

    }
}
```

Please Note you can also use the KTX extensions to initialize the ViewModel lazy.
Just add the:
implementation("androidx.activity:activity-ktx:1.4.0")

To your app Gradle file and write the following code:

```
private  val viewModel : MainViewModel by viewModels()

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

    //viewModel = ViewModelProvider(this)[MainViewModel::class.java]
```

If you want your view model in your Fragment:
implementation("androidx.fragment:fragment-ktx:1.4.0")

In conclusion this is how to get your View Model with the KTX in activity or fragment:

```
// Get a reference to the ViewModel scoped to this Fragment
val viewModel by viewModels<MyViewModel>()

// Get a reference to the ViewModel scoped to its Activity
val viewModel by activityViewModels<MyViewModel>()
```

Or like before you can mention the variable type and let the generic be inferred from that.

```
val sharedViewModel : MainViewModel by activityViewModels()

val viewModel : MainViewModel by viewModels()
```

LiveData & MutableLiveData

Thats all very nice but please note the repetitive code to read the data from the ViewModel.
If we use LiveData or MutableLiveData (if the contaned data can change) we can observe the return result and get notified automatically in two cases:
1. The inner value that the LiveData wraps changes.
2. A lifecycle event that requires refreshing the value occurred.

LiveData is a holder class which holds and updates the activity/fragment keeping in mind about their state. It uses special function called **observe** which will update activity/fragment instantly if anything changes in the LiveData. As LiveData class get the latest updated data but couldn't find activity or fragment, then they just hold the data and next time when activity or fragment is resumed the observer will fetch updated data itself and provide it to activity/fragment. For example, an activity that was in the background receives the latest data right after it returns to the foreground.
If an activity or fragment is recreated due to a configuration change, like device rotation, it immediately receives the latest available data. our observers are bounded to activity or fragment so they will be destroyed when the activity/fragment is destroyed. No need to handle it manually.

LiveData has some characteristics according to Google I/O 2017:

- LiveData is an observable data holder so it can be observed.
- Its lifecycle aware that prevents memory leakage in such a situation like configuration changes.
- LiveData automatically manages subscriptions. If you are observing a liveData you don't need to unsubscribe. The right things will happen in the right times.
- Doesn't matter how many observers you have or what state they are, all of it are merged into one lifecycle.
- It doesn't have any activity or fragment inside it but it works with both

of them.
- Also liveData makes testing easy because it's kind of Android free(it can be tested with our device).
- The LiveData instance is doing all the fetching and updating work on the Dispatchers.IO and not on the main thread.

Adding LiveData

1. In the ViewModel class replace the type of the system result from Double to MutableLiveData<Double> - it is mutable since the inside value - the double can change. Remove the get and set and create a new function called setValue that receives the new Double value and update the value field of the LiveData with the converted amount

```
const val EURO_DOLLAR_RATE = 1.17

class MainViewModel : ViewModel() {

    val result = MutableLiveData<Double>()

    fun setValue(value: Double) {
            result.value = value * EURO_DOLLAR_RATE
    }
```

2. In the MainActivity remove all of the result Text View updates. Remove also the viewModel update from before. Now in the onCreate() set an Observer to the ViewModel's LiveData field, passing it the activity as the lifecycle owner (for all the reasons mentioned above) and in the callback add the one and only result Text View update.

```
viewModel.result.observe( owner: this) {  it: Double!
    binding.resultText.text = it.toString()
}
```

3. Now in the onClick just call the the ViewModel setValue function passing it the user dollar value and Thats it - The LiveData will do the rest!

```
binding.button.setOnClickListener { it: View!

    viewModel.setValue(binding.dollarText.text.toString().toDouble())

    //binding.resultText.text = "${binding.dollarText.text.toString().toD
    // viewModel.result = binding.dollarText.text.toString().toDouble()
    // binding.resultText.text = viewModel.result.toString()
}
```

ViewModel to Communicate between fragments and Their hosting activity

ViewModel is an ideal choice when you need to share data between multiple fragments or between fragments and their hosting Activity.

Lets look at this ItemViewModel which will be shared by both the Activity and its hosted Fragment:

```
class ItemViewModel : ViewModel() {
    private val mutableSelectedItem = MutableLiveData<Item>()
    val selectedItem: LiveData<Item> get() = mutableSelectedItem

    fun selectItem(item: Item) {
        mutableSelectedItem.value = item
    }
}
```

Please note that while the actual stored value is MutableLiveData the get only return LiveData this ensures consistency of our data.

Both your fragment and its host activity can retrieve a shared instance of a ViewModel with activity scope by passing the activity into the ViewModelProvider constructor. The ViewModelProvider handles instantiating the ViewModel or retrieving it if it already exists. Both components can observe and modify this data (in this example we use the KTX extensions library to get a delegate that initial their view model lazy):

```
class MainActivity : AppCompatActivity() {
    // Using the viewModels() Kotlin property delegate from the activity-ktx
    // artifact to retrieve the ViewModel in the activity scope
    private val viewModel: ItemViewModel by viewModels()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        viewModel.selectedItem.observe(this, Observer { item ->
            // Perform an action with the latest item data
        })
    }
}

class ListFragment : Fragment() {
    // Using the activityViewModels() Kotlin property delegate from the
    // fragment-ktx artifact to retrieve the ViewModel in the activity scope
    private val viewModel: ItemViewModel by activityViewModels()

    // Called when the item is clicked
    fun onItemClicked(item: Item) {
        // Set a new item
        viewModel.selectItem(item)
    }
}
```

Share data between fragments

Two or more fragments in the same activity often need to communicate with each other. For example, imagine one fragment that displays a list and another that allows the user to apply various filters to the list.

These fragments can share a ViewModel using their activity scope to handle this communication. By sharing the ViewModel in this way, the fragments do not need to know about each other, and the activity does not need to do anything to facilitate the communication.

```
class ListFragment : Fragment() {
    // Using the activityViewModels() Kotlin property delegate from the
    // fragment-ktx artifact to retrieve the ViewModel in the activity scope
    private val viewModel: ListViewModel by activityViewModels()
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        viewModel.filteredList.observe(viewLifecycleOwner, Observer { list ->
            // Update the list UI
        }
    }
}

class FilterFragment : Fragment() {
    private val viewModel: ListViewModel by activityViewModels()
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        viewModel.filters.observe(viewLifecycleOwner, Observer { set ->
            // Update the selected filters UI
        }
    }
```

Notice that both fragments use their host activity as the scope for the ViewModelProvider. Because the fragments use the same scope, they receive the same instance of the ViewModel, which enables them to communicate back and forth.

Saved State module for ViewModel

As mentioned before View Model can survive configuration changes and store our data. Before using it we used to pass our data through savedInstanceState Bundle. We can still use the onSavednstanceState() as a backup in case our recreation comes from system-initiated process death. In that case our View Models will be killed also.

But onSavednstanceState() function is from the Activity where the ViewModel actually the one that stores or remembers the UI state so that can cause allot of boilerplate code. To solve this the view model has its own Bundle that can store data between sessions.

All you have to do is to get the SaveStateHandle in you ViewModel's constructor (We will see later on that it also has a default binding)

```
class SavedStateViewModel(private val state: SavedStateHandle) : ViewModel() {
```

Don't worry you don't have to do any additional configuration because the default ViewModel factory provides the appropriate SavedStateHandle to your ViewModel. So just go ahead and retrieve your View model like you did before

```
class MainFragment : Fragment() {
    val vm: SavedStateViewModel by viewModels()

    ...
}
```

The SavedStateHandle class is a key-value map that allows you to write and retrieve data to and from the saved state through the set() and get() methods. Additionally, you can retrieve values from SavedStateHandle that are wrapped in a LiveData observable using getLiveData(). When the key's value is updated, the LiveData receives the new value. Most often, the value is set due to user interactions, such as entering a query to filter a list of data. This updated value can then be used to transform LiveData.

```
class SavedStateViewModel(private val savedStateHandle: SavedStateHandle) : ViewModel() {
    val filteredData: LiveData<List<String>> =
        savedStateHandle.getLiveData<String>("query").switchMap { query ->
        repository.getFilteredData(query)
    }

    fun setQuery(query: String) {
        savedStateHandle["query"] = query
    }
}
```

By using SavedStateHandle, the query value is retained across **process death**, ensuring that the user sees the same set of filtered data before and after recreation without the activity or fragment needing to manually save, restore, and forward that value back to the ViewModel.

Here is a simple example on how to save the current user in SaveStateHandle

```
class MyViewModel(state : SavedStateHandle) : ViewModel() {

    // Keep the key as a constant
    companion object {
        private val USER_KEY = "userId"
    }

    private val savedStateHandle = state

    fun saveCurrentUser(userId: String) {
        // Sets a new value for the object associated to the key.
        savedStateHandle.set(USER_KEY, userId)
    }

    fun getCurrentUser(): String {
        // Gets the current value of the user id from the saved state handle
        return savedStateHandle.get(USER_KEY)?: ""
    }
}
```

Usually you will use LiveData in your ViewModel. For that you can use the SavedStateHandle.getLiveData() method. Here's an example of replacing getCurrentUser with a LiveData, which allows for observation:

```
// getLiveData gets MutableLiveData associated with a key.
// When the value associated with the key updates, the MutableLiveData does as well.
private val _userId : MutableLiveData<String> = savedStateHandle.getLiveData(USER_KEY)

// Only expose a immutable LiveData
val userId : LiveData<String> = _userId
```

SavedStateHandle also has other methods you might expect when interacting with a key-value map:

- contains(String key) - Checks if there is a value for the given key.
- remove(String key) - Removes the value for the given key.
- keys() - Returns all keys contained within the SavedStateHandle.

For supported types please refer to https://developer.android.com/topic/libraries/architecture/viewmodel-savedstate#types

Transform LiveData

You may want to make changes to the value stored in a LiveData object before dispatching it to the observers, or you may need to return a different LiveData instance based on the value of another one. The Lifecycle package provides the Transformations class which includes helper methods that support these

scenarios.

In both **map** and **switchMap** there is a **source** (or trigger) live data, and in both cases you want to **transform** it to another **live data**. Which one will you use - depends on the task that your transformation is doing.

Map() is conceptually identical to the use in RXJava, basically you are changing a parameter of LiveData in another one
SwitchMap() instead you are going to substitute the LiveData itself with another one! Typical case is when you retrieve some data from a Repository for instance and to "eliminate" the previous LiveData (to garbage collect, to make it more efficient the memory usually) you pass a **new** LiveData that execute the same action(getting a query for instance)

To understand that difference Let's take an example, there is a LiveData which emits a string and we want to display that string in capital letters:

With map (in activity or fragment)

```
Transformations.map(stringsLiveData, String::toUpperCase)
    .observe(this, textView::setText);
```

the function passed to the map returns a string only, but it's the Transformation#map which ultimately returns a LiveData.

With SwitchMap (also in activity or fragment)

```
Transformations.switchMap(stringsLiveData, this::getUpperCaseStringLiveData)
            .observe(this, textView::setText);

private LiveData<String> getUpperCaseStringLiveData(String str) {
    MutableLiveData<String> liveData = new MutableLiveData<>();
    liveData.setValue(str.toUpperCase());
    return liveData;
}
```

If you see Transformations#switchMap has actually switched the LiveData. So, again as per the documentation **The function passed to switchMap() must return a LiveData object**.

So, in case of map it is the **source** LiveData you are transforming and in case of switchMap the passed LiveData will act as a **trigger** on which it will switch to another LiveData after unwrapping and dispatching the result downstream.

Transformation are useful because they are computed lazily (meaning that they are not calculated unless someone is observing their retuned LiveData) that's why they goes well with the observer's lifecycle without any additional configuration.

They are very useful in case where a change in one object should return another one. For example if we have a UI component that gets and address and return postal code, then the we must register to a LiveData returned from our repository. We can do it like this:

```
class MyViewModel(private val repository: PostalCodeRepository) : ViewModel()

    private fun getPostalCode(address: String): LiveData<String> {
        // DON'T DO THIS
        return repository.getPostCode(address)
    }
}
```

Which is not a good idea for two reasons the first is that each time the activity or fragment is recreated and we are doing a new database fetch because we don't store the old value but rather fetching it all over again and the second one is that each time he calls this function he is actually registering a new Live data which is costly.

What we should be doing in that case is:

```
class MyViewModel(private val repository: PostalCodeRepository) : ViewModel()
    private val addressInput = MutableLiveData<String>()
    val postalCode: LiveData<String> = Transformations.switchMap(addressInput)
            address -> repository.getPostCode(address) }


    private fun setInput(address: String) {
        addressInput.value = address
    }
}
```

In this case, the postalCode field is defined as a transformation of the addressInput. As long as your app has an active observer associated with the postalCode field, the field's value is recalculated and retrieved whenever addressInput changes and that's it, no extra calculations are done.

https://medium.com/androiddevelopers/viewmodels-with-saved-state-jetpack-navigation-data-binding-and-coroutines-df476b78144e

ViewModel NavGraph Integration

Before we saw that we can share information between fragments and their hosting activity using the shared View Model that we can access from all fragments:

```
// Any fragment's onCreate or onActivityCreated
// This ktx requires at least androidx.fragment:fragment-ktx:1.1.0
val sharedViewModel: ActivityViewModel by activityViewModels()
```

But what can we do if we want a shared view Model by some of the fragments and not all of them, While they all share the same Activity?

The solution is to this is to create a **nested navigation graph and share a view model to that graph**

The new Navigation API introduces ViewModels associated to a Navigation Graph. In practice, this means you can take a collection of associated destinations, such as an onboarding flow, a login flow, or a checkout flow; put them into a nested navigation graph; and enable shared data just between those screens.

To create a nested navigation graph, you can select your screens, right click, and select **Move to Nested Graph → New Graph:**

In the XML view, note the **id** of the nested navigation graph, in this case checkout_graph:

```
<navigation app:startDestination="@id/homeFragment" ...>
    <fragment android:id="@+id/homeFragment" .../>
    <fragment android:id="@+id/productListFragment" .../>
    <fragment android:id="@+id/productFragment" .../>
    <fragment android:id="@+id/bargainFragment" .../>

    <navigation
        android:id="@+id/checkout_graph"
        app:startDestination="@id/cartFragment">

        <fragment android:id="@+id/orderSummaryFragment".../>
        <fragment android:id="@+id/addressFragment" .../>
        <fragment android:id="@+id/paymentFragment" .../>
        <fragment android:id="@+id/cartFragment" .../>

    </navigation>

</navigation>
```

Once you've done this, you get the ViewModel using by navGraphViewModels:

```
val viewModel: CheckoutViewModel by navGraphViewModels(R.id.checkout_graph)
```

But don't forget to add the KTX dependency:
implementation 'androidx.navigation:navigation-fragment-ktx:2.4.2'

To check it in our project simply a create a ViewModel with a single int property

```
class NavGraphViewModel : ViewModel() {

    val x = 9
}
```

In each of the fragments get a reference to it by using the navGraphViewModel and supply it with your root navigation graph id and simply Toast the value stored in your view model

```
class SignUpFragment : Fragment() {

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

    private val viewModel : NavGraphViewModel by navGraphViewModels(R.id.my_nav)

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {

        Toast.makeText(requireContext(), text: "${viewModel.x}",Toast.LENGTH_SHORT).show()
```

```
class FinishFragment() : Fragment() {

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

    private val viewModel : NavGraphViewModel by navGraphViewModels(R.id.my_nav)

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {

        Toast.makeText(requireContext(), text: "${viewModel.x}", Toast.LENGTH_SHORT).show()
```

ViewModelProvider Factory

Sometimes we want to initiate our view model and pass parameters to his constructor so that we can use it's init function for data fetching request for example.

We will see that Hilt can inject necessary components such as a repository that handles network and database requests. This is the preferred way, And it will be show in the Hilt Model.

However, if you don't want to use Hilt or simply Dagger you can inherit from the NewInstanceFactory which implement the ViewmodelProvider.Factory interface and override it's create() function. Pass your arguments to your factory implementation and in it call you view model constructor with the parameters and return your constructor initiated view model. Pass your implementation to the ViemodelProvider constructor which also gets a factory method besides the lifecycle owner and That's it!

Let's say we want to pass this view model a string so he can use it in his init function to initiate a database fetch

```
class ParamConstructedViewModel(var str:String): ViewModel() {

    init {
        //TODO: Use 'str' to init process when VM is created. i.e. Get data request.
        str = "$str dsdfs"
    }
}
```

Now we need to create his Factory implementation (we pass the factory params to our view model constructor):

```
// Override ViewModelProvider.NewInstanceFactory to create the ViewModel (VM).
class ParamConstructedModelFactory(private var str: String)
    : ViewModelProvider.NewInstanceFactory() {

    override fun <T : ViewModel?>
            create(modelClass: Class<T>): T = ParamConstructedViewModel(str) as T
}
```

And in the fragment or activity we can do this:

```
private lateinit var viewModel : ParamConstructedViewModel

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

    viewModel = ViewModelProvider( owner: this,
        ParamConstructedModelFactory( str: "another string"))[ParamConstructedViewModel::class.java]

    Toast.makeText( context: this,viewModel.str,Toast.LENGTH_SHORT).show()
```

Or use the KTX lazy delegate:

```
class MainActivity : AppCompatActivity() {

    private val viewModel : ParamConstructedViewModel by viewModels {
        ParamConstructedModelFactory( str: "some string")
    }
```