[LiveData](#_e12gjnr2eaxo)

[Transformations of LiveData](#_bcanofhgskn8)

[Creating new transformations.](#_2ef8i644756e)

## LiveData

LiveData is a data holder class which keeps a value and allows this value to be observed.

Unlike a regular observable, LiveData respects the lifecycle of App Components, such that the Observer can specify a Lifecycle in which it should observe.

LiveData considers an Observer in *active* state if the Observer’s Lifecycle is in STARTED or RESUMED state.

|  |
| --- |
| public class LocationLiveData extends LiveData<Location> {  private LocationManager locationManager;   private SimpleLocationListener listener = new SimpleLocationListener() {  @Override  public void onLocationChanged(Location location) {  setValue(location);  }  };   public LocationLiveData(Context context) {  locationManager = (LocationManager) context.getSystemService(  Context.LOCATION\_SERVICE);  }   @Override  protected void onActive() {  locationManager.requestLocationUpdates(LocationManager.GPS\_PROVIDER, 0, 0, mListener);  }   @Override  protected void onInactive() {  locationManager.removeUpdates(listener);  } } |

There are 3 important parts in this implementation of the Location listener:

* **onActive:** This method is called when the LiveData has an active observer. This means we need to start observing the location updates from the device.
* **onInactive:** This method is called when the LiveData does not have any active observers. Since no observers are listening, there is no reason to stay connected to the LocationManager service. This is important because staying connected will consume significant battery power without any benefit.
* **setValue:** Calling this method updates the value of the LiveData instance and notifies active observers about the change.

You might notice that we don’t have the **Callback** field, and we don’t check the lifecycle of the observer manually anymore. LiveData takes care of these.

We can use the new MyLocationListener as follows:

|  |
| --- |
| public class MyFragment extends LifecycleFragment {  public void onActivityCreated (Bundle savedInstanceState) {  LiveData<Location> myLocationListener = ...;  Util.checkUserStatus(result -> {  if (result) {  myLocationListener.addObserver(getLifecycle(), location -> {  // update UI  });  }  });  } } |

Notice that the *addObserver* method is passed the Lifecycle as the first argument. Doing so denotes that this observer should be bound to that lifecycle, meaning:

* If the Lifecycle is not in active state (STARTED or RESUMED), the observer will not be called even if the value changes.
* If the Lifecycle is destroyed, the observer will be removed automatically.

The fact that LiveData is lifecycle-aware provides us a new opportunity: We can share it between multiple activities, fragments etc. To keep our example simple, we can make it singleton as follows:

|  |
| --- |
| public class LocationLiveData extends LiveData<Location> {  private state LocationLiveData sInstance;  private LocationManager locationManager;   @MainThread  public static LocationLiveData get(Context context) {  if (sInstance == null) {  sInstance = new LocationLiveData(context.getApplicationContext());  }  return sInstance;  }   private SimpleLocationListener listener = new SimpleLocationListener() {  @Override  public void onLocationChanged(Location location) {  setValue(location);  }  };   private LocationLiveData(Context context) {  locationManager = (LocationManager) context.getSystemService(  Context.LOCATION\_SERVICE);  }   @Override  protected void onActive() {  locationManager.requestLocationUpdates(LocationManager.GPS\_PROVIDER, 0, 0, mListener);  }   @Override  protected void onInactive() {  locationManager.removeUpdates(listener);  } } |

Now the Fragment can use it as follows:

|  |
| --- |
| public class MyFragment extends LifecycleFragment {  public void onActivityCreated (Bundle savedInstanceState) {  Util.checkUserStatus(result -> {  if (result) {  MyLocationListener.get(getActivity())  .addObserver(getLifecycle(), location -> {  // update UI  });  }  });  } } |

There might be multiple fragments / activities that are observing our *MyLocationListener* instance, and *LiveData* will gracefully manage them such that it will only connect to the system service if any of them is visible (a.k.a active).

Below is a list of advantages provided by the LiveData class:

* No memory leaks: Since the Observers are bound to their own Lifecycles, they will be automatically cleaned when their Lifecycle is destroyed.
* No crashes due to stopped Activities: If the Observer’s Lifecycle is inactive (like an Activity in the back stack), they won’t receive change events.
* Always up to date data: If a Lifecycle starts again (like an Activity going back to started state from back stack) it will receive the latest Location data (if it didn’t already).
* Proper configuration change: If an Activity / Fragment is re-created due to a configuration change (like device rotation), they will instantly receive the last available location data.
* Sharing Resources: Now we can keep a single instance of MyLocationListener, connect to the system service just once and properly support all observers in the Application.
* No More Manual Lifecycle Handling: As you might notice, our Fragment just observes the data when it wants to, does not worry about being stopped or start observing after being stopped. All of this is automatically managed by LiveData since the Fragment provided its Lifecycle while observing.

### Transformations of LiveData

Sometimes, you may want to make changes on the LiveData value 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 a Transformations class which includes helper methods for these operations.

* Transformations.map():

Applies a function on the LiveData value, and dispatches the result downstream.

|  |
| --- |
| LiveData<User> userLiveData = ...; LiveData<String> userName = Transformations.map(userLiveData, user -> {  user.name + " " + user.lastName }); |

* Transformations.switchMap():

Similar to **map()**, applies a function to the value and unwraps and dispatches the result downstream. The function passed into switchMap must return a **LiveData**.

|  |
| --- |
| private LiveData<User> getUser(String id) {  ...; }  LiveData<String> userId = ...; LiveData<User> user = Transformations.switchMap(userId, id -> getUser(id) ); |

Using these transformations allows carrying over the observer Lifecycle information across the chain such that these transformations will not be calculated unless the returned LiveData is observed by an observer. This lazy calculation nature of transformations allows implicitly passing down lifecycle-related behavior without adding explicit calls or dependencies.

Whenever you think you need a Lifecycle object inside a ViewModel, a transformation is probably the solution.

For instance, let’s assume that we have a UI where the user puts in an address and they receive the postal code for that address.

The naive ViewModel for this UI could be like this:

|  |
| --- |
| class MyViewModel extends ViewModel {  private final PostalCodeRepository repository;  public MyViewModel(PostalCodeRepository repository) {  this.repository = repository;  }   private LiveData<String> getPostalCode(String address) {  // DON'T DO THIS  return repository.getPostCode(address);  } } |

If this is the implementation, the UI would need to unregister from the previous LiveData and re-register to the new one each time they call **getPostalCode**. Moreover, if the UI is re-created, it triggers another call to **repository.getPostCode** instead of using the previous call’s result.

Instead of that, the postal code information can be implemented as a transformation of the address input.

|  |
| --- |
| class MyViewModel extends ViewModel {  private final PostalCodeRepository repository;  private final MutableLiveData<String> addressInput = new MutableLiveData();  public final LiveData<String> postalCode = Transformations.switchMap(addressInput, (address) -> {  return repository.getPostCode(address);  });   public MyViewModel(PostalCodeRepository repository) {  this.repository = repository  }   private void setInput(String address) {  addressInput.setValue(address);  } } |

Notice that we’ve even made the **postalCode** field **public final** because it will never change.

It is defined as a transformation of the addressInput such that, when addressInput changes, **if there is an active observer**, *repository.getPostCode()* will be called. If there are no active observers at the time of the call, no calculations will be made until an observer is added.

This mechanism allows lower levels of the application to create LiveData objects that will lazily be calculated on demand. ViewModel can easily obtain them and define transformation rules on top of them

### Creating new transformations

There are a dozen different specific transformation that may be useful in your application, but they aren’t provided by default. To implement your own transformation you can you use MediatorLiveData class, which is specially created to properly listen to other LiveData instances and process events emitted by them. MediatorLiveData takes cares to correctly propagate its active/inactive states to the source LiveData. You can check the implementation of the Transformations class for details.