Platform-Based Development: Android Architecture Components

BS UNI studies, Spring 2019/2020

Dr Veljko Pejović
Veljko.Pejovic@fri.uni-lj.si

Partly based on: Smyth, Neil. “Android Studio 3.3 Development Essentials - Android 9 Edition”

University *of Ljubljana*
Faculty *of Computer and Information Science*

Android Architecture Components

- Introduced in 2017 to make common mobile programming tasks easier, efficient, reliable
- Common tasks:
  - Lifecycle-dependent tasks
  - Store data in a database
  - Preserve data when a component is killed
  - Display data changes in UI
  - Bind UI views to the code and the data
  - Load data over a network
  - Perform background computations

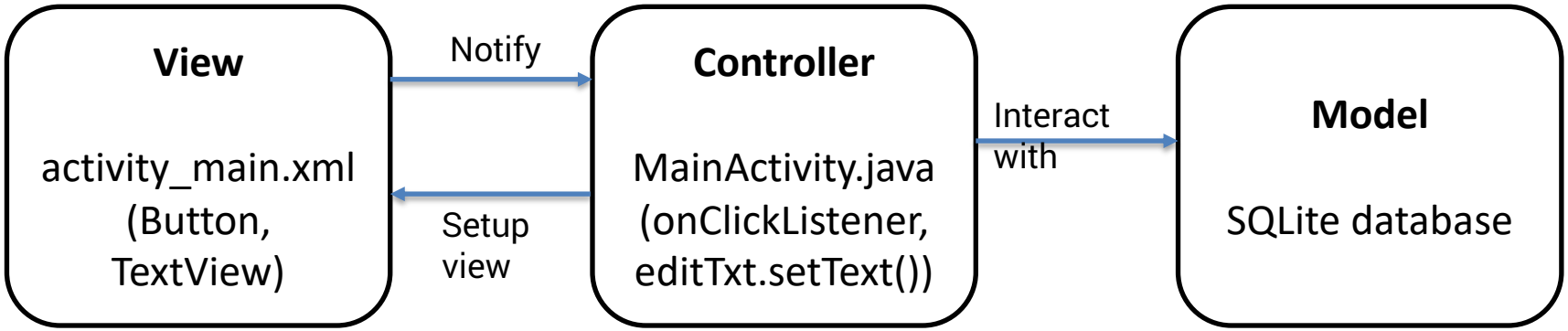
Some of these tasks already tackled by third-party libraries, e.g. Butterknife

Android Architecture Components

- New programming paradigms:
  - From Model-View-Controller (MVC) to Model-View-ViewModel (MVVM)
- New classes/methods/libraries:
  - LifecycleOwner and LifecycleObserver
  - RoomDatabase
  - ViewModel
  - LiveData
  - Data binding
  - Paging library
  - WorkManager

From MVC to MVVM

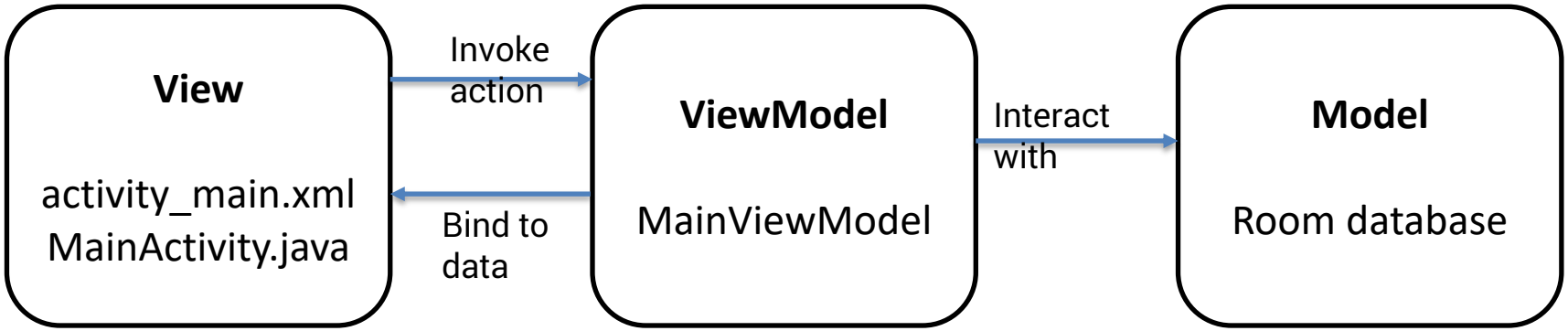
- MVC in Android



- Drawbacks:
  - Controller and View are tightly connected - change the view, you have to change the controller
  - Controller depends on user interactions (Activity)

From MVC to MVVM

- MVVM in Android



- Key points:
  - ViewModel is responsible for wrapping the model and preparing observable data
    - Does not know who is observing, can be more than one view
  - View binds to observable data invokes actions exposed by the ViewModel

Lifecycle-Awareness with Architecture Components Library

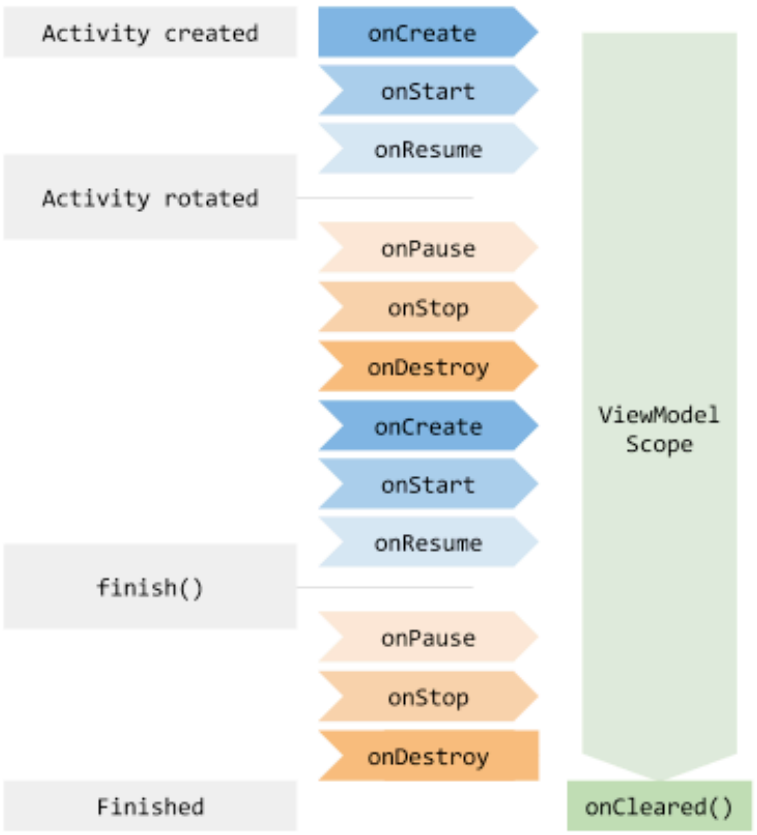
- Components that need to be aware of an Activity's lifecycle state can use androidx.lifecycle package classes
- LifecycleObserver
  - Get notified when a LifecycleOwner (such as an Activity) moves to ON_START, ON_RESUME, etc.
- Note: a similar functionality can be achieved within the lifecycle methods, but this is more elegant

ViewModel Class

- Problems:
  - Handling data in Activity/Fragment:
    - Prone to loss as the component may get destroyed (e.g. on screen rotation)
    - Prone to memory leaks as an asynchronous call from Activity/Fragment may return to a destroyed component
  - The same data might be needed at different views, Activity/Fragment is tightly connected with views
- Solution:
  - A new class that
    - Survives Activity/Fragment lifecycle changes
    - Provides the data, but is not aware of views using the data

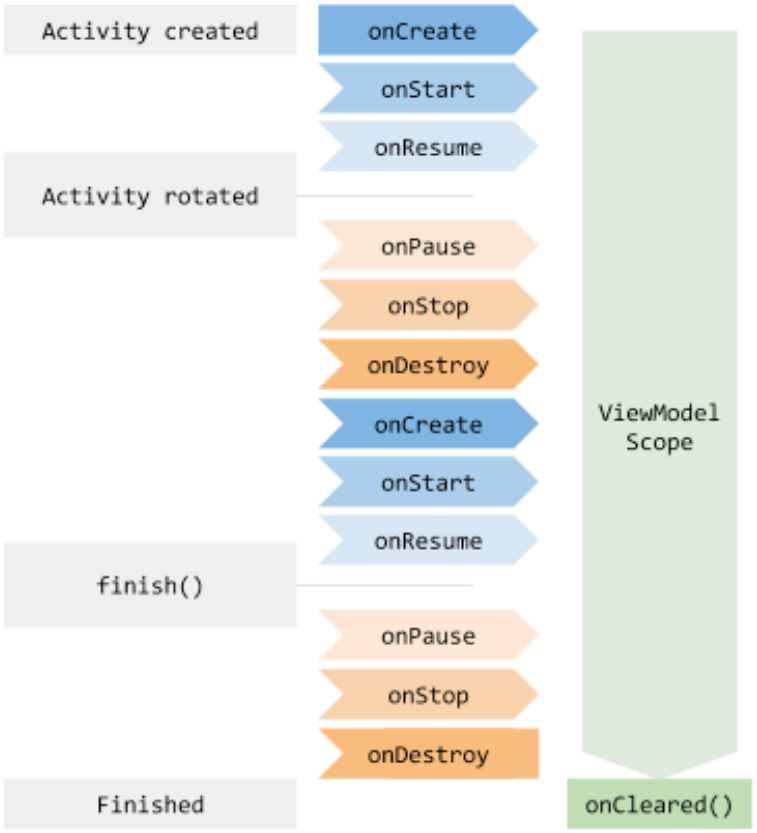
ViewModel Class

- ViewModel
  - Scoped to the ViewModelProvider's Lifecycle
    - Survives Activity onDestroy calls, but not application killed events!
  - In-memory data, not preserved on the long run!
- Implementation:
  - We usually extend AndroidViewModel class



ViewModel Class

- Pros
  - Data survives screen orientation changes
  - Multiple views can use the same ViewModel
  - No data leakage
- Cons:
  - Views must query the ViewModel to detect any changes in the data

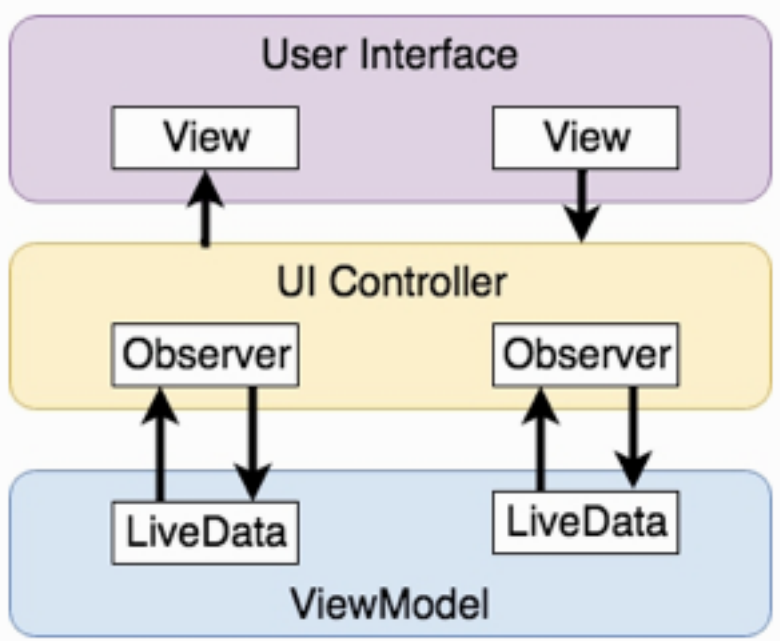


LiveData Class

- Problem:
  - Data (e.g. financial stocks) is updated frequently, the view must constantly check for the updates in an infinite loop
  - Views (or other entities) might want to update the data in the ViewModel – the info should be propagated to the Model
- Solution:
  - A new class that:
    - Holds the data, allows the data to be observed, notifies the observer when the data changes
    - Is lifecycle-aware – no updates if the observer is paused

LiveData Class

- Pros:
  - Data (e.g. financial stocks) is updated frequently, the view must constantly check for the updates in an infinite loop
- Cons:
  - Code still needs to be written to set and get View properties (e.g. TextViews) when the data changes



Data Binding

- Data Binding library
  - Allows for data from a ViewModel to be directly mapped to specific views in the XML layout file
  - Often used in conjunction with LiveData from a ViewModel
- More than just view binding (e.g. ButterKnife)
  - If you need just view binding:

```
android {
    ...
    viewBinding {
        enabled = true
    }
}
```

Using Data Binding

- Modify Gradle file

```
android {
    dataBinding {enabled = true}
}
```

- Modify XML to have <layout> as a root view
- Add <data> variables in the layout
  - These will be connected with the actual objects
- Example layout:

```
<layout>
<data>
   <variable
      name="myViewModel”
      type="si.uni_lj.fri.lrk.myapp.MainViewModel" />
</data>
<ConstraintLayout>…</ConstraintLayout>
…
</layout>
```

Using Data Binding

- Binding classes are automatically generated
  - E.g. MainFragmentBinding for main_fragment.xml
- Instantiate the binding class
  - E.g.
```
MainFragmentBinding binding;
binding = DataBindingUtil.inflate(inflater, R.layout.main_fragment,
                                  container, false);
```
- Configure data binding variables
  - E.g. `binding.setVariable(viewModel, myViewModel);`
- Binding Expressions
  - Define how Views interact with bound objects
    - E.g. Which function of the bound object is called onClick, which data stored in a ViewModel is show in a TextView

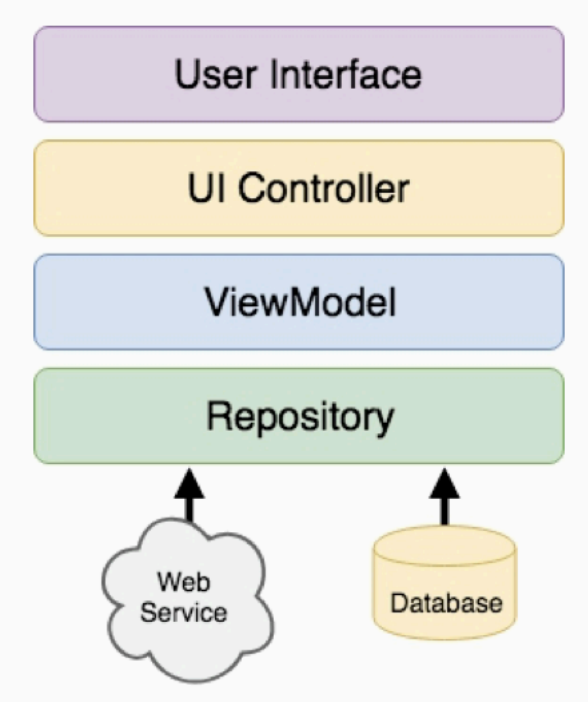
Using Data Binding – Binding Expressions

- One-way
  - The view is updated with the data from the binding, but changes in the view are not propagated to the data (i.e. a ViewModel)
  - E.g. android:text="@{myViewModel.result}"
- Two-way
  - The data is updated in response to changes in the view
  - E.g. android:text="@={myViewModel.result}"
- Event and listener binding
  - E.g. android:onClick="@{()->myViewModel.methodOne()}"

ViewModel, LiveData, Data Binding Example

Managing Data Flow

- Modern Android Architecture
  - MVVM
  - Data can come from multiple source
    - Database
    - SharedPreferences
    - Remote API
  - Use Repository
    - Not a part of Android framework, but a class you create to handle data storing

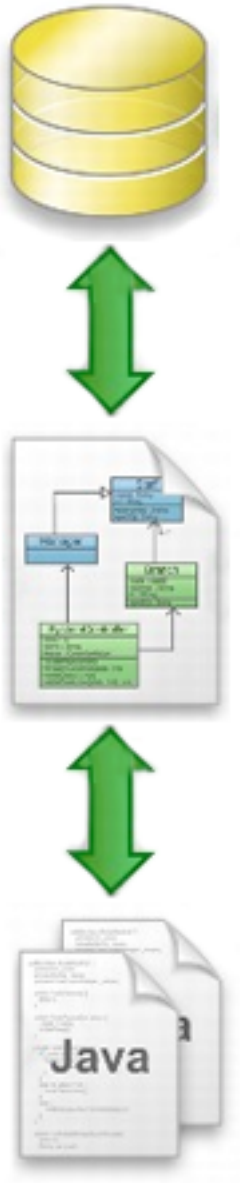


Object-relational Mapping
ORM

- Problem:
  - Object-oriented languages work with objects that can be relatively complex
  - (Relational) databases store and manipulate simple scalar values in tables
  - Converting objects to table entries is cumbersome and prone to errors
- Solution
  - Object-Relational Mapping (ORM)

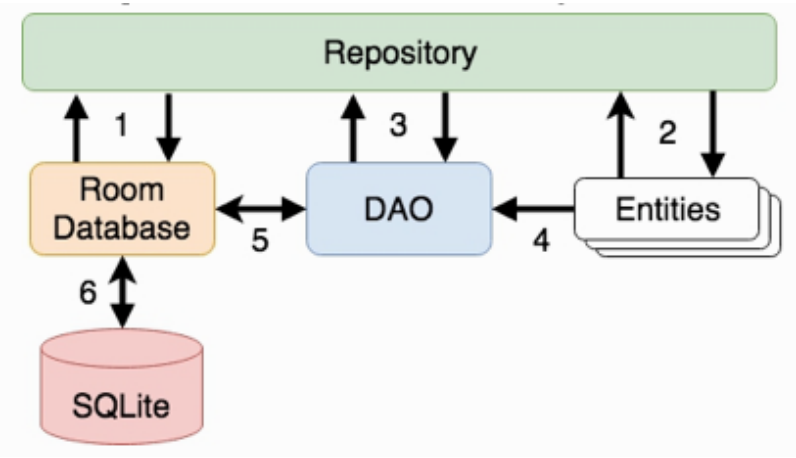
Room Database

- Data storage
  - Underlying Android SQLite database
- Object files (Entities)
  - Annotated Java models
- Data Access Object (DAO)
  - Interface between the database and Java objects
- Note: this is not another database, but a layer over your SQLite DB!



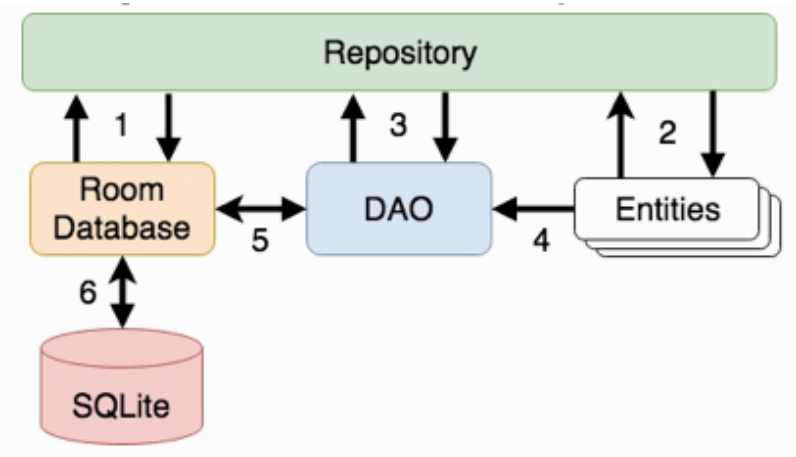
Room Database

- Data Flow
  - Repository gets Room DB instance, obtain references to DAO instances
  - Repository creates entity instances, passes them to the DAO
  - Repository calls methods on the DAO passing through entities to be inserted in the DB and receives entity instances back in response to search queries
  - When DAO has results it packages them into entity objects



Room Database

- Data Flow
  - DAO interacts with Room DB to initiate database operations and handle results
  - Room DB handles all low-level interactions with the underlying SQLite DB, submitting queries and receiving results



Room DB – Entities

- Each DB table needs an associated Entity class
  - Defines the schema for the table
  - A standard Java class with Room annotations

```
public class Customer {

  private int id;
  private String name;

…
```

```
@Entity(tableName="customers"
public class Customer {

  @PrimaryKey(autoGenerate="true")
  @NonNull
  @ColumnInfo(name="customerId")
  private int id;
  @ColumnInfo(name="customerName")
  private String name;

…
```

Room DB – Data Access Object (DAO)

- Provides a way to access the data stored within the database
- A standard Java interface with additional annotations

```
@Dao
public interface CustomerDao {

  @Query("SELECT * FROM customers")
  LiveData<List<Customer>> getAllCustomers();

...
}
```

Queries that will be executed

LiveData enables the Repository to observe changes in the data

Room DB – Database Instance

- Helper class for accessing the SQLite DB
- Extends RoomDatabase + additional annotation

```
@Database(entities = {Customer.class}, version = 1)
public class CustomerRoomDatabase extends RoomDatabase {
    public abstract CustomerDao customerDao();
    private static CustomerRoomDatabase INSTANCE;
    static CustomerRoomDatabase getDatabase(final Context context) {
        if (INSTANCE == null) {
            synchronized (CustomerRoomDatabase.class) {
                if (INSTANCE == null) {
                    INSTANCE = Room.databaseBuilder(
                            context.getApplicationContext(),
                            CustomerRoomDatabase.class, "customer_database")
                            .build();
                }
            }
        }
        return INSTANCE;
    }
```

Room DB – Practical Considerations

- Running on the main thread is considered a bad practice
  - Disabled by default, enable with allowMainThreadQueries()
  - Use Executors instead (see Lab 8)
- Repository should handle Database instantiation

```
public class CustomerRepository {
   private CustomerDao customerDao;
   private CustomerRoomDatabase db;
   public CustomerRepository(Application application) {
      db = CustomerRoomDatabase.getDatabase(application);
      customerDao = db.customerDao();
   }
…
}
```