Android Architecture Components

Mayank Bhatnagar
Ivy Comptech
@mayank.bhatnagar

Darshan Parikh
AYA Carpool
@activesince93

Overview

- Announced at Google IO 2017
- Provide framework to create more maintainable and robust app
- Encourage decoupled components within the app

WHY AAC ?

Persist Data

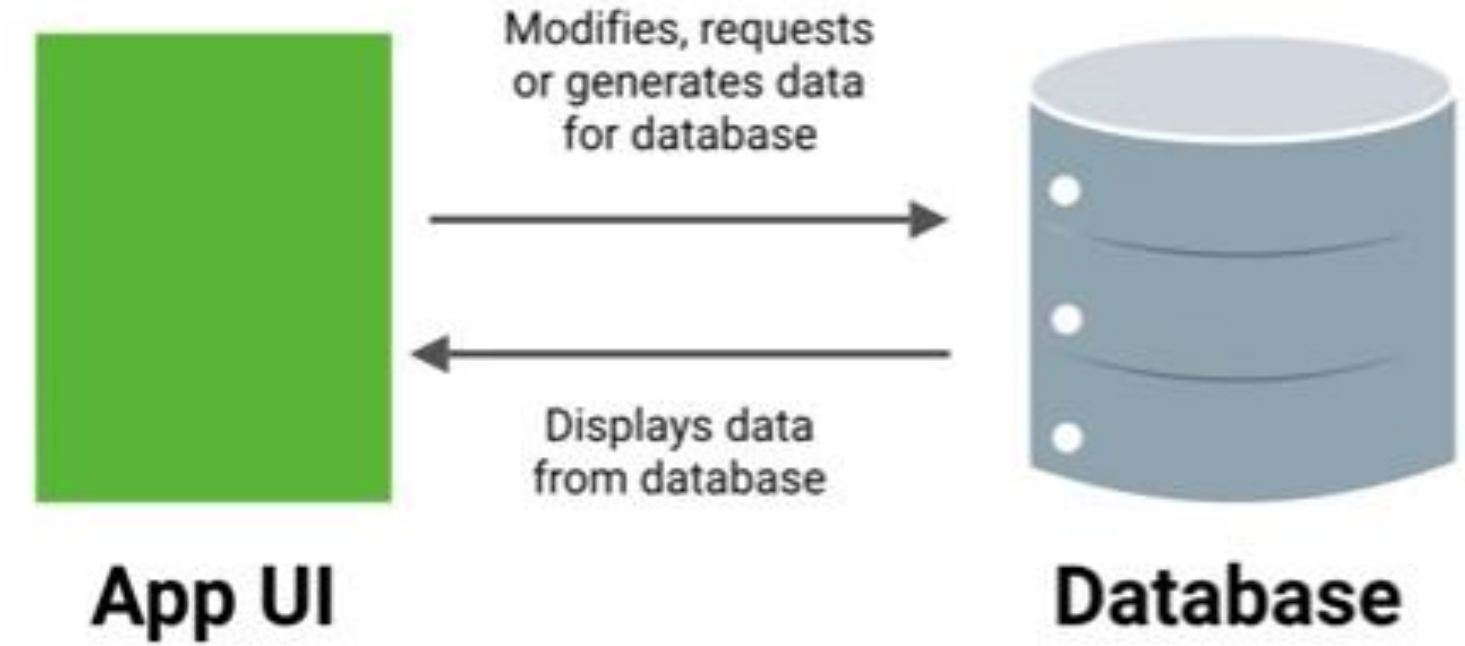
Manage Lifecycle

Modular

Defense Against
Common Errors

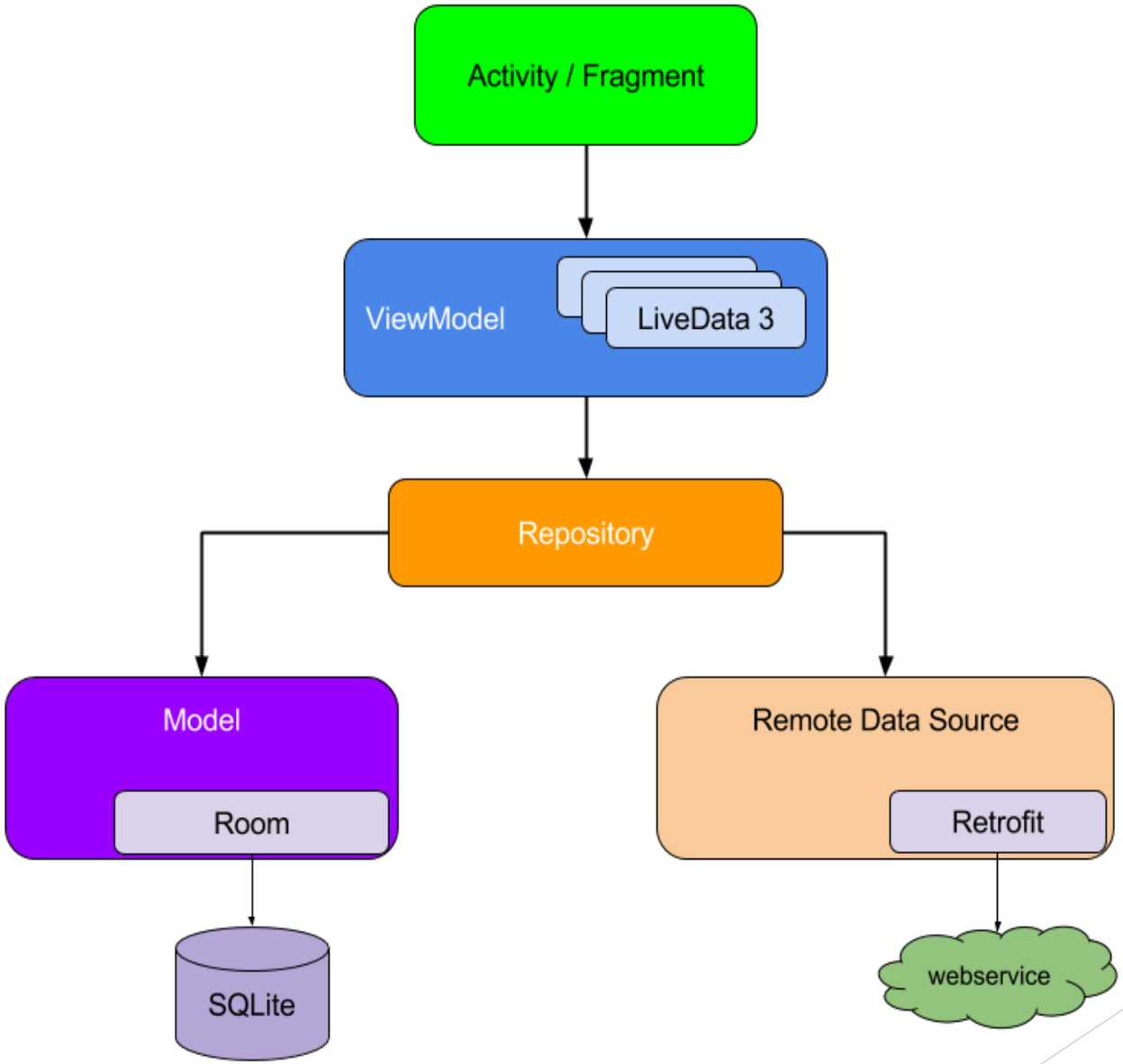
Less
Boilerplate

Basic app


Modifies, requests
or generates data
for database
Displays data
from database
App UI
Database

Components

- Room
- Live Data
- Life cycle
- ViewModel



- This year some more components got added to under AAC.

- Navigation
- Paging
- WorkManager

- Together all these components are now a part of Android Jetpack

Data Binding

Lifecycles

LiveData

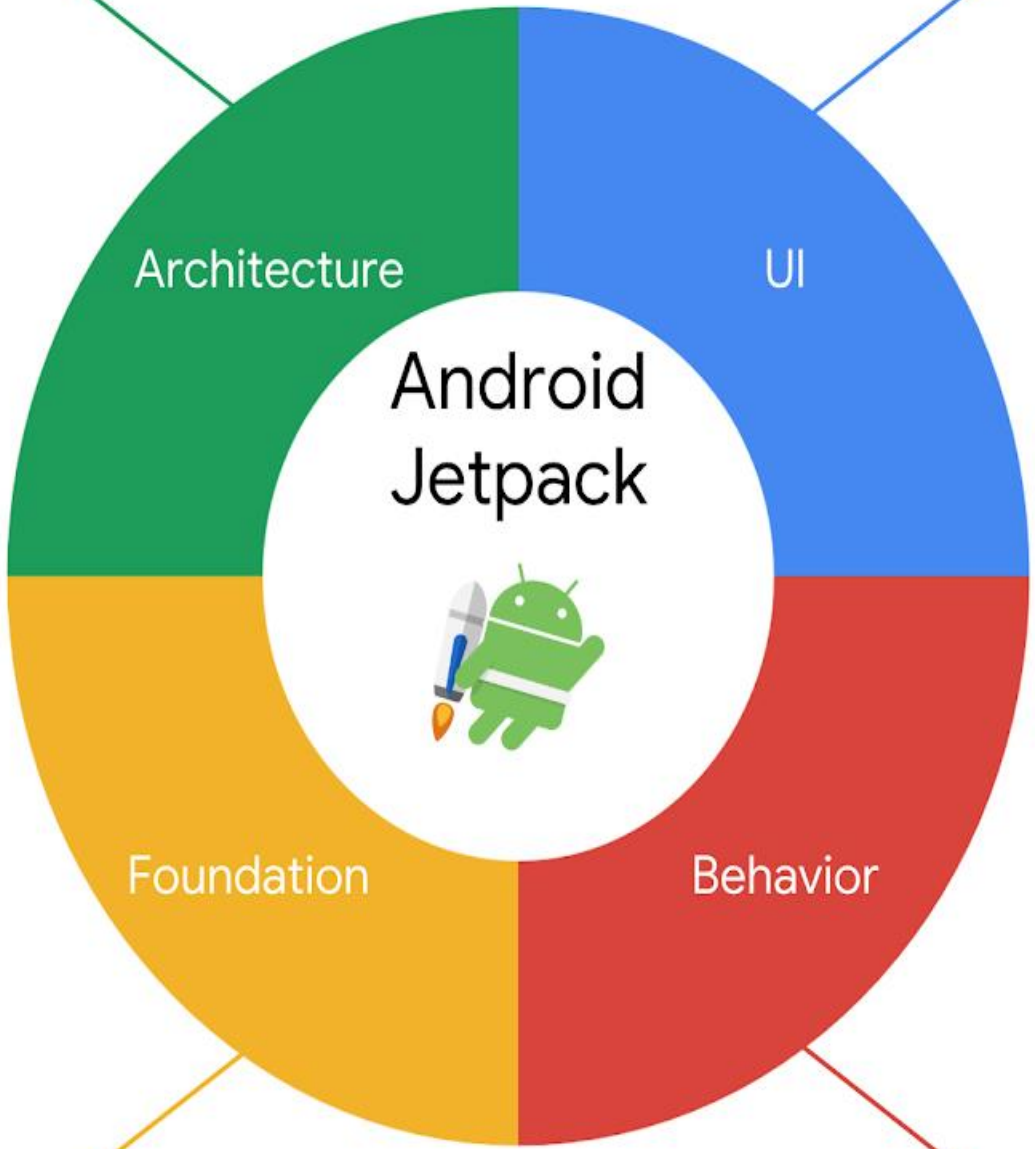
Navigation *new!*

Paging *new!*

Room

ViewModel

WorkManager *new!*



Animation & Transitions

Auto, TV & Wear

Emoji

Fragment

Layout

Palette

AppCompat

Android KTX *new!*

Multidex

Test

Download Manager

Media & Playback

Permissions

Notifications

Sharing

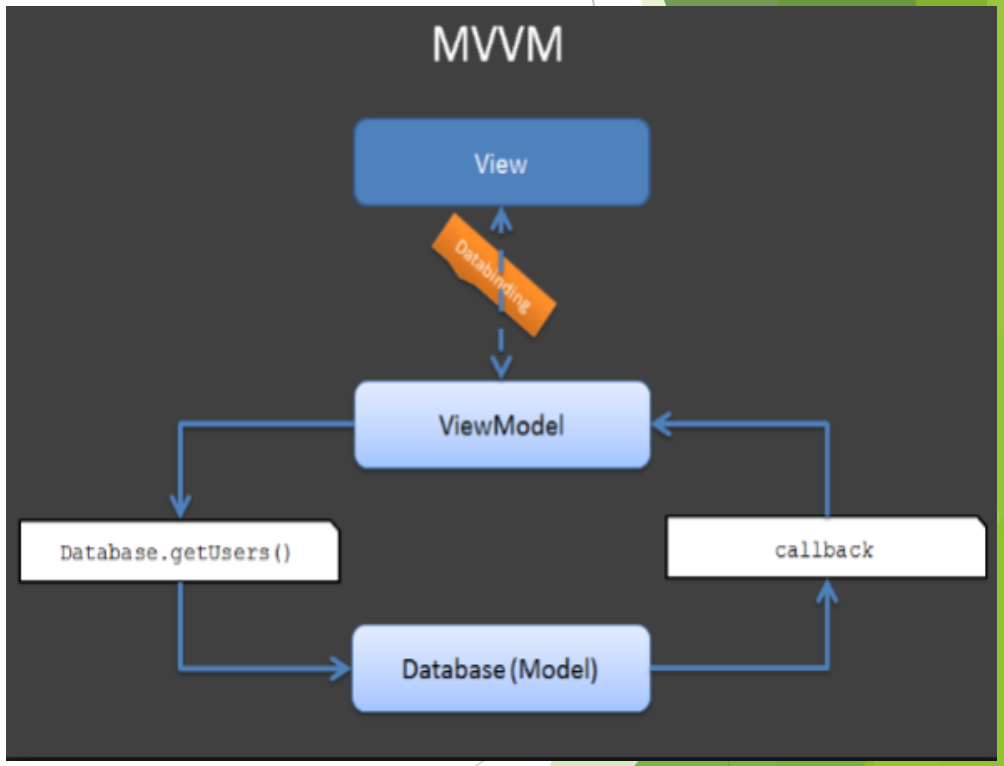
new! Slices

Model View ViewModel (MVVM)

Model
- Entity class

View
- Activity/Fragment or CustomView

ViewModel
- Interaction with model
- Updating the UI



Data Persistence

- Content Providers
- SharedPreferences
- SQLite

Problems with SQLite?

- Complex and confusing

And all we get is a **Cursor!**

Read information from a database

To read from a database, use the `query()` method, passing it your selection criteria and desired columns. The method combines elements of `insert()` and `update()`, except the column list defines the data you want to fetch (the "projection"), rather than the data to insert. The results of the query are returned to you in a `Cursor` object.

KOTLIN JAVA

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    BaseColumns._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_SUBTITLE
    };

// Filter results WHERE "title" = 'My Title'
String selection = FeedEntry.COLUMN_NAME_TITLE + " = ?";
String[] selectionArgs = { "My Title" };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_SUBTITLE + " DESC";

Cursor cursor = db.query(
    FeedEntry.TABLE_NAME,   // The table to query
    projection,             // The array of columns to return (pass null to get all)
    selection,              // The columns for the WHERE clause
    selectionArgs,          // The values for the WHERE clause
    null,                   // don't group the rows
    null,                   // don't filter by row groups
    sortOrder               // The sort order
    );
```

The third and fourth arguments (`selection` and `selectionArgs`) are combined to create a WHERE clause. Because the arguments are provided separately from the selection query, they are escaped before being combined. This makes

Contents

Problems with SQLite?

- Complex and confusing
- Too much to handle for a small use case

Save data using SQLite

Saving data to a database is ideal for repeating or structured data, such as contact information. This page assumes that you are familiar with SQL databases in general and helps you get started with SQLite databases on Android. The APIs you'll need to use a database on Android are available in the `android.database.sqlite` package.

Caution: Although these APIs are powerful, they are fairly low-level and require a great deal of time and effort to use:

- There is no compile-time verification of raw SQL queries. As your data graph changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
- You need to use lots of boilerplate code to convert between SQL queries and data objects.

For these reasons, we **highly recommended** using the Room Persistence Library as an abstraction layer for accessing information in your app's SQLite databases.

self-documenting way.

A contract class is a container for constants that define names for URIs, tables, and columns. The contract class allows you to use the same constants across all the other classes in the same package. This lets you change a column name in one place and have it propagate throughout your code.

A good way to organize a contract class is to put definitions that are global to your whole database in the root level of the class. Then create an inner class for each table. Each inner class enumerates the corresponding table's columns.

Note: By implementing the `BaseColumns` interface, your inner class can inherit a primary key field called `_ID` that some Android classes such as `CursorAdapter` expect it to have. It's not required, but this can help your database work

Problems with SQLite?

- Complex and confusing
- Too much to handle for a small use case
- No compile time verification

```
String TABLE_KEYS = "keys";
long KEY_VALUE = 1234;
String KEY_ID = "key_id";
String GET_KEYS = "SELECT * FROM" + TABLE_KEYS + " WHERE " + KEY_ID + " = " +
KEY_VALUE;
db.execSQL(GET_KEYS);
```

```
Caused by:
android.database.sqlite.SQLiteException:
near "FROMkeys": syntax error(code 1):
,while compiling: SELECT * FROMkeys where
key_id = 1234
```

Solution?

Room

SQLite mapping library

Room

- Boilerplate-free code
- SQLite support
- Compile time verification
- Works well with observables

Classes required to implement database handling in **SQLite**

- DatabaseHandler extends **SQLiteOpenHelper**
- UI handler (Activity/Fragment)
- Entity class

Main components of **Room**

- @Entity

```
@Entity
class User {
    @PrimaryKey
    int userId;
    String name;
    String email;
}
```

Main components of **Room**

- @Entity
- @Dao

```
@Dao
interface UserDao {
    @Query("SELECT * FROM users WHERE user_id = :userId")
    User get(long userId)
}
```

```
@Entity
class User {
    @PrimaryKey
    int userId;
    String name;
    String email;
}
```

Main components of **Room**

- @Entity
- @Dao
- @Database

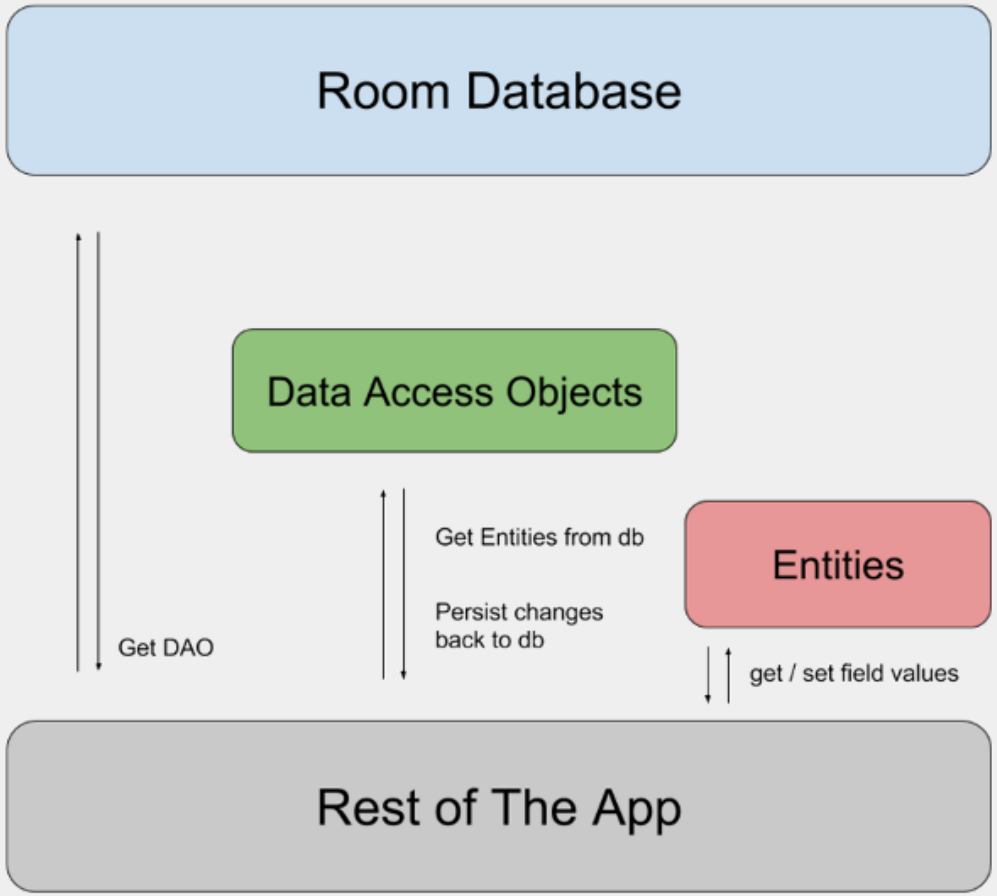
```
@Dao
interface UserDao {
    @Query("SELECT * FROM users WHERE user_id = :userId")
    User get(long userId)
}
```

```
@Entity
class User {
    @PrimaryKey
    int userId;
    String name;
    String email;
}
```

```
@Database(entities = {User.class}, version = 1)
abstract class MyDatabase extends RoomDatabase {
    abstract UserDao userDao();
}
```

Main components of **Room**

- @Entity
- @Dao
- @Database



What else?

- Room understands **SQLite**

```java
@Dao
interface UserDao {
    @Query("SELECT * FROM userss WHERE user_id = :userId")
    User get(long userId);
}
```

```
[SQLITE_ERROR] SQL error or missing database(no such table: userss)
```

What else?

- Room understands **SQLite**
- Room perfectly works with **LiveData**

```
@Dao
interface UserDao {
    @Query("SELECT * FROM users WHERE user_id = :userId")
    LiveData<User> get(long userId);
}
```

Updates in I/O 2018 (Room 1.1)

- Supports multithreading (WAL compatibility)

What is WAL (WriteAheadLogging) compatibility?

- Enables **parallel execution** of queries from **multiple threads** on the same database

- When enabled, write operations occur in a separate log file which allows reads to proceed concurrently

- Maximum number of connections dependent upon the device memory

How does it work?

- While write is in progress, readers on other threads will perceive the state of the database as it was before the write began. When the write completes, readers on other threads will then perceive the new state of the database.

- It’s automatically taken care if device is running **JB+** and not running on low RAM (Generally means 1GB or low RAM)

Updates in I/O 2018 (Room 1.1)

- Supports multithreading (WAL compatibility)
- @RawQuery annotation

Problem with @Query

- Querying with multiple parameters is complex to maintain
- Even if dynamic implementation is written, it returns **Cursor**!

```java
@Dao
interface UserDao {
    @RawQuery
    List<User> getUsers(SupportSQLiteQuery query);
}
```

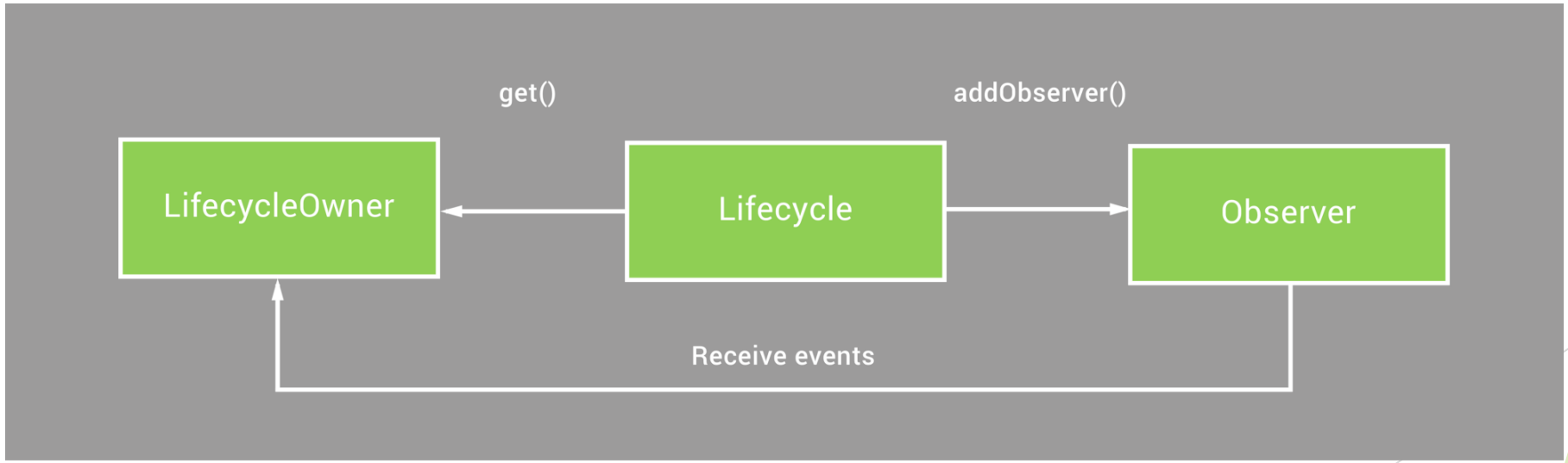
```java
List<Object> params = ...
String query = "SELECT * FROM users where ...";
RoomDatabase db = ...

UserDao userDao = db.userDao();
SimpleSQLiteQuery supportQuery = new SimpleSQLiteQuery(query, params);
List<User> users = userDao.getUsers(supportQuery);
```

Room

- Boilerplate-free code
- SQLite support
- Compile time verification
- Works well with observables

Handling Lifecycle with Lifecycle-Aware Components

- Performs action in response to change in the lifecycle status of another component, such as activities and fragments.



Lifecycle Problem

- LocationManager.start()

LocationManager.stop()

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

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

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

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
    }
}
```

Solution

- Lifecycle Owners- objects with Lifecycle like Activities and fragments
- Lifecycle Observers- observe Lifecycle Owners and are notified lifecycle changes

Lifecycle Owner

- Single method interface that denotes that the class has a Lifecycle
- To maintain Lifecycle of a application process extend ProcessLifecycleOwner

```
getLifecycle().addObserver(new MyObserver());
```

```java
package android.arch.lifecycle;

import android.support.annotation.NonNull;

/**
 * A class that has an Android lifecycle. These events can be used by custom components to
 * handle lifecycle changes without implementing any code inside the Activity or the Fragment.
 *
 * @see Lifecycle
 */
/WeakerAccess, unused/
public interface LifecycleOwner {
    /**
     * Returns the Lifecycle of the provider.
     *
     * @return The lifecycle of the provider.
     */
    @NonNull
    Lifecycle getLifecycle();
}
```

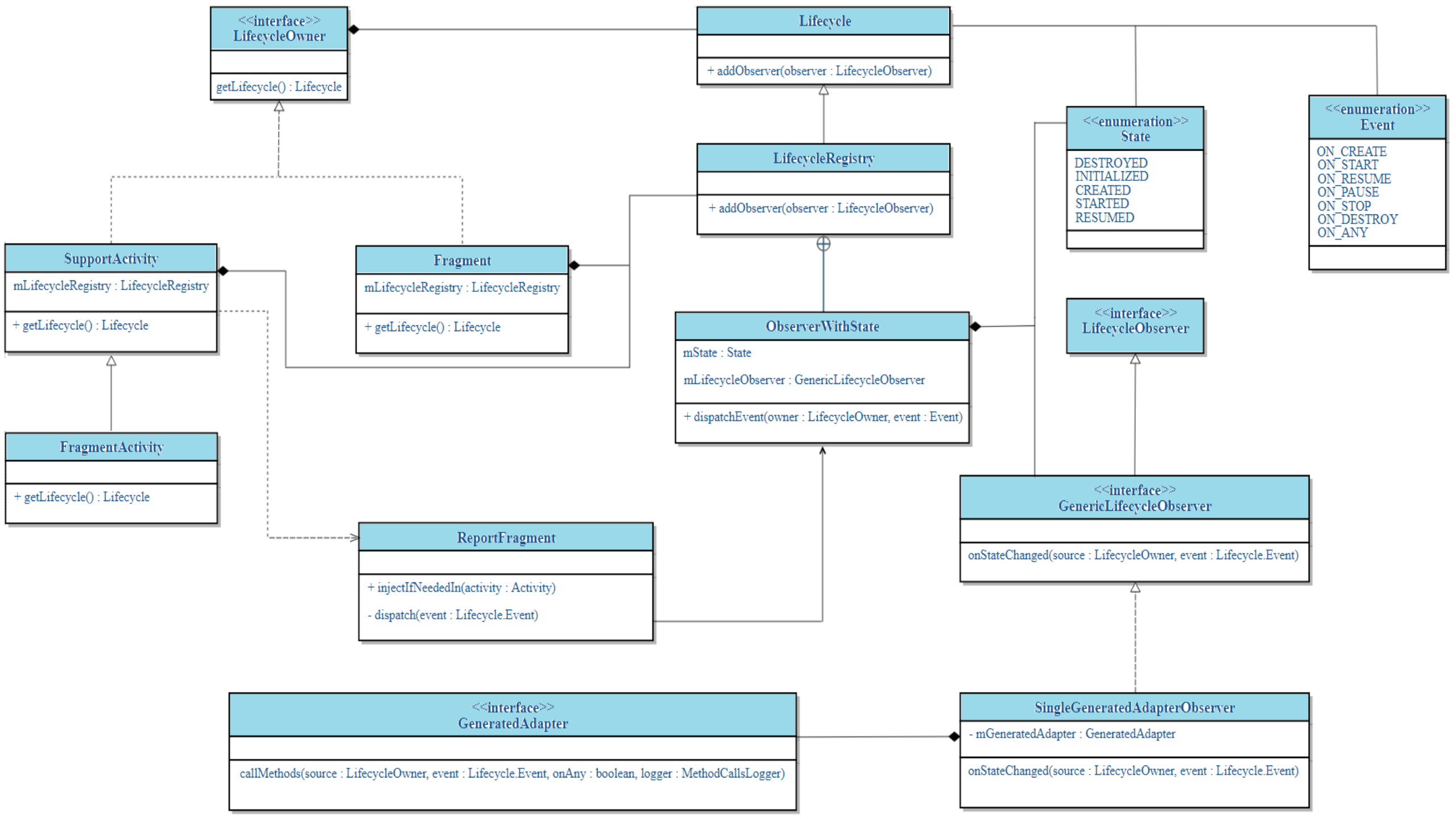
Lifecycle Observer

```
public class MyObserver implements LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
    public void connectListener() {
        ...
    }

    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
    public void disconnectListener() {
        ...
    }
}
```

Use cases of Lifecycle-Aware Components

- Switching between coarse and fine-grained location updates.
- Stopping and starting video buffering.
- Starting and stopping network connectivity.
- Pausing and resuming animated drawables.

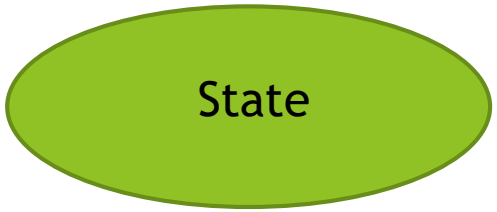
BEHIND THE SCENES

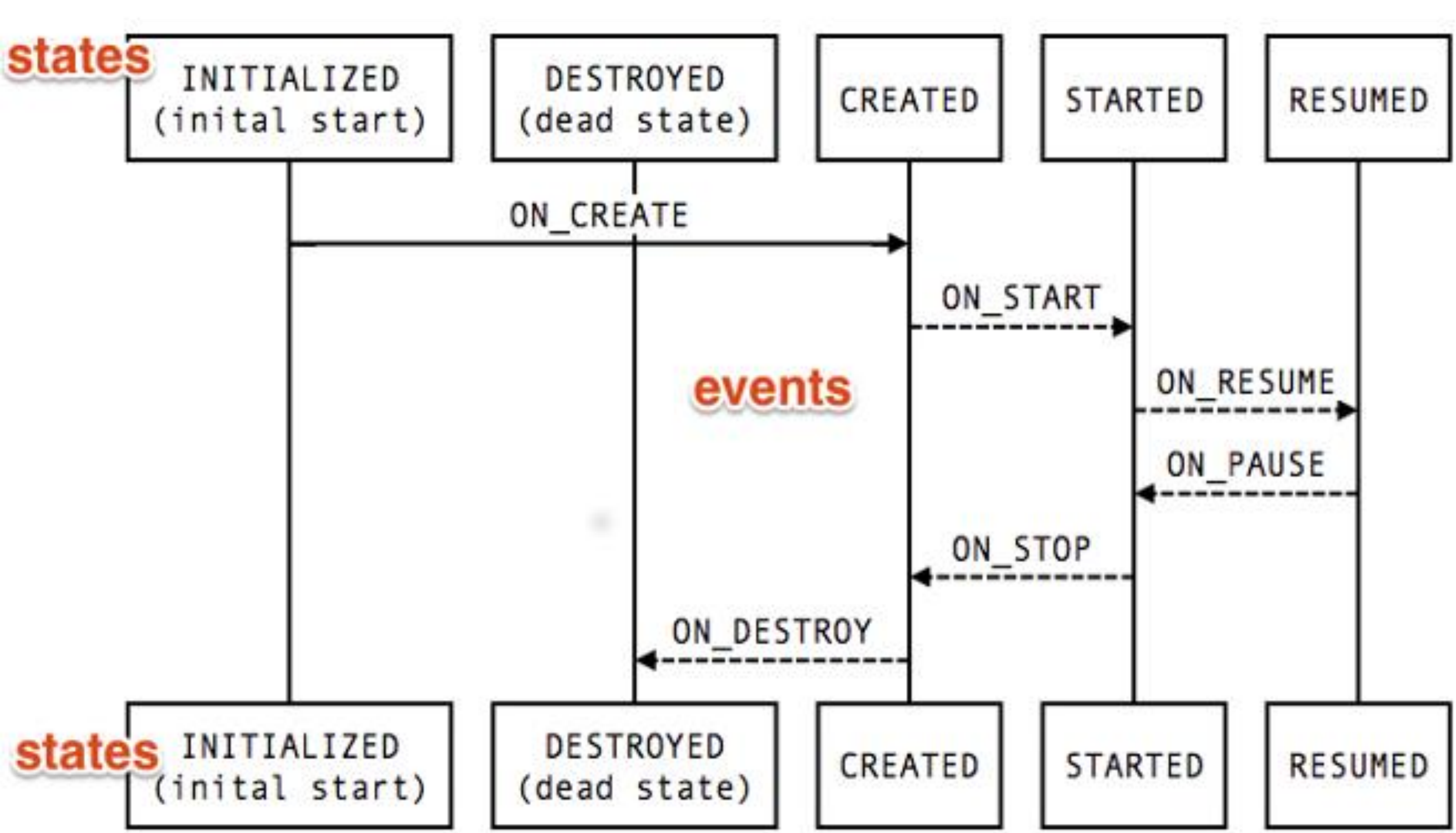

<<interface>>
LifecycleOwner
getLifecycle() : Lifecycle
Lifecycle
+ addObserver(observer : LifecycleObserver)
<<enumeration>>
State
DESTROYED
INITIALIZED
CREATED
STARTED
RESUMED
<<enumeration>>
Event
ON_CREATE
ON_START
ON_RESUME
ON_PAUSE
ON_STOP
ON_DESTROY
ON_ANY
LifecycleRegistry
+ addObserver(observer : LifecycleObserver)
SupportActivity
mLifecycleRegistry : LifecycleRegistry
+ getLifecycle() : Lifecycle
Fragment
mLifecycleRegistry : LifecycleRegistry
+ getLifecycle() : Lifecycle
ObserverWithState
mState : State
mLifecycleObserver : GenericLifecycleObserver
+ dispatchEvent(owner : LifecycleOwner, event : Event)
<<interface>>
LifecycleObserver
FragmentActivity
+ getLifecycle() : Lifecycle
ReportFragment
+ injectIfNeededIn(activity : Activity)
- dispatch(event : Lifecycle.Event)
<<interface>>
GenericLifecycleObserver
onStateChanged(source : LifecycleOwner, event : Lifecycle.Event)
<<interface>>
GeneratedAdapter
callMethods(source : LifecycleOwner, event : Lifecycle.Event, onAny : boolean, logger : MethodCallsLogger)
SingleGeneratedAdapterObserver
- mGeneratedAdapter : GeneratedAdapter
onStateChanged(source : LifecycleOwner, event : Lifecycle.Event)

Lifecycle

- Holds the information about the lifecycle state of a component (like an activity or a fragment) and allows other objects to observe this state.

- It contains two enumerations to track the lifecycle status




states
INITIALIZED
(inital start)
DESTROYED
(dead state)
CREATED
STARTED
RESUMED
ON_CREATE
ON_START
events
ON_RESUME
ON_PAUSE
ON_STOP
ON_DESTROY
states
INITIALIZED
(inital start)
DESTROYED
(dead state)
CREATED
STARTED
RESUMED

Observing events in JAVA 8

DefaultLifecycleObserver

```
public interface DefaultLifecycleObserver
implements LifecycleObserver
```

android.arch.lifecycle.DefaultLifecycleObserver

Callback interface for listening to `LifecycleOwner` state changes.

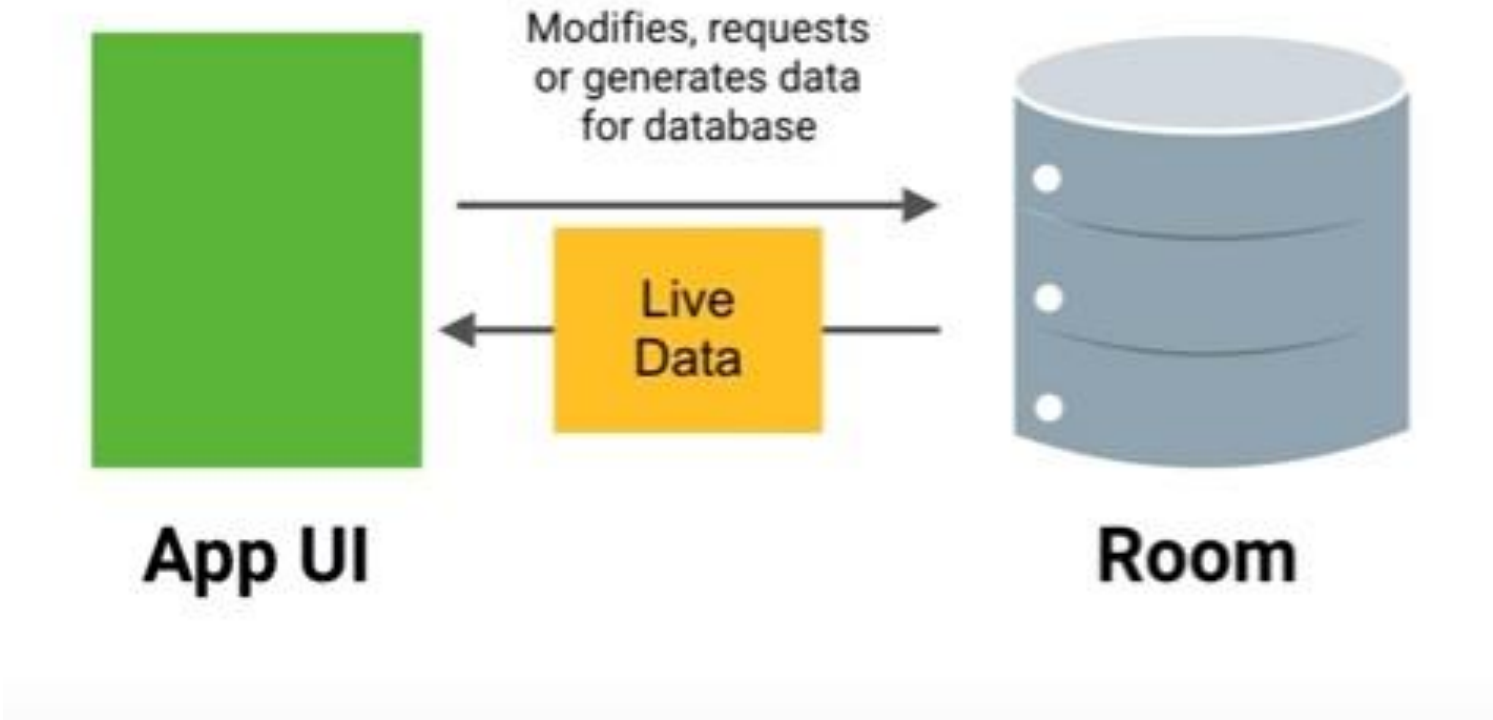
If you use Java 8 language, **always** prefer it over annotations.

Summary

Public methods	
`default void`	`onCreate(LifecycleOwner owner)` Notifies that `ON_CREATE` event occurred.
`default void`	`onDestroy(LifecycleOwner owner)` Notifies that `ON_DESTROY` event occurred.
`default void`	`onPause(LifecycleOwner owner)` Notifies that `ON_PAUSE` event occurred.
`default void`	`onResume(LifecycleOwner owner)` Notifies that `ON_RESUME` event occurred.
`default void`	`onStart(LifecycleOwner owner)` Notifies that `ON_START` event occurred.
`default void`	`onStop(LifecycleOwner owner)` Notifies that `ON_STOP` event occurred.

Live Data

- An observable data holder
- It is lifecycle-aware



Advantages of Live Data

- Ensure your UI matches
- No memory leaks
- No crashes due to stopped activities
- No more manual lifecycle handling
- Always up to date data
- Proper configuration changes

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...
}
```

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 and the observer.
        mModel.getCurrentName().observe(this, nameObserver);
    }
}
```

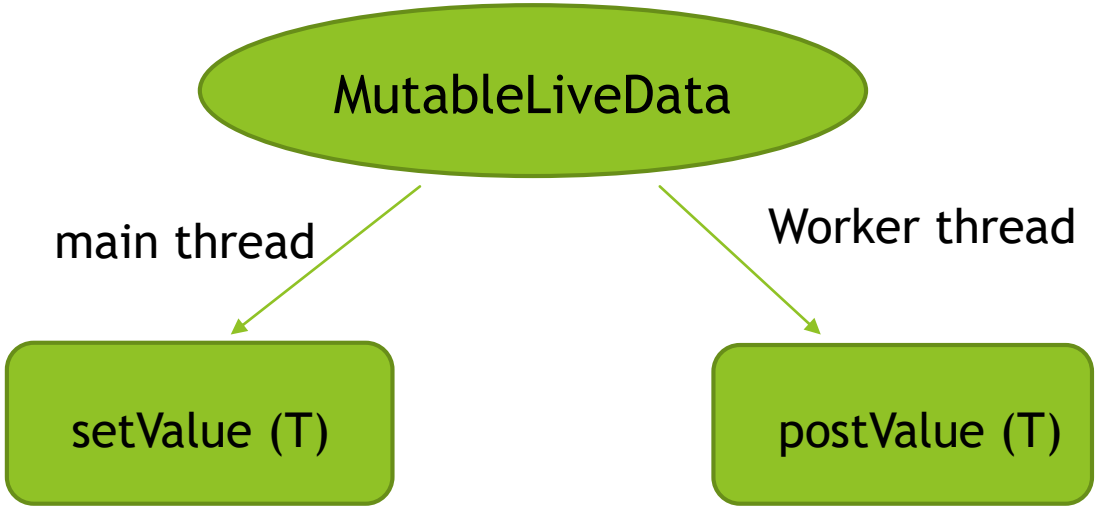
Where should LiveData objects reside?

WHY ???

VIEWMODEL

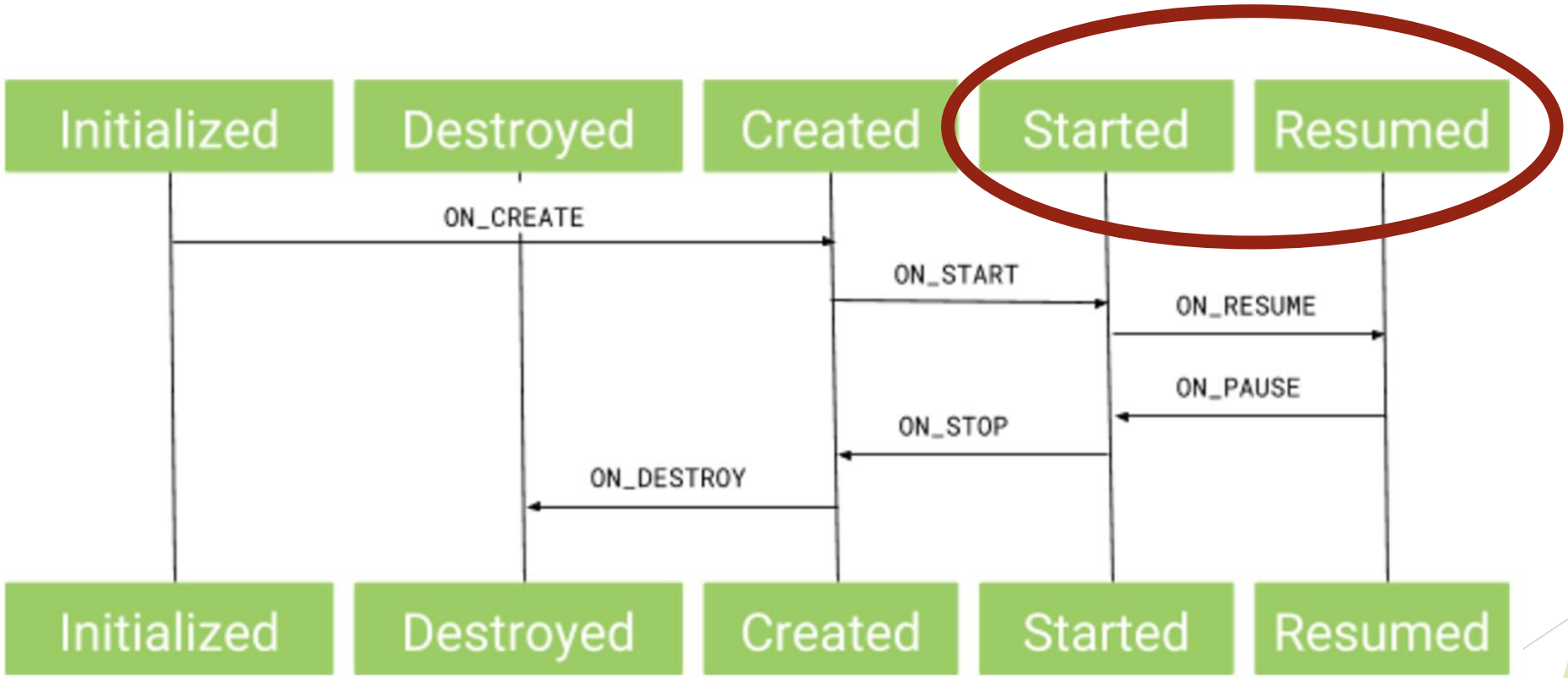
- To avoid bloating UI controllers i.e. activities and fragments
- To decouple LiveData instances from specific UI controllers

How to trigger updates using LiveData ?




```
mCurrentName () . setValue (“Mayank”);
```

What is an Active state?



Extend LiveData

```
public class StockLiveData extends LiveData<BigDecimal> {
    private StockManager mStockManager;

    private SimplePriceListener mListener = new SimplePriceListener() {
        @Override
        public void onPriceChanged(BigDecimal price) {
            setValue(price);
        }
    };

    public StockLiveData(String symbol) {
        mStockManager = new StockManager(symbol);
    }

    @Override
    protected void onActive() {
        mStockManager.requestPriceUpdates(mListener);
    }

    @Override
    protected void onInactive() {
        mStockManager.removeUpdates(mListener);
    }
}
```

Transform LiveData

- Make changes to the value stored in a LiveData object before dispatching it to the observers

- Helper functions

  - Transformation.map()

  - Transformation.switchMap()

BEHIND THE SCENES

Diving into “Observe” method

```
@MainThread
public void observe(@NonNull LifecycleOwner owner, @NonNull Observer<T> observer) {
    if (owner.getLifecycle().getCurrentState() == DESTROYED) {
        // ignore
        return;
    }
    LifecycleBoundObserver wrapper = new LifecycleBoundObserver(owner, observer);
    ObserverWrapper existing = mObservers.putIfAbsent(observer, wrapper);
    if (existing != null && !existing.isAttachedTo(owner)) {
        throw new IllegalArgumentException("Cannot add the same observer"
                + " with different lifecycles");
    }
    if (existing != null) {
        return;
    }
    owner.getLifecycle().addObserver(wrapper);
}
```

Depends upon LifecycleOwner and Observer instance being passed

How Observers listens to data update ?

```
@MainThread
protected void setValue(T value) {
    assertMainThread( methodName: "setValue");
    mVersion++;
    mData = value;
    dispatchingValue( initiator: null);
}

private void dispatchingValue(@Nullable ObserverWrapper initiator) {
    if (mDispatchingValue) {
        mDispatchInvalidated = true;
        return;
    }
    mDispatchingValue = true;
    do {
        mDispatchInvalidated = false;
        if (initiator != null) {
            considerNotify(initiator);
            initiator = null;
        } else {
            for (Iterator<Map.Entry<Observer<T>, ObserverWrapper>> iterator =
                    mObservers.iteratorWithAdditions(); iterator.hasNext(); ) {
                considerNotify(iterator.next().getValue());
                if (mDispatchInvalidated) {
                    break;
                }
            }
        }
    } while (mDispatchInvalidated);
    mDispatchingValue = false;
}
```

Updates **data** and notify observers

- Invalidates previous dispatch if needed
- Iterates over observers to notify data set change


MAGIC

Notifying Observers

```
private void considerNotify(ObserverWrapper observer) {
    if (!observer.mActive) {
        return;
    }
    // Check latest state b4 dispatch. Maybe it changed state but we didn't get the event yet.
    //
    // we still first check observer.active to keep it as the entrance for events. So even if
    // the observer moved to an active state, if we've not received that event, we better not
    // notify for a more predictable notification order.
    if (!observer.shouldBeActive()) {
        observer.activeStateChanged( newActive: false);
        return;
    }
    if (observer.mLastVersion >= mVersion) {
        return;
    }
    observer.mLastVersion = mVersion;
    //noinspection unchecked
    observer.mObserver.onChanged((T) mData);
}
```

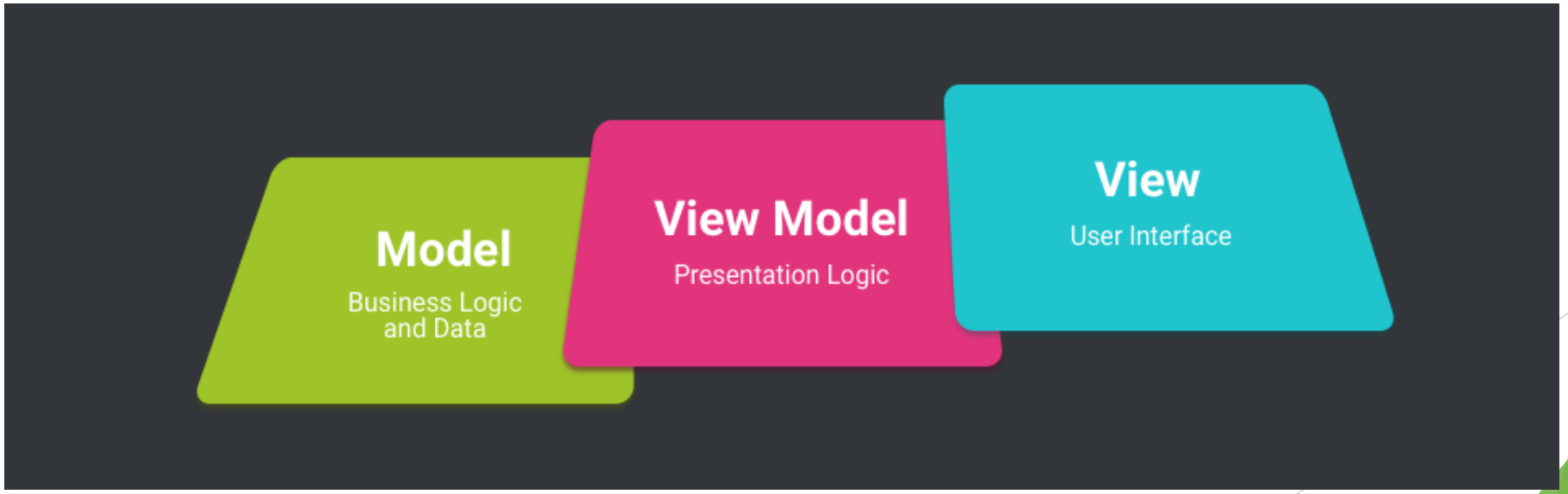
- Checks if Lifecycle state is active
- Checks latest data version is dispatched
- Finally calls onChanged method with new Data

Disposing the subscription

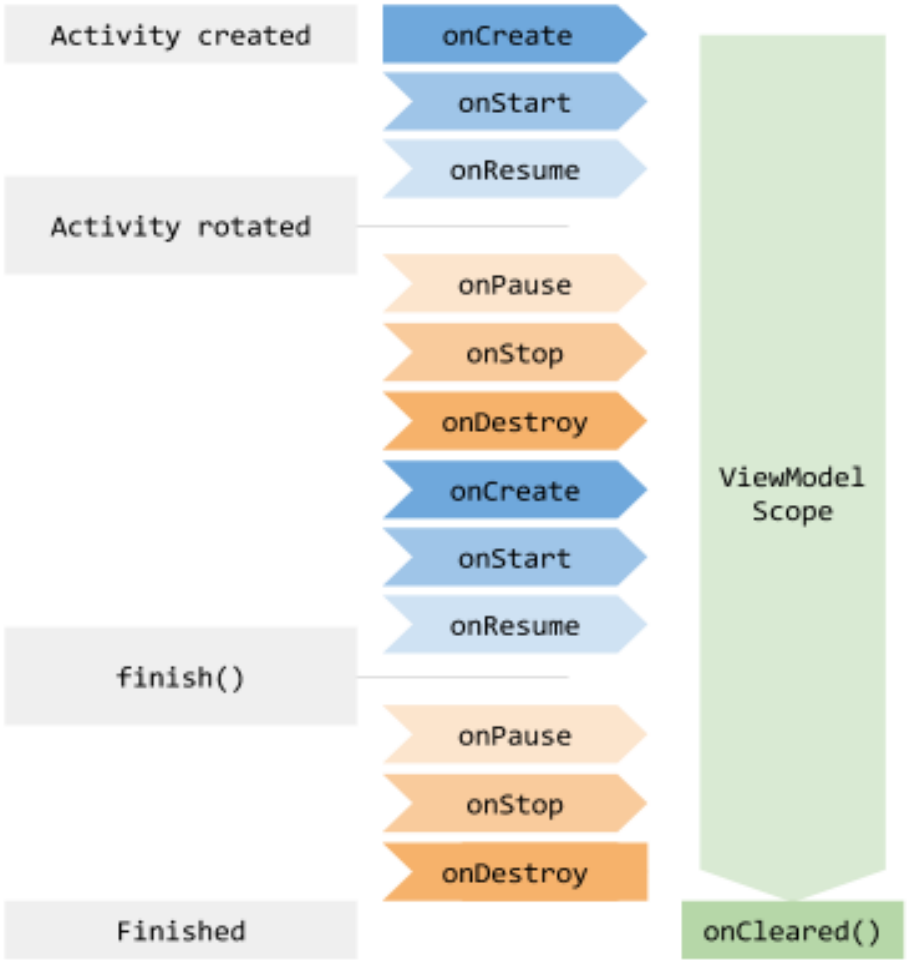
```
class LifecycleBoundObserver extends ObserverWrapper implements GenericLifecycleObserver {
    @NonNull final LifecycleOwner mOwner;

    LifecycleBoundObserver(@NonNull LifecycleOwner owner, Observer<T> observer) {
        super(observer);
        mOwner = owner;
    }

    @Override
    boolean shouldBeActive() {
        return mOwner.getLifecycle().getCurrentState().isAtLeast(STARTED);
    }

    @Override
    public void onStateChanged(LifecycleOwner source, Lifecycle.Event event) {
        if (mOwner.getLifecycle().getCurrentState() == DESTROYED) {
            removeObserver(mObserver);
            return;
        }
        activeStateChanged(shouldBeActive());
    }

    @Override
    boolean isAttachedTo(LifecycleOwner owner) {
        return mOwner == owner;
    }

    @Override
    void detachObserver() {
        mOwner.getLifecycle().removeObserver(this);
    }
}
```

ViewModel

- Store and manage UI-related data in a lifecycle conscious way.
- It allows data to survive configuration changes such as screen rotations.



Lifecycle of ViewModel Component



Getting instance of a ViewModel

```
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.
    }
}
```

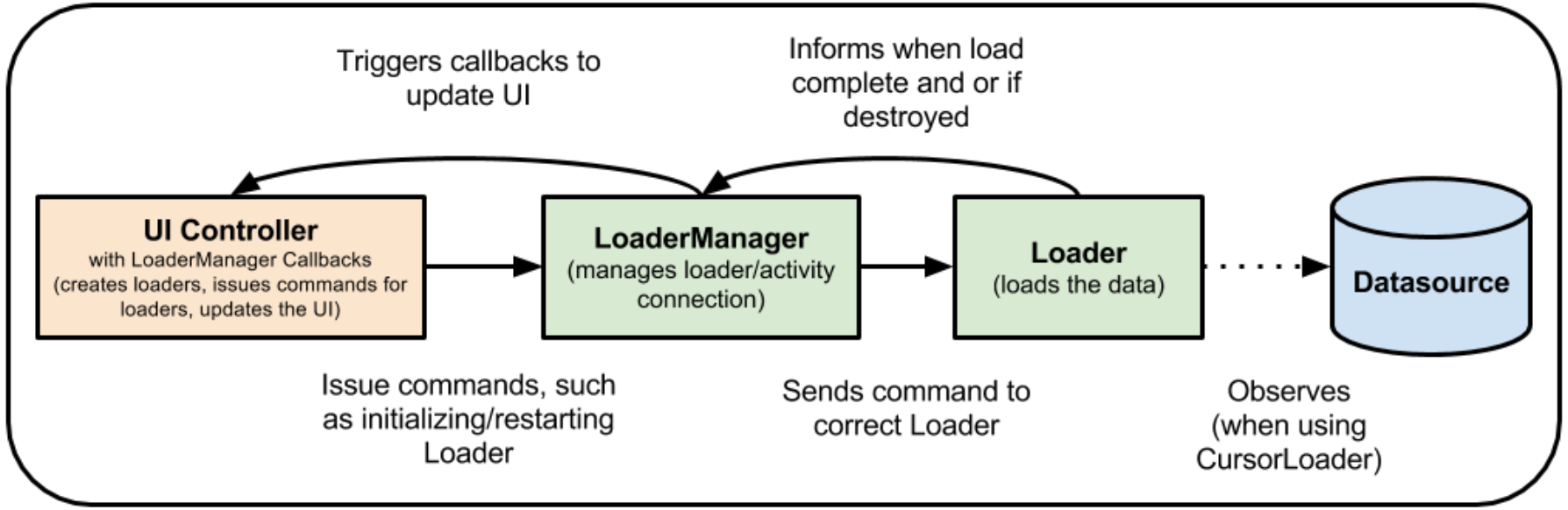
Implementing your ViewModel class

```
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
        });
    }
}
```

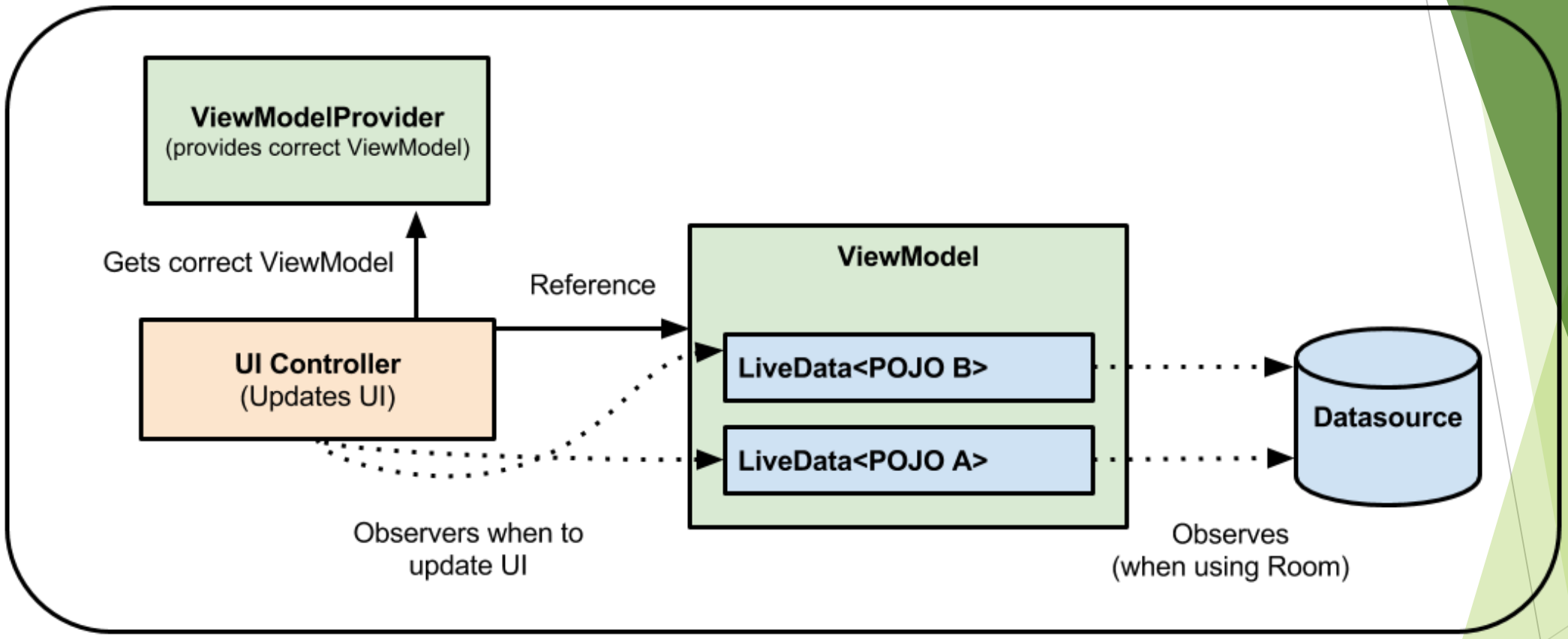
Share Data between Fragments

```
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;
    }
}


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.
        });
    }
}
```

Replacing Loaders with ViewModel

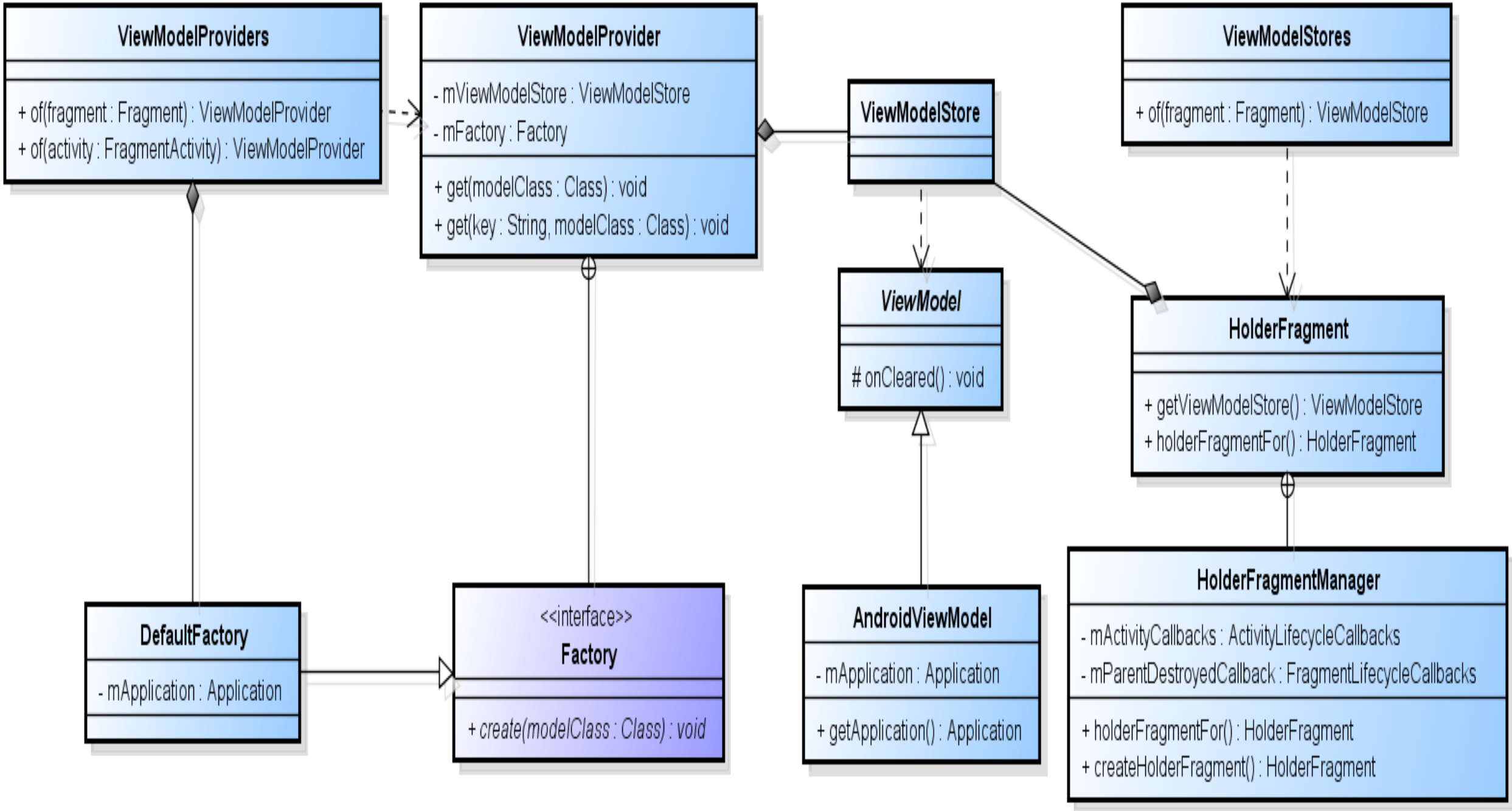


Loading data with loaders



Loading data with ViewModel

BEHIND THE SCENES


ViewModelProviders
+ of(fragment : Fragment) : ViewModelProvider
+ of(activity : FragmentActivity) : ViewModelProvider
ViewModelProvider
- mViewModelStore : ViewModelStore
- mFactory : Factory
+ get(modelClass : Class) : void
+ get(key : String, modelClass : Class) : void
ViewModelStore
ViewModelStores
+ of(fragment : Fragment) : ViewModelStore
ViewModel
onCleared() : void
HolderFragment
+ getViewModelStore() : ViewModelStore
+ holderFragmentFor() : HolderFragment
DefaultFactory
- mApplication : Application
<<interface>>
Factory
+ create(modelClass : Class) : void
AndroidViewModel
- mApplication : Application
+ getApplication() : Application
HolderFragmentManager
- mActivityCallbacks : ActivityLifecycleCallbacks
- mParentDestroyedCallback : FragmentLifecycleCallbacks
+ holderFragmentFor() : HolderFragment
+ createHolderFragment() : HolderFragment

ViewModelProviders.of(this)

```java
/**
 * Creates a {@link ViewModelProvider}, which retains ViewModels while a scope of given Activity
 * is alive. More detailed explanation is in {@link ViewModel}.
 * <p>
 * It uses the given {@link Factory} to instantiate new ViewModels.
 *
 * @param activity an activity, in whose scope ViewModels should be retained
 * @param factory  a {@code Factory} to instantiate new ViewModels
 * @return a ViewModelProvider instance
 */
@NonNull
@MainThread
public static ViewModelProvider of(@NonNull FragmentActivity activity,
        @Nullable Factory factory) {
    Application application = checkApplication(activity);
    if (factory == null) {
        factory = ViewModelProvider.AndroidViewModelFactory.getInstance(application);
    }
    return new ViewModelProvider(ViewModelStores.of(activity), factory);
}
```

ViewModelProviders.of(this).get(MyViewModel.class)

```
@NonNull
@MainThread
public <T extends ViewModel> T get(@NonNull String key, @NonNull Class<T> modelClass) {
    ViewModel viewModel = mViewModelStore.get(key);

    if (modelClass.isInstance(viewModel)) {
        //noinspection unchecked
        return (T) viewModel;
    } else {
        //noinspection StatementWithEmptyBody
        if (viewModel != null) {
            // TODO: log a warning.
        }
    }

    viewModel = mFactory.create(modelClass);
    mViewModelStore.put(key, viewModel);
    //noinspection unchecked
    return (T) viewModel;
}
```

How does HolderFragment retains the state?

```
public HolderFragment() {
    setRetainInstance(true);
}
```

Revision 27.1.0 Release

(February 2018)

Important Changes

- The underlying implementation of Loaders has been rewritten to use Lifecycle. While the API remains unchanged, there are a number of behavior changes:
  - `initLoader()`, `restartLoader()`, and `destroyLoader()` can now only be called on the main thread.
  - A Loader's `onStartLoading()` and `onStopLoading()` are now called when the containing FragmentActivity/Fragment is started and stopped, respectively.
  - `onLoadFinished()` will only be called between `onStart()` and `onStop`. As a result, Fragment transactions can now safely be done in `onLoadFinished()`.
  - The FragmentController methods related to Loaders are now deprecated.
- DialogFragment's `getDialog()` will now be non-null up until `onDestroyView()`, instead of becoming null in `dismiss()`. You can now determine if the Dialog was manually dismissed in `onStop()` by checking if `getDialog().isShowing()` returns false.

New APIs

- `ListAdapter` for `RecyclerView` (along with `AsyncListDiffer`) make it easier to compute list diffs on a background thread. These can help your RecyclerView animate content changes automatically, with minimal work on the UI thread. They use `DiffUtil` under the hood.
- `SortedList.ReplaceAll` enables updating all data in a SortedList, which runs all appropriate animations for inserts, removals, changes, and moves (moves are treated as removals and inserts).
- FragmentActivity and Fragment now implement `ViewModelStoreOwner` and can now be used with the `ViewModelProvider` constructors as an alternative to using `ViewModelProviders.of()`
- Fragments now have `requireContext()`, `requireActivity()`, `requireHost()`, and `requireFragmentManager()` methods, which return a `NonNull` object of the equivalent get methods or throw an

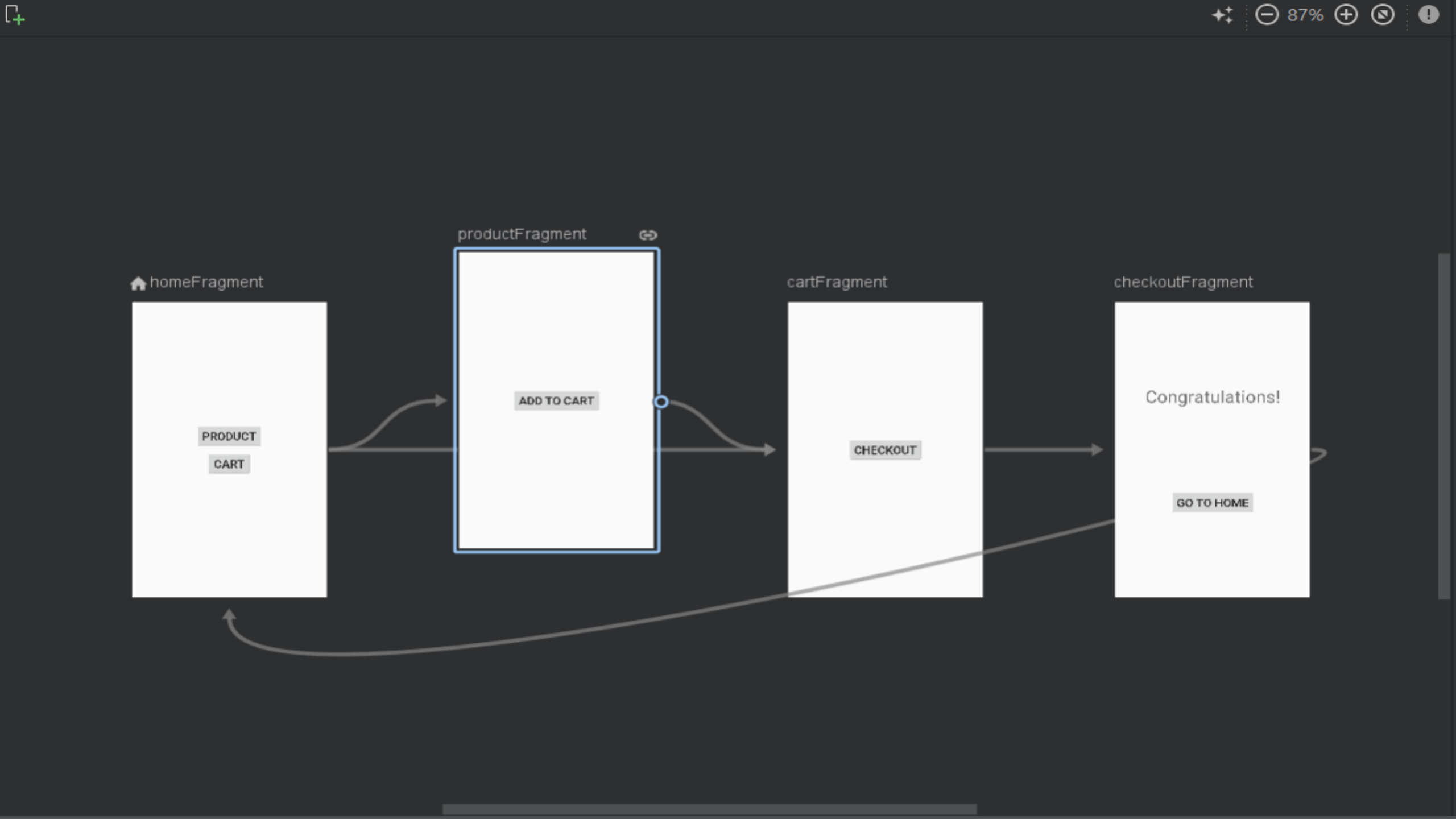
Navigation Principles

- App should have fixed starting destination
- Stack should define “navigation state”
- Up button never quits the app
- Up and back are equivalent within app’s task
- Deep linking to destination or navigating to the same destination should yield the same stack


THIS IS TOO MUCH

Navigation Component

- Easy in-app navigation


87%
productFragment
homeFragment
cartFragment
checkoutFragment
PRODUCT
CART
ADD TO CART
CHECKOUT
Congratulations!
GO TO HOME

```xml
<?xml version="1.0" encoding="utf-8"?>
<navigation ...
    app:startDestination="@id/homeFragment">
    <fragment ...
        android:id="@+id/homeFragment">
        <action
            android:id="@+id/action_homeFragment_to_cartFragment"
            app:destination="@id/cartFragment" />
    </fragment>
    <fragment ...>
        <action ... />
        <deepLink app:uri="www.demoshop.com/{productId}" />
        <argument
            android:name="prodId"
            android:defaultValue="0"
            app:argType="integer" />
    </fragment>
</navigation>
```

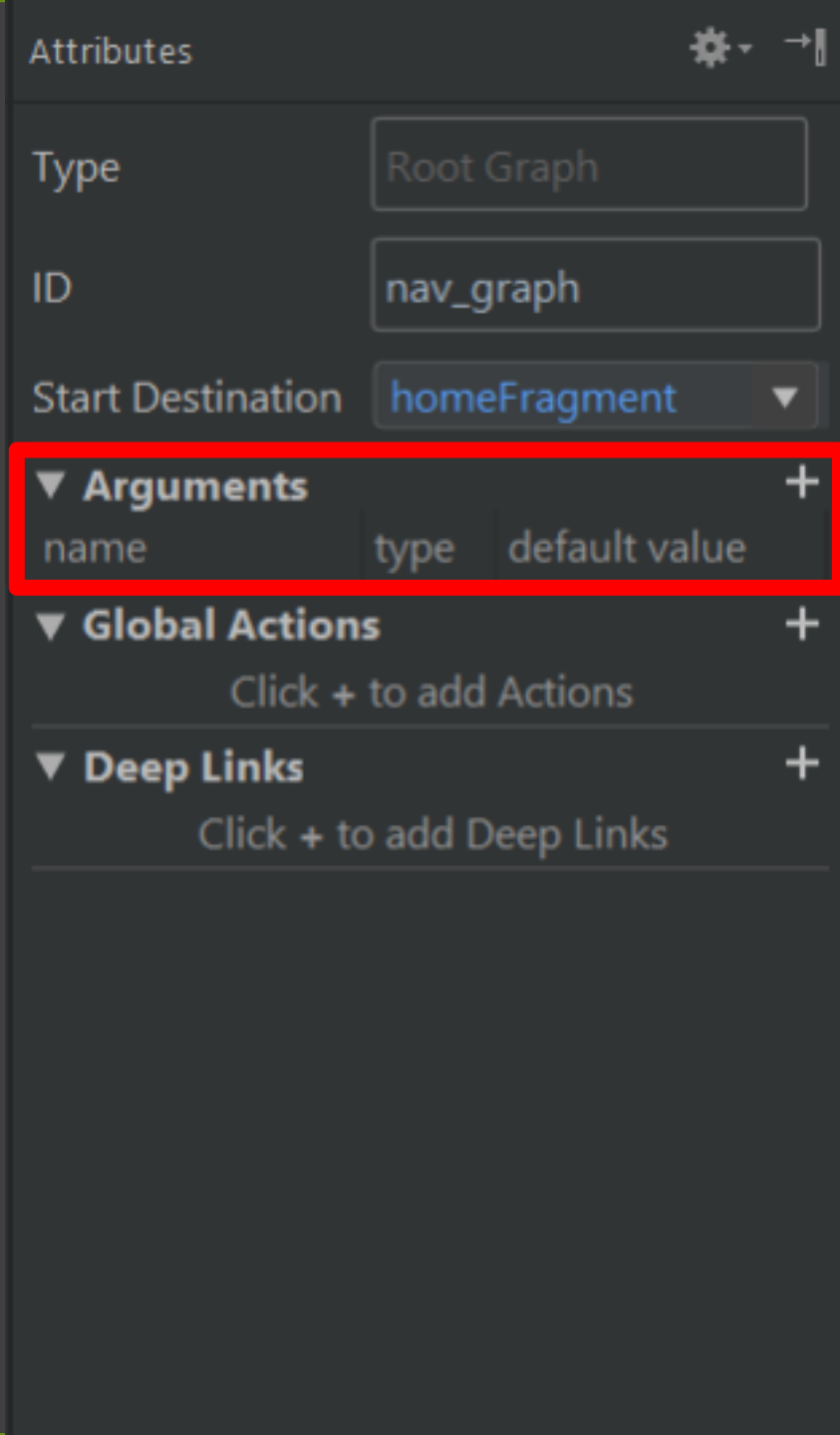
Navigation Component

- Easy in-app navigation
- Animations

Navigation Component

- Easy in-app navigation
- Animations
- Passing Arguments

```
<fragment ...>
    <action ... />
    <argument
        android:name="prodId"
        android:defaultValue="0"
        app:argType="integer" />
</fragment>
```



Navigation Component

- Easy in-app navigation
- Animations
- Passing Arguments
- DeepLinks

Explicit DeepLinks

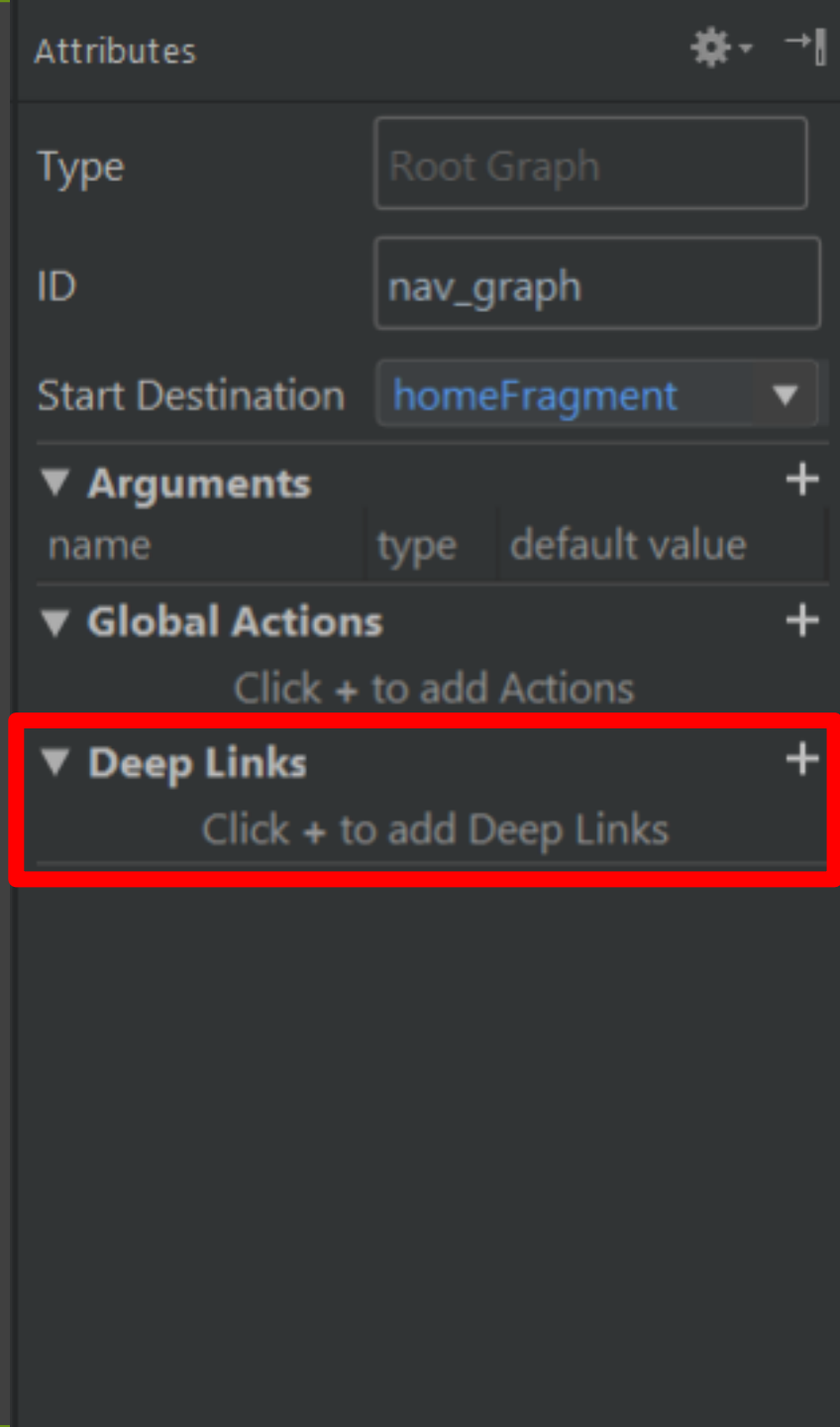
- Notifications
- App Shortcuts
- App Widgets
- Actions
- Slices

```
NavDeepLinkBuilder navDeepLinkBuilder = new NavDeepLinkBuilder(context)
    .setGraph(R.navigation.nav_graph)
    .setDestination(R.id.android)
    .setArguments(bundle);

PendingIntent pendingIntent = navDeepLinkBuilder.createPendingIntent();
notificationBuilder.setContentIntent(pendingIntent);
```

Implicit DeepLinks

- Web URLs
- Custom Scheme URLs

```
<fragment ...>
    <action ... />
    <deepLink app:uri="www.demoshop.com/{productId}" />
</fragment>
```



Navigation Component

- Easy in-app navigation
- Animations
- Passing Arguments
- DeepLinks
- No more fragment transactions!

```
Navigation
    .findNavController(view)
    .navigate(R.id.action_homeFragment_to_productFragment);
```

AnroidManifest.xml

```
<activity
    android:name=".MainActivity">
    <nav-graph android:value="@navigation/nav_graph"/>
</activity>
```

Background Jobs

- Sync data
- Uploading logs
- Data backups

~~Option~~ Options!

- JobScheduler (API 21+)
- FirebaseJobDispacher (API 14+, requires **Google Play Services**)
- AlarmManager
- Services
- Loaders

Too much to handle!

Threads
Executors
Services
AsyncTasks
Handlers and Loopers

Jobs (API 21+)
GcmNetworkManager
SyncAdapters
Loaders
AlarmManager

WorkManager

Components

- Worker
- WorkResult

  SUCCESS,
  FAILURE,
  RETRY

MyWorker.java

```
public class MyWorker extends Worker {
    ...
    public WorkerResult doWork() {
        // Implement background task
        return WorkerResult.SUCCESS;
    }
}
```

Workflow

- Worker
- WorkResult
- **WorkRequest**

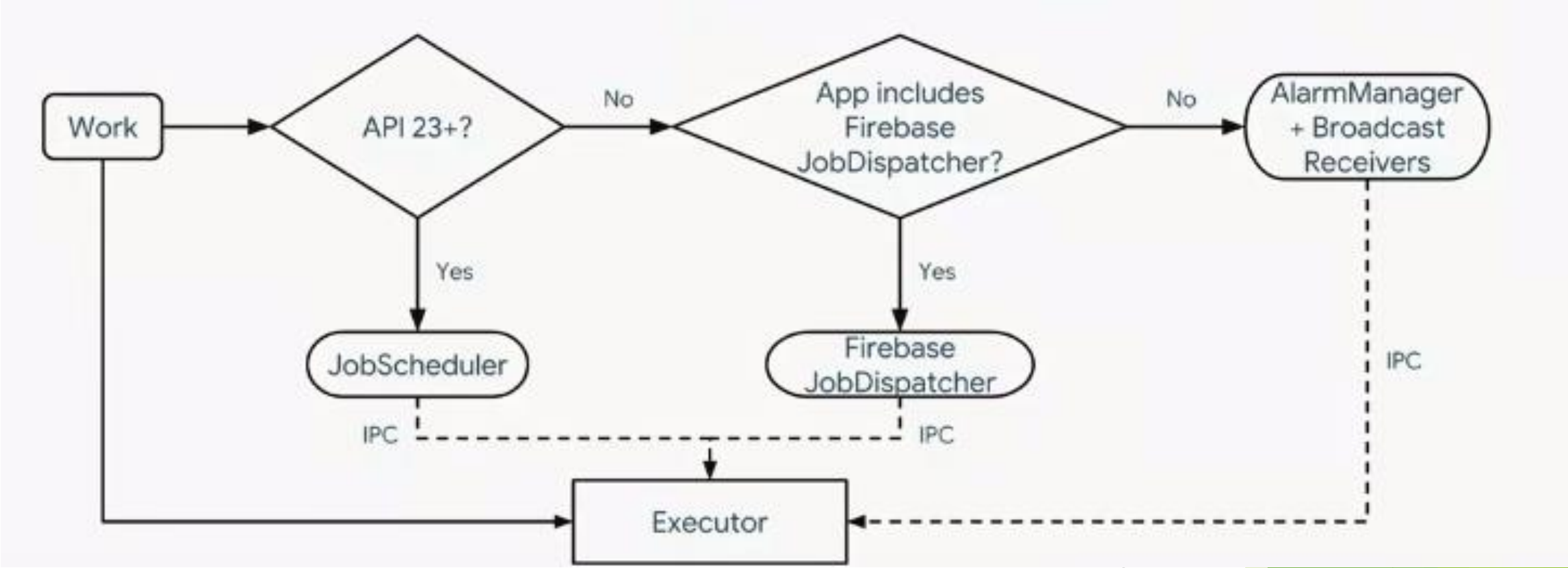
```
OneTimeWorkRequest uploadWork =
        new OneTimeWorkRequest.Builder
        (MyWorker.class)
    .build();
```

Workflow

- Worker
- WorkResult
- WorkRequest
- **WorkManager**

```
OneTimeWorkRequest oneTimeWork =
        new OneTimeWorkRequest
        .Builder(MyWorker.class)
.build();

// Enqueue work
WorkManager
        .getInstance()
        .enqueue(oneTimeWork);
```

Behind the scenes



What else?

- Observing Work

```
WorkManager
    .getInstance()
    .getStatusById(myWork.getId())
    .observe(lifecycleOwner, workStatus -> {
        // Do something with the status
        if (workStatus != null &&
workStatus.getState().isFinished())
        { ... }
});
```

What else?

- Observing Work
- **Task constraints**

```
// Whether device should be idle
.setRequiresDeviceIdle(boolean)

// Whether device should be plugged in
.setRequiresCharging(boolean)

// Whether device should have a particular NetworkType
.setRequiredNetworkType(NetworkType)

// Battery should not be below critical threshold
.setRequiresBatteryNotLow(boolean)

// Storage should not be below critical threshold
.setRequiresStorageNotLow(boolean)
```

What else?

- Observing Work
- Task constraints
- **Cancelling task**

```
UUID myWorkId = myWorkRequest.getId();
WorkManager.getInstance().cancelByWorkId(myWorkId);
```

What else?

- Observing Work
- Task constraints
- Cancelling task
- **Recurring tasks**

```
// Work will run every 12 hours
new PeriodicWorkRequest.Builder photoWorkBuilder =
        new PeriodicWorkRequest
.Builder(MyWorker.class, 12, TimeUnit.HOURS);
// Create the actual work object
PeriodicWorkRequest myWork =
photoWorkBuilder.build();
// Then enqueue the recurring task
WorkManager.getInstance().enqueue(myWork);
```

There’s more!

- Chaining work

```
WorkManager.getInstance()
    .beginWith(workA)
    // beginWith() eturns a WorkContinuation object
    .then(workB)
    // then() returns new WorkContinuation instance
    .then(workC)
    .enqueue();
```

There’s more!

- Chaining work
- **Input/Output data**

setInputData(iData)

```
Data iData = new Data.Builder()
    // We need to pass three integers: X, Y, and Z
    .putInt(KEY_X_ARG, 42)
    .putInt(KEY_Y_ARG, 421)
    .build();

// Enqueue a OneTimeWorkRequest using those arguments
OneTimeWorkRequest mathWork = new
OneTimeWorkRequest.Builder(MathWorker.class)
        .setInputData(iData)
        .build();
```

There’s more!

- Chaining work
- **Input/Output data**

setOutputData(oData)

```
@Override
public Worker.Result doWork() {
    // Fetch arguments (specify default values)
    int x = getInputData().getInt(KEY_X_ARG, 0);
    int y = getInputData().getInt(KEY_Y_ARG, 0);

    // ...do the math...
    int result = myCrazyMathFunction(x, y);

    //...set the output, and we're done!
    Data oData = new Data.Builder()
        .putInt(KEY_RESULT, result)
        .build();
    setOutputData(oData);
    return Result.SUCCESS;
}
```

Some more AAC are also there!

- Paging
- Data binding


THANK YOU!