# Handling Lifecycles in a Jetpack way

Han Yin @ *Oct, 2018*

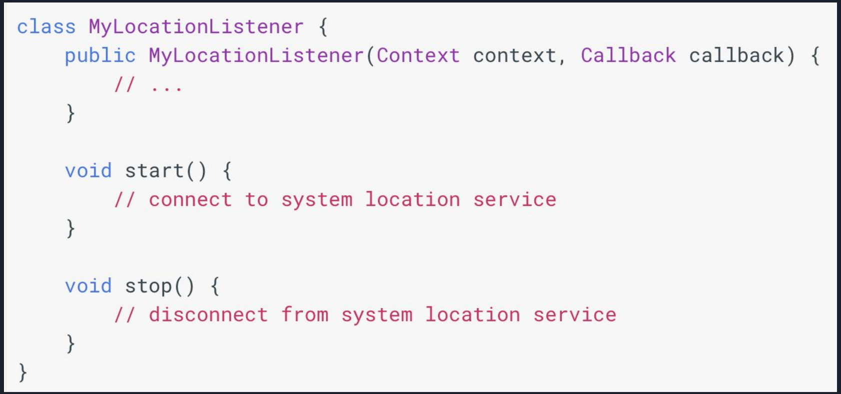
## TOC

## Problem

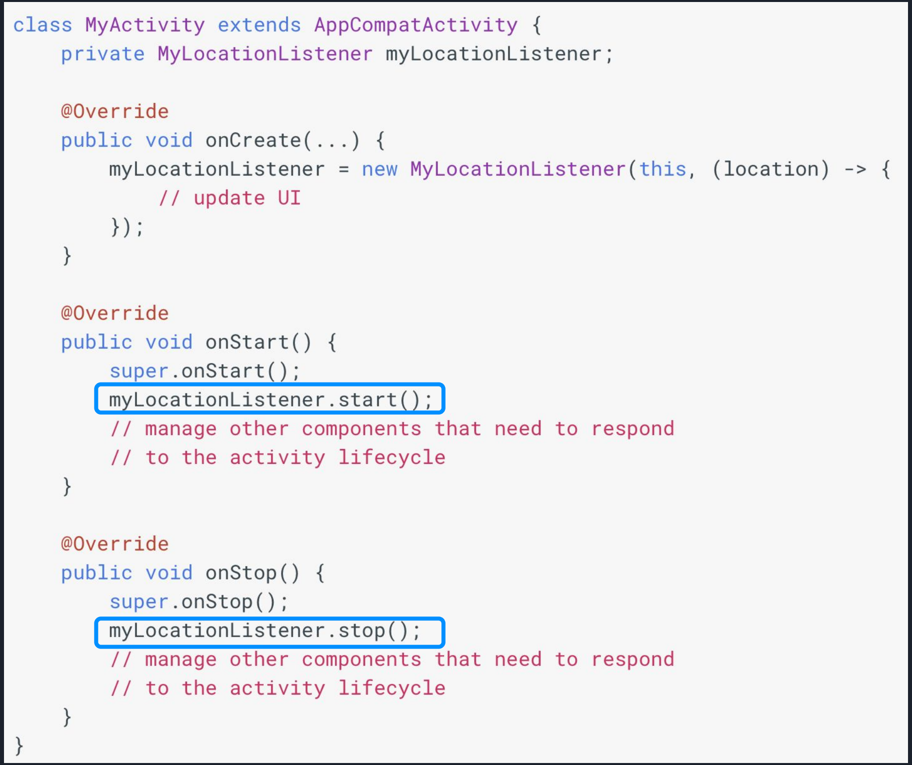
- A common pattern to perform actions in response to a change in the lifecycle status of another component (e.g. activities, fragments):
  - Implement the actions of the dependent components in the lifecycle methods of activities and fragments.
  - Or some custom callbacks, e.g. handleFooBarLoaded()
- However, this pattern leads to a poor organization of the code and to the proliferation of errors.

# Problem

- Sample of the common approach:
  - Start or stop a component in *onStart()* and *onStop()*



```java
class MyLocationListener {
    public MyLocationListener(Context context, Callback callback) {
        // ...
    }

    void start() {
        // connect to system location service
    }

    void stop() {
        // disconnect from system location service
    }
}
```



```java
class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

    @Override
    public void onCreate(...) {
        myLocationListener = new MyLocationListener(this, (location) -> {
            // update UI
        });
    }

    @Override
    public void onStart() {
        super.onStart();
        myLocationListener.start();
        // manage other components that need to respond
        // to the activity lifecycle
    }

    @Override
    public void onStop() {
        super.onStop();
        myLocationListener.stop();
        // manage other components that need to respond
        // to the activity lifecycle
    }
}
```

# Problem

- Might cause a *race hazard*:
  - The asynchronous call returned, but the activity is already stopped
  - Hold a reference and manually check

```java
class MyLocationListener {
    public MyLocationListener(Context context, Callback callback) {
        // ...
    }

    void start() {
        // connect to system location service
    }

    void stop() {
        // disconnect from system location service
    }
}
```

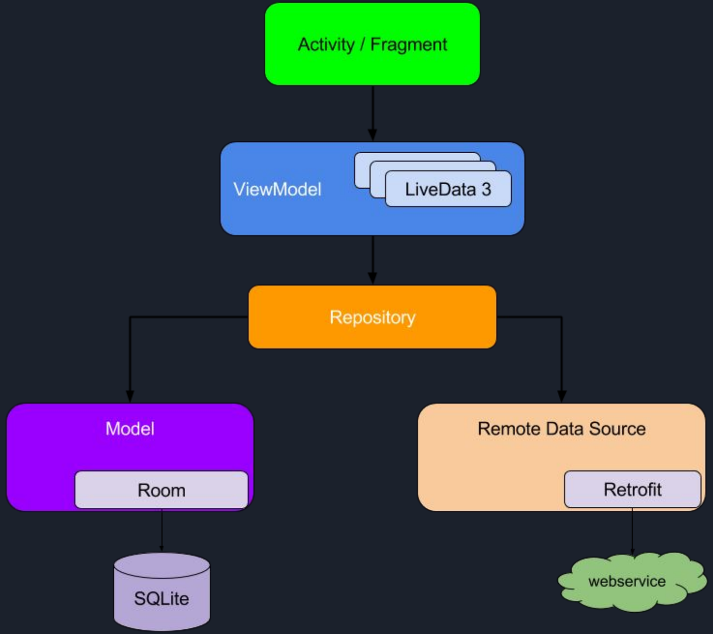
```java
class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

    public void onCreate(...) {
        myLocationListener = new MyLocationListener(this, location -> {
            // update UI
        });
    }

    @Override
    public void onStart() {
        super.onStart();
        Util.checkUserStatus(result -> {
            // what if this callback is invoked AFTER activity is stopped?
            if (result) {
                myLocationListener.start();
            }
        });
    }

    @Override
    public void onStop() {
        super.onStop();
        myLocationListener.stop();
    }
}
```

## Solution

- By using lifecycle-aware components, you can move the code of dependent components out of the lifecycle methods and into the components themselves.
  - *Lifecycle*
    - A class that holds the information about the lifecycle state of a component (e.g. activity, fragment) and allows other objects to observe this state.
  - *ViewModel*
    - Store and manage UI-related data in a lifecycle conscious way, allows data to survive configuration changes such as screen rotations.
  - *LiveData*
    - An observable data holder class, only updates app component observers that are in an active lifecycle state.

# Solution

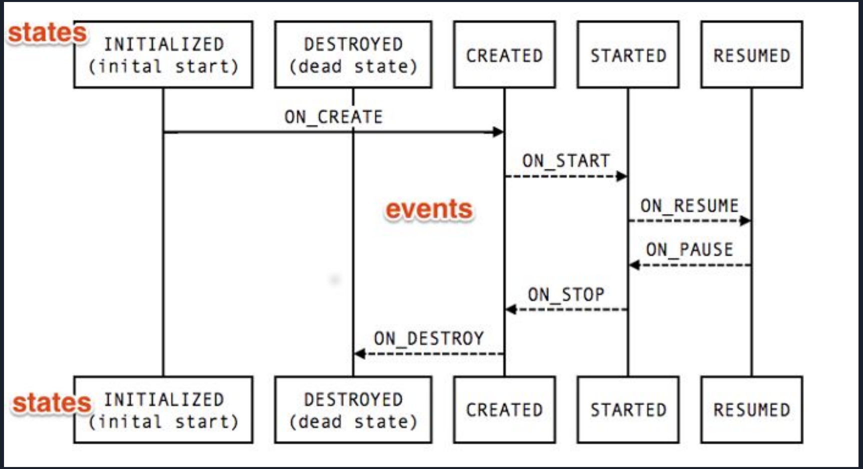
- Diagram of the recommended app architecture in *Guide to app architecture*:



## Lifecycle

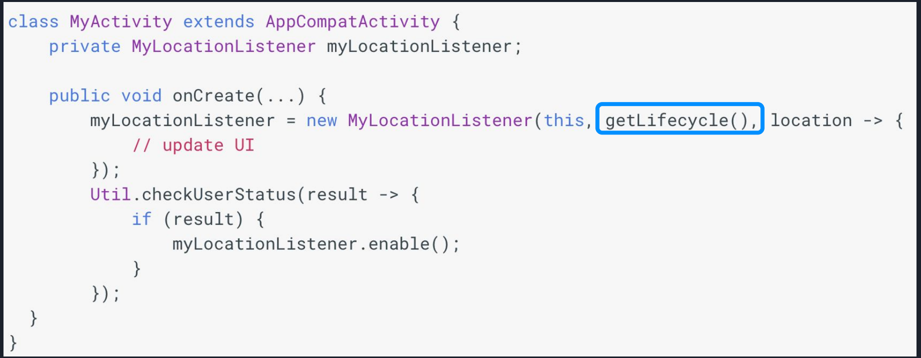
- *Event*
  - The lifecycle events dispatched from the framework and the *Lifecycle* class.
  - These events map to the callback events in activities and fragments.
- *State*
  - The current state of the component tracked by the *Lifecycle* object.

- Think of the *states* as nodes of a graph, and *events* as the edges between these nodes.

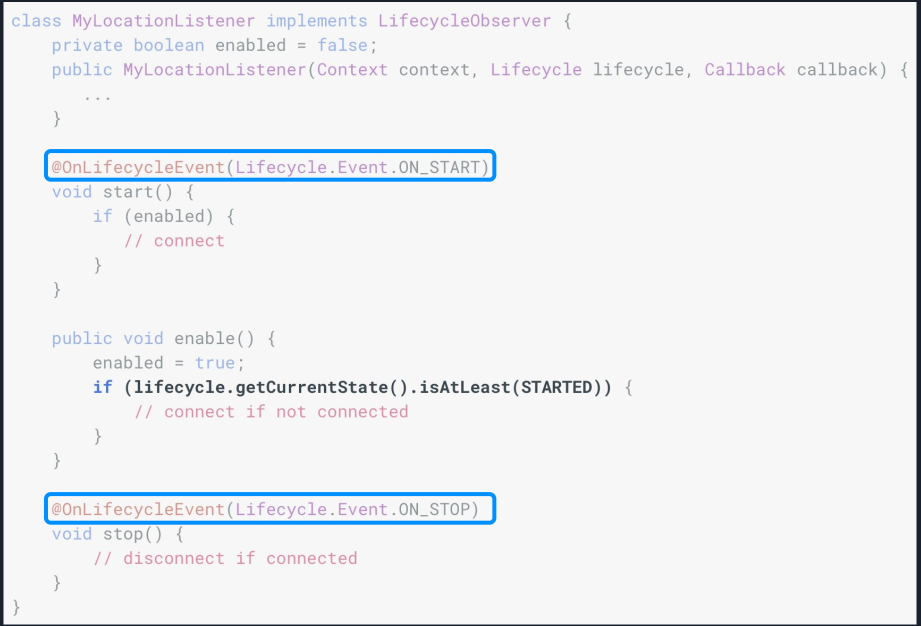


# Lifecycle

- *LifecycleOwner*
  - A single method interface that denotes that the class has a Lifecycle: *getLifecycle()*
- *LifecycleObserver*
  - Does not have any methods.
  - Relies on *OnLifecycleEvent* annotated methods.



```
class MyActivity extends AppCompatActivity {
    private MyLocationListener myLocationListener;

    public void onCreate(...) {
        myLocationListener = new MyLocationListener(this, getLifecycle(), location -> {
            // update UI
        });
        Util.checkUserStatus(result -> {
            if (result) {
                myLocationListener.enable();
            }
        });
    }
}
```



```
class MyLocationListener implements LifecycleObserver {
    private boolean enabled = false;
    public MyLocationListener(Context context, Lifecycle lifecycle, Callback callback) {
        ...
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    void start() {
        if (enabled) {
           // connect
        }
    }

    public void enable() {
        enabled = true;
        if (lifecycle.getCurrentState().isAtLeast(STARTED)) {
            // connect if not connected
        }
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    void stop() {
        // disconnect if connected
    }
}
```

## ViewModel

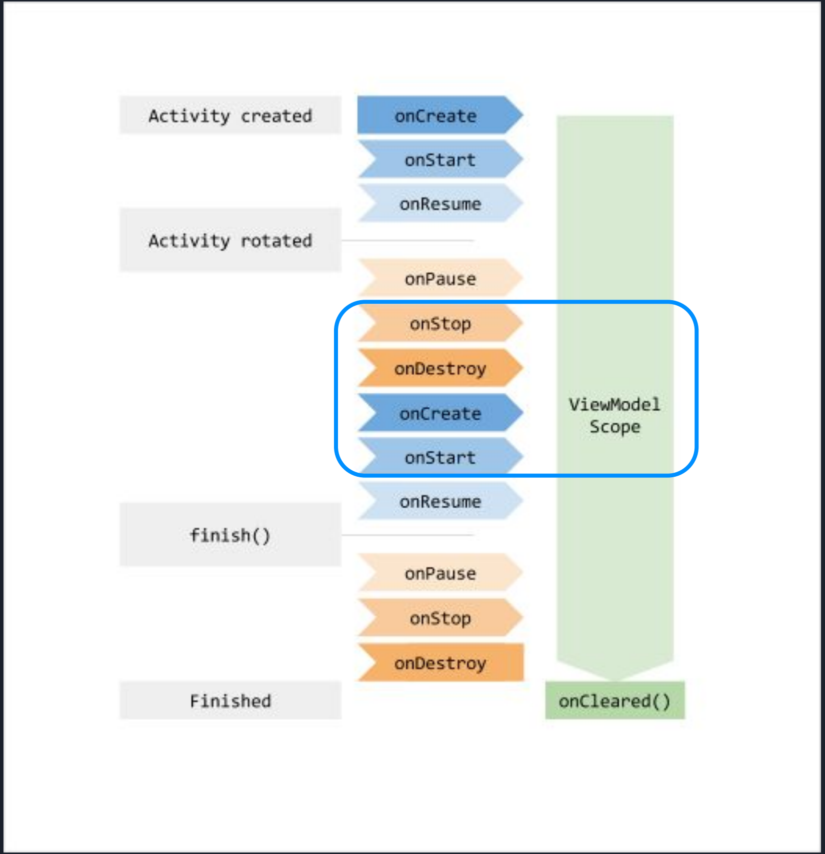
- If the activity is re-created, it receives the **<u>same</u>** *MyViewModel* instance that was created by the first activity.
- When the owner activity is finished, the framework calls the ViewModel objects's *onCleared()* method, so that it can clean up resources.

```java
public class MyViewModel extends ViewModel {
    private MutableLiveData<List<User>> users;
    public LiveData<List<User>> getUsers() {
        if (users == null) {
            users = new MutableLiveData<List<User>>();
            loadUsers();
        }
        return users;
    }

    private void loadUsers() {
        // Do an asynchronous operation to fetch users.
    }
}
```

```java
public class MyActivity extends AppCompatActivity {
    public void onCreate(Bundle savedInstanceState) {
        // Create a ViewModel the first time the system calls an activity's onCreate() method.
        // Re-created activities receive the same MyViewModel instance created by the first activity.

        MyViewModel model = ViewModelProviders.of(this).get(MyViewModel.class);
        model.getUsers().observe(this, users -> {
            // update UI
        });
    }
}
```

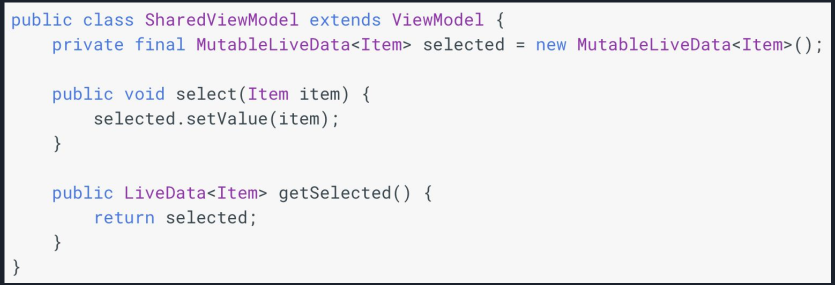
## ViewModel

- If the activity is re-created, it receives the **<u>same</u>** *MyViewModel* instance that was created by the first activity.
- When the owner activity is finished, the framework calls the ViewModel objects's *onCleared()* method, so that it can clean up resources.

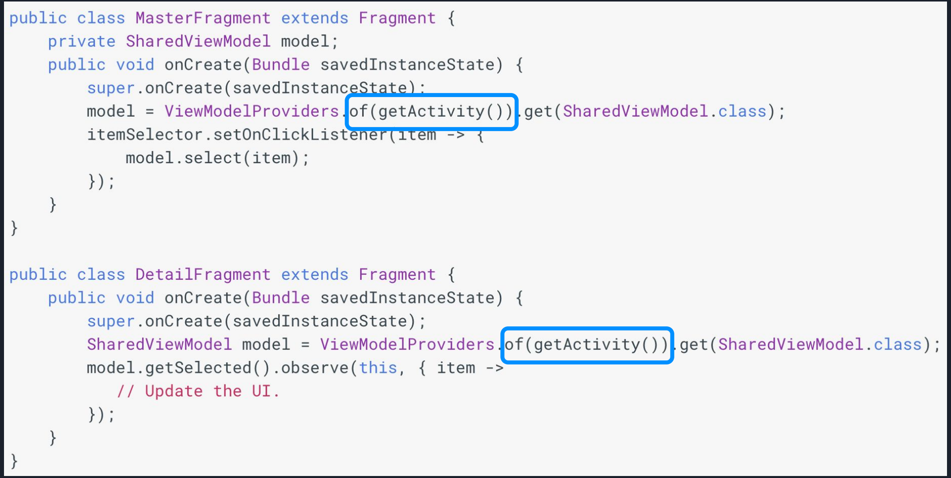


# ViewModel

- A Sample of sharing data between master-detail fragments
  - Share the same *ViewModel* by specifying their activity scope.
  - Avoid accessing the activity’s field.



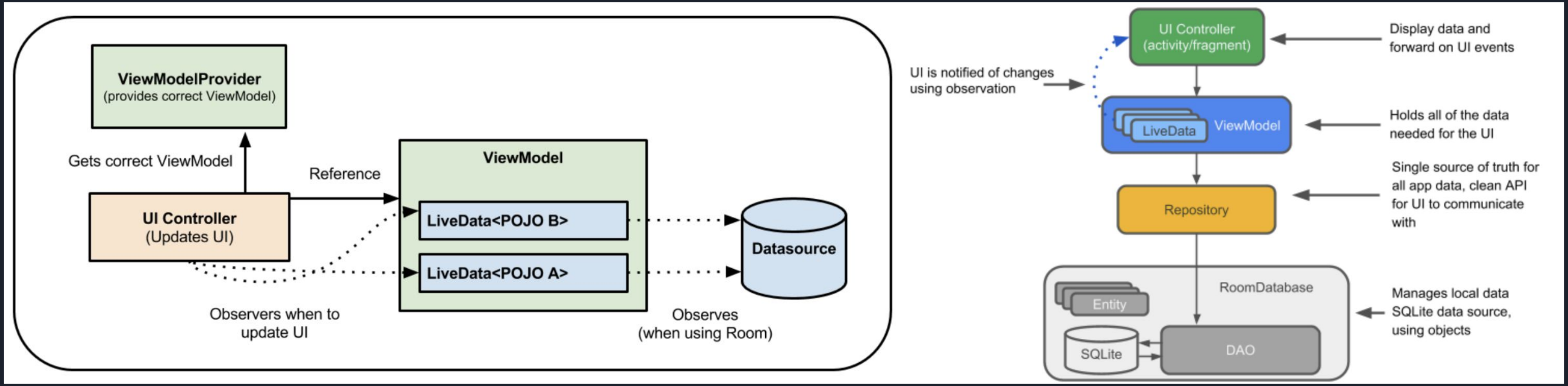
```java
public class SharedViewModel extends ViewModel {
    private final MutableLiveData<Item> selected = new MutableLiveData<Item>();

    public void select(Item item) {
        selected.setValue(item);
    }

    public LiveData<Item> getSelected() {
        return selected;
    }
}
```



```java
public class MasterFragment extends Fragment {
    private SharedViewModel model;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class);
        itemSelector.setOnClickListener(item -> {
            model.select(item);
        });
    }
}

public class DetailFragment extends Fragment {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        SharedViewModel model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class);
        model.getSelected().observe(this, { item ->
           // Update the UI.
        });
    }
}
```

# LiveData

- Look back at *ViewModel* in a more detailed diagram:
  - *ViewModel* ensures that the data survives a device configuration change.
  - *Room* informs your *LiveData* when database changes.
  - *LiveData*, in turn, updates your UI with revised data.


ViewModelProvider
(provides correct ViewModel)
Gets correct ViewModel
Reference
ViewModel
UI Controller
(Updates UI)
LiveData<POJO B>
LiveData<POJO A>
Datasource
Observers when to update UI
Observes
(when using Room)
UI Controller
(activity/fragment)
Display data and forward on UI events
UI is notified of changes using observation
LiveData
ViewModel
Holds all of the data needed for the UI
Repository
Single source of truth for all app data, clean API for UI to communicate with
RoomDatabase
Entity
SQLite
DAO
Manages local data SQLite data source, using objects

## LiveData

- *LiveData* considers an observer (represented by the *Observer* class) to be in an active state if its lifecycle is in the *STARTED* or *RESUMED* state.

### No more manual lifecycle handling

UI components just observe relevant data and don’t stop or resume observation.

### Ensures your UI matches data state

Notifies Observer objects when the lifecycle state changes.

### No memory leaks or crashes

Bound to Lifecycle objects and clean up; Inactive observer doesn’t receive any LiveData events.

### Always up to date data

If a lifecycle becomes inactive, it receives the latest data upon becoming active again.

## LiveData

1. **Create an instance of *LiveData*** to hold a certain type of data.

   This is usually done within your *ViewModel* class.

2. **Create an *Observer*** object that defines the *onChanged()* method, which controls what happens when the *LiveData* object's held data changes.

   You usually create an *Observer* object in a UI controller, such as an activity or fragment.

3. **Attach the *Observer*** object to the *LiveData* object using the *observe()* method. This subscribes the *Observer* object to the *LiveData* object so that it is notified of changes.

   You usually attach the *Observer* object in a UI controller, such as an activity or fragment.

# LiveData

Create *LiveData* objects

```
public class NameViewModel extends ViewModel {

// Create a LiveData with a String
private MutableLiveData<String> mCurrentName;

    public MutableLiveData<String> getCurrentName() {
        if (mCurrentName == null) {
            mCurrentName = new MutableLiveData<String>();
        }
        return mCurrentName;
    }

// Rest of the ViewModel...
}
```

- *LiveData* is a wrapper that can be used with any data, including objects that implement *Collections*, such as *List*.

# LiveData

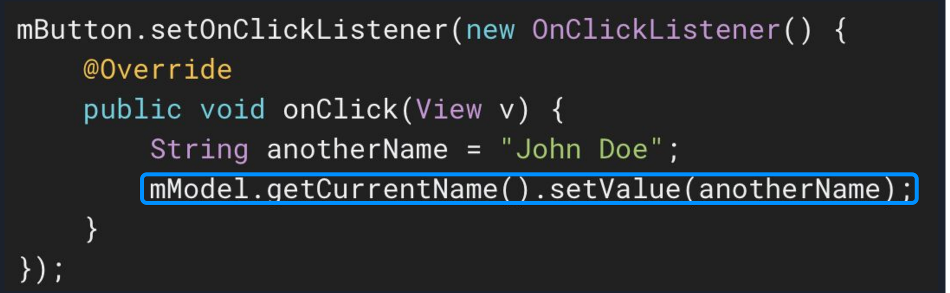
## Observe *LiveData* objects

```
public class NameActivity extends AppCompatActivity {

    private NameViewModel mModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Other code to setup the activity...

        // Get the ViewModel.
        mModel = ViewModelProviders.of(this).get(NameViewModel.class);


        // Create the observer which updates the UI.
        final Observer<String> nameObserver = new Observer<String>() {
            @Override
            public void onChanged(@Nullable final String newName) {
                // Update the UI, in this case, a TextView.
                mNameTextView.setText(newName);
            }
        };

        // Observe the LiveData, passing in this activity as the LifecycleOwner
        mModel.getCurrentName().observe(this, nameObserver);
    }
}
```

- The *observe()* method takes a *LifecycleOwner* object.

## LiveData

Update *LiveData* objects



```
mButton.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        String anotherName = "John Doe";
        mModel.getCurrentName().setValue(anotherName);
    }
});
```

- Call *postValue()* from a another thread.

## LiveData

Sharing resources

- You can extend a *LiveData* object using the singleton pattern to wrap system services, so that they can be shared in your app.
- The *LiveData* object connects to the system service once, and then any observer that needs the resource can just watch the *LiveData* object.
- In order to manage the lifecycle of a whole application process in this case, see *ProcessLifecycleOwner*.

For more information, see Extend LiveData.

## CodeLab

Step 1. Open the link and clone the repo.

Step 2

Add a *ViewModel*

Step 3

Wrap Data Using *LiveData*

Step 4

Subscribe to *Lifecycle* Events

Step 5

Share a *ViewModel* between Fragments

# Thank you!

Q & A session