# Android Architecture Components and Room Database

IT6306 - Mobile Application Development

Level III - Semester 6

# Overview

Welcome to the world of Android Architecture Components (AAC), a set of powerful libraries introduced by Google to help developers build robust and maintainable Android apps. AAC provides pre-built solutions to common app development problems, making it easier and more efficient for developers to create high-quality applications.

One of the key components of AAC is Room database, which provides an abstraction layer over SQLite database, simplifying the process of working with databases in Android apps. In this section, you'll discover how to use Room database and other AAC components to build scalable, maintainable, and efficient Android applications. So, get ready to level up your Android development skills and create apps that stand out in today's competitive market!

# Intended Learning Outcomes

At the end of this section, You will be able to;

1. Understand the key features and benefits of Android Architecture Components (AAC) and how they can simplify the development process of Android apps.

2. Explore how to use Room database and other AAC components to build scalable, maintainable, and efficient Android applications.

3. Develop an understanding of best practices for using AAC components and Room database to create high-quality Android apps that stand out in today's competitive market.

# List of sub topics

5.1. Introduction to Android Architecture Components

- 5.1.1. Activity / Fragment
- 5.1.2. ViewModel
- 5.1.3. Repository

5.2. Room Database

- 5.2.1. Room Overview
- 5.2.2. Components of Room
  - 5.2.2.1. Entity
  - 5.2.2.2. DAO (Data Access Object)
  - 5.2.2.3. Database

5.3. Lifecycle-aware Components

- 5.3.1. Usecases and Lifecycle library
- 5.3.2. Lifecycle Events and Observers
- 5.3.3. LiveData

# 5.1. Introduction to Android Architecture Components

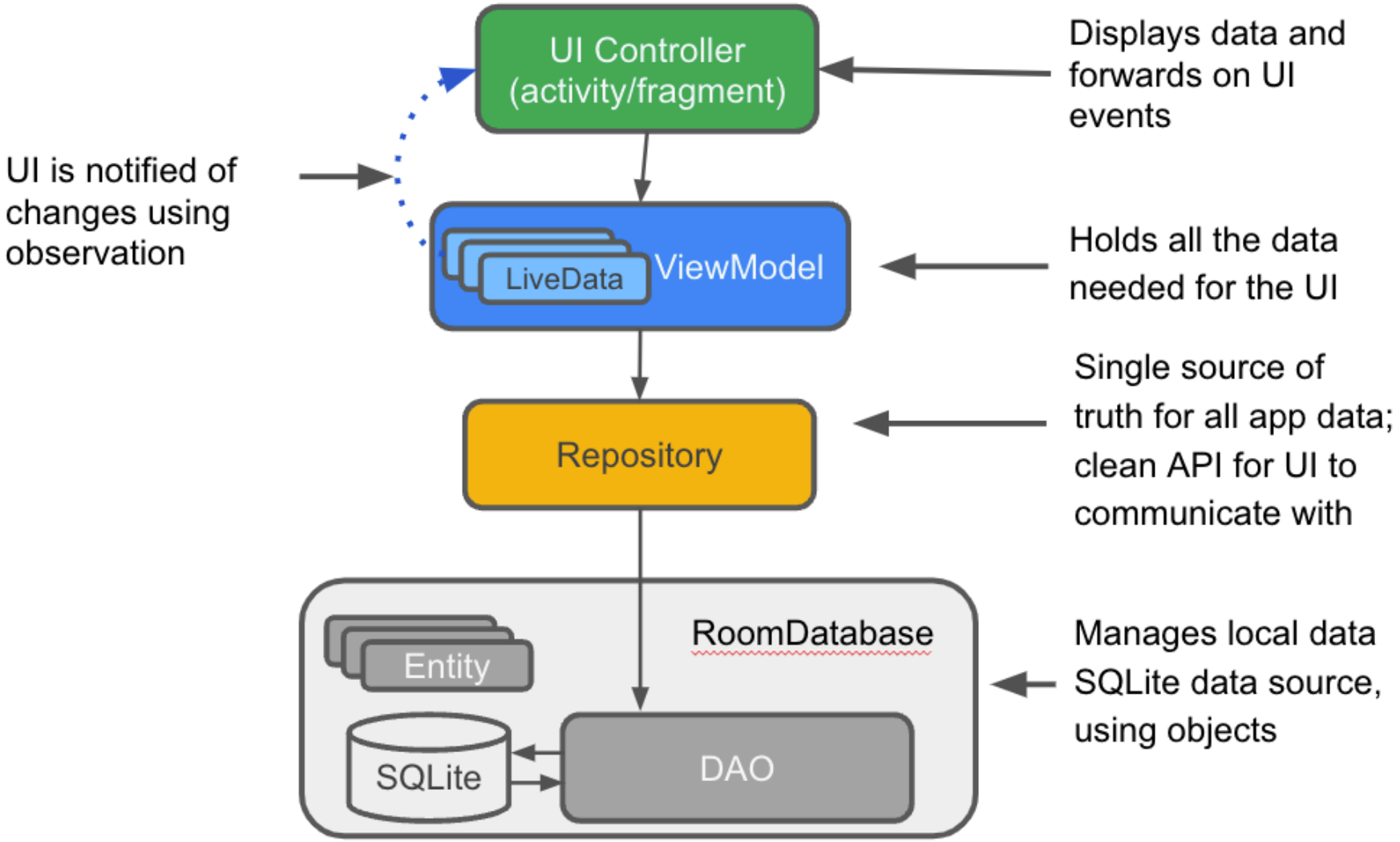
## Architecture Components

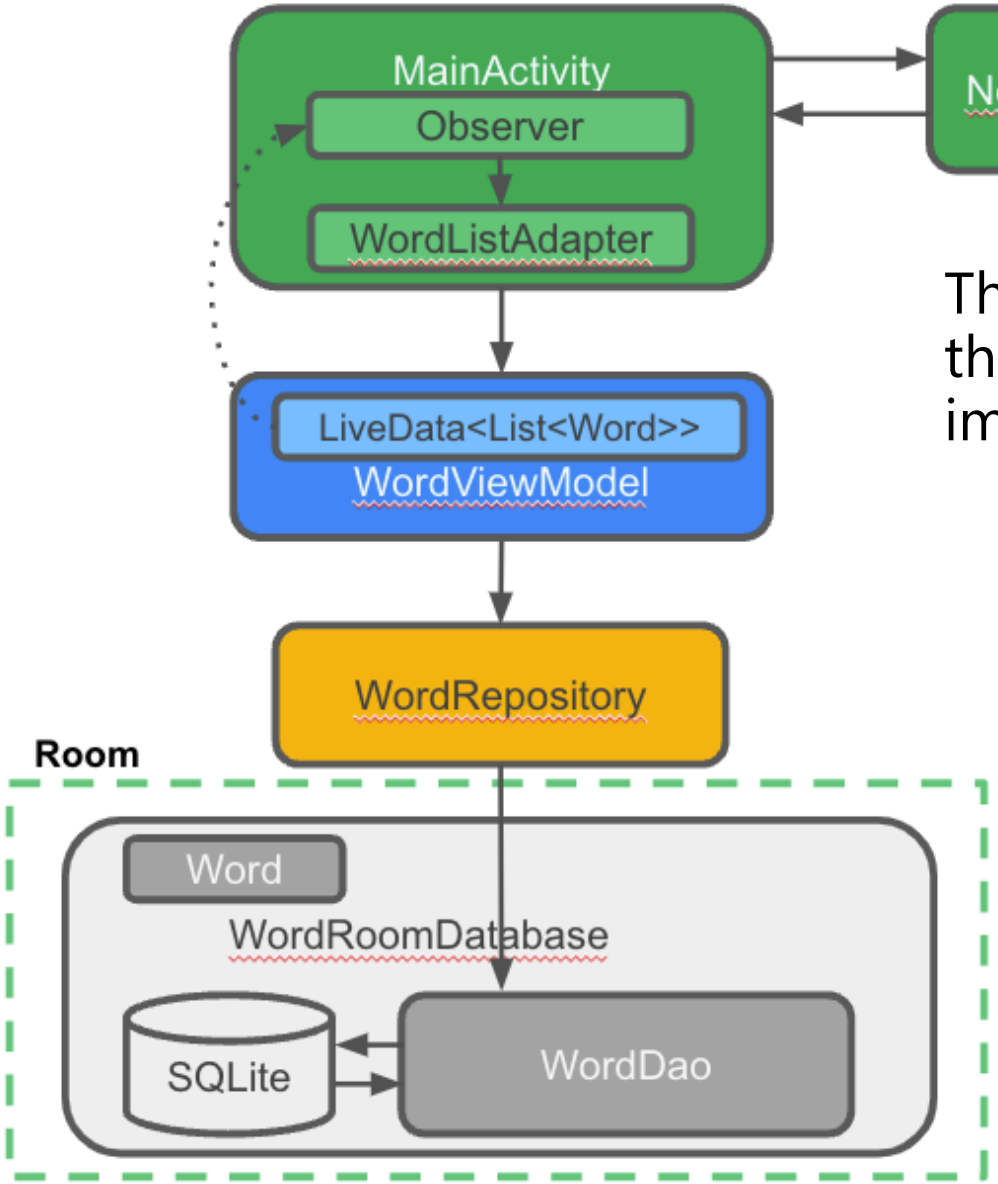
A set of Android libraries for structuring your app in a way that is robust, testable, and maintainable.

- Consist of best architecture practices and libraries
- Encourage recommended app architecture
- **A LOT LESS** boilerplate code
- Testable because of clear separation
- Fewer dependencies
- Easier to maintain

References Guide to app architecture

# Overview

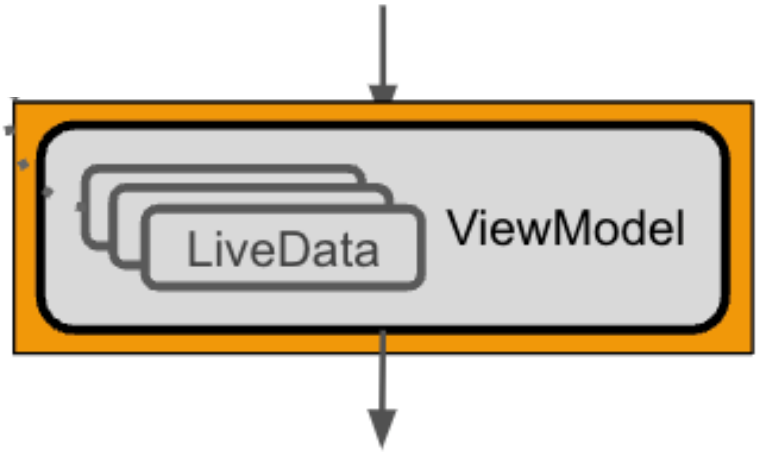




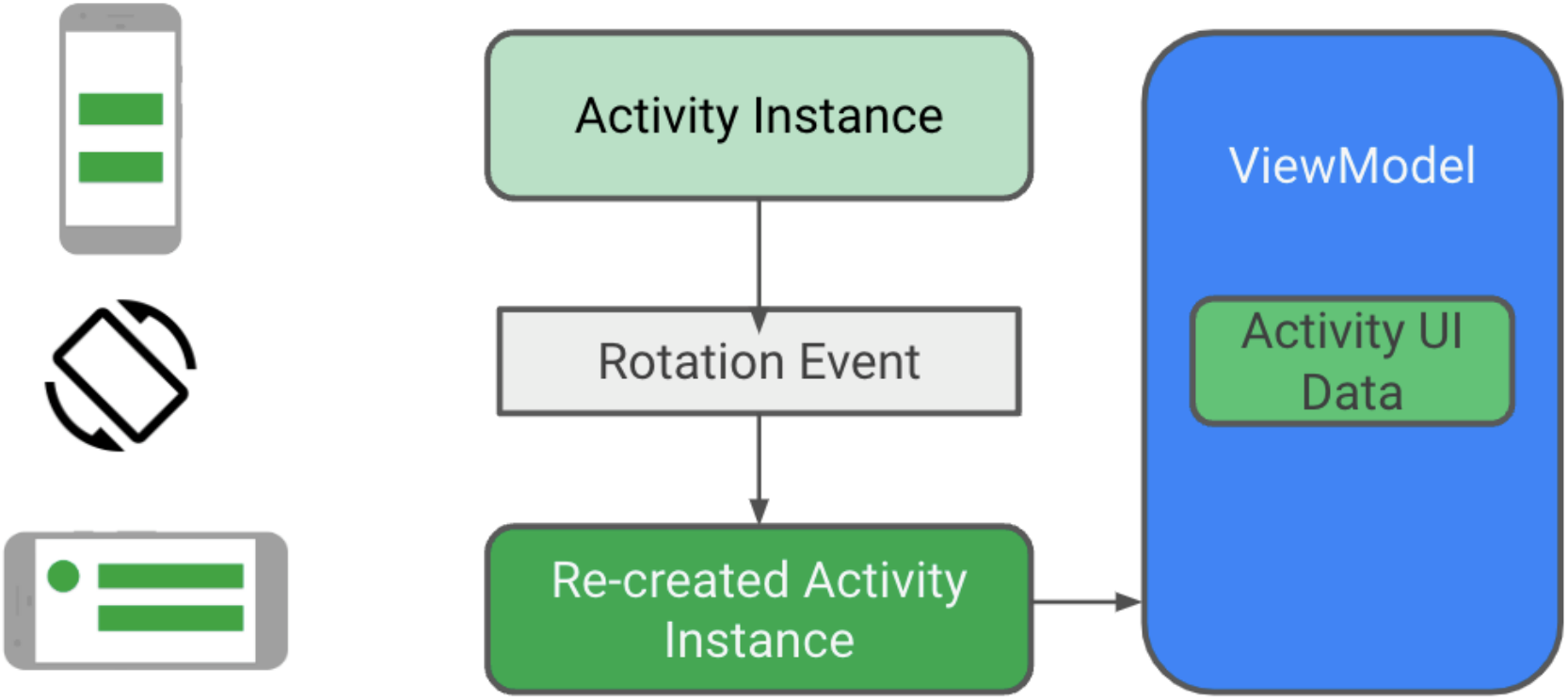

The RoomWordsSample app that you build in the practical implements this architecture

# 5.1.2. ViewModel

- View models are objects that provide data for UI components and survive configuration changes.
- Provides data to the UI
- Survives configuration changes
- You can also use a ViewModel to share data between fragments
- Part of the lifecycle library



# Survives configuration changes



# ViewModel serves data

- `ViewModel serves data to the UI`
- Data can come from Room database or other sources
- ViewModel's role is to return the data, it can get help to find or generate the data

# Best practice to use repository

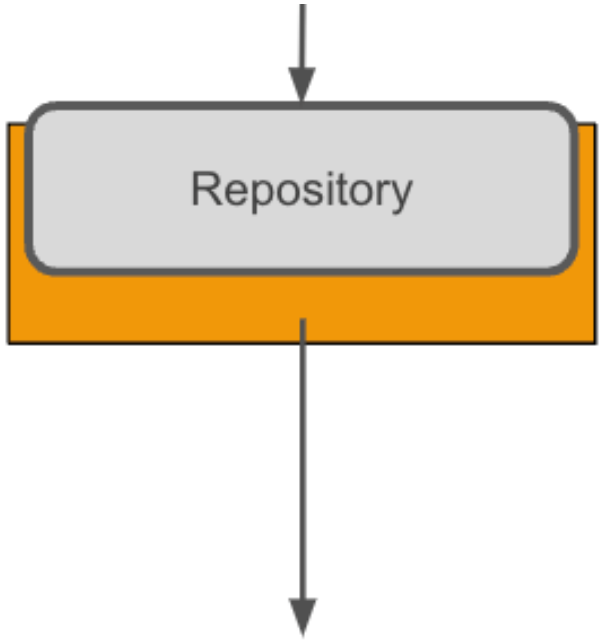
- Use a repository to do the work to get the data
- Keeps `ViewModel` as clean interface between app and data

# Restaurant analogy

| | | |
|---|---|---|
| • Customer requests meal from server |  | • UI requests data from ViewModel |
| • Server takes order to chefs |  | • ViewModel asks Repository for data |
| • Chefs prepare meal |  | • Repository gets data |
| • Server delivers meal to customer |  | • ViewModel returns data to UI |

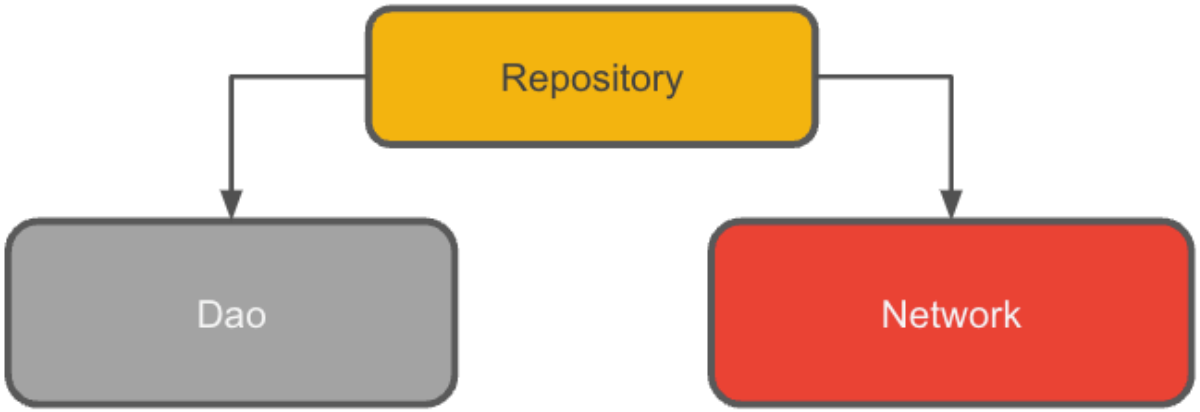
# 5.1.3. Repository

- Best practice, not part of Architecture Components libraries
- Implement repository to provide single, clean API to app data
- Use repository to fetch data in the background
- Analogy: chefs prepare meals behind the scenes



# Multiple backends

- Potentially, repository could manage query threads and allow you to use multiple backends
- **Example**: in Repository, implement logic for deciding whether to fetch data from a network or use results cached in the database



# 5.2. Room Database

## 5.2.1. Room overview

Room is a robust SQL object mapping library

- Generates SQLite Android code
- Provides a simple API for your database

# 5.2.2. Components of Room

**Entity**:

Defines schema of database table.

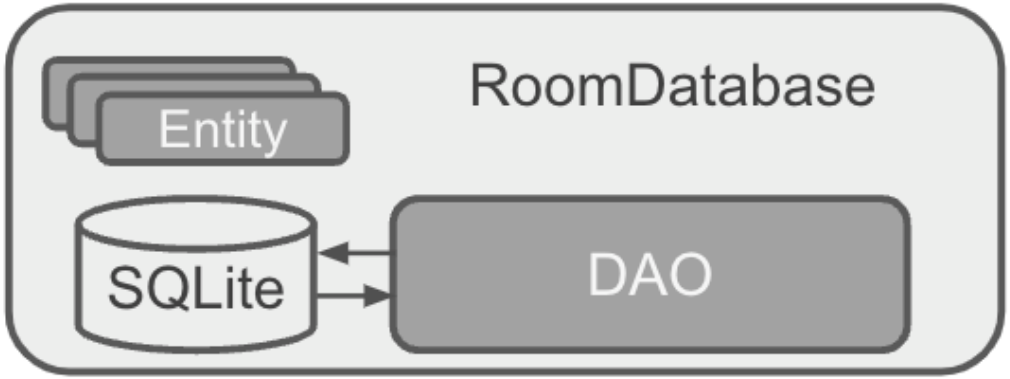
**DAO**: Database Access Object

Defines read/write operations for database.

**Database**:

A database holder.

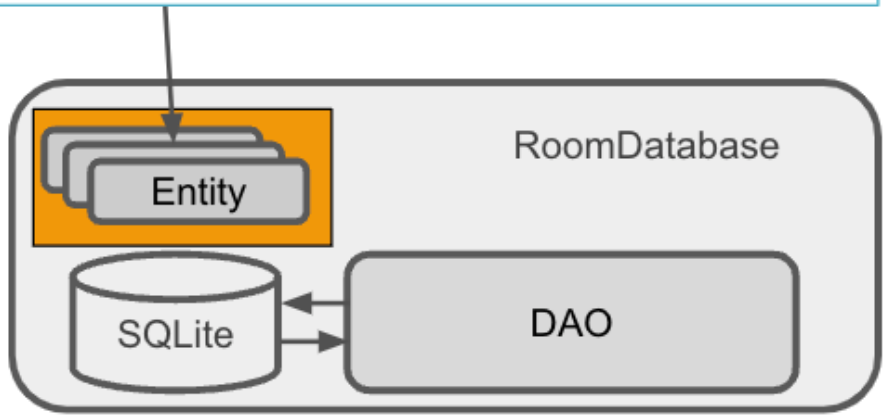
Used to create or connect to database



# 5.2.2.1. Entity

- Entity instance = row in a database table
- Define entities as POJO classes
- 1 instance = 1 row
- Member variable = column name

```
public class Person {
    private int uid;
    private String firstName;
    private String lastName;
}
```



# Entity instance = row in a database table

```
public class Person {
    private int uid;
    private String firstName;
    private String lastName;
}
```

| uid | firstName | lastName |
|---|---|---|
| 12345 | Aleks | Becker |
| 12346 | Jhansi | Kumar |

# Annotate entities

```
@Entity
public class Person {

    @PrimaryKey (autoGenerate=true)
    private int uid;

    @ColumnInfo(name = "first_name")
    private String firstName;

    @ColumnInfo(name = "last_name")
    private String lastName;

    // + getters and setters if variables are
private.
}
```

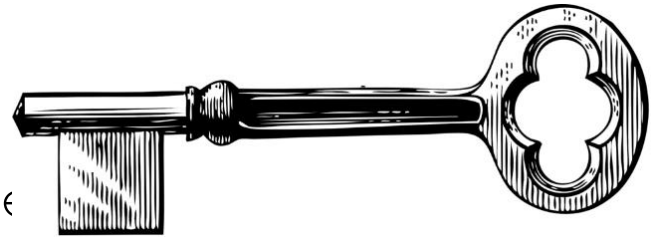
# @Entity annotation

```
@Entity(tableName = "word_table")
```

- Each @Entity instance represents an entity/row in a table
- Specify the name of the table if different from class name

# @PrimaryKey annotation

```
@PrimaryKey (autoGenerate=true
```



- Entity class must have a field annotated as primary key
- You can auto-generate unique key for each entity
- See Defining data using Room entities

# @NonNull annotation

`@NonNull`


- Denotes that a parameter, field, or method return value can never be null
- Use for mandatory fields
- Primary key must use `@NonNull`

# @ColumnInfo annotation

```
@ColumnInfo(name = "first_name")
private String firstName;
@ColumnInfo(name = "last_name")
private String lastName;
```

- Specify column name if different from member variable name

# Getters, setters

Every field that's stored in the database must

- be public

OR

- have a "getter" method

… so that Room can access it

# Relationships

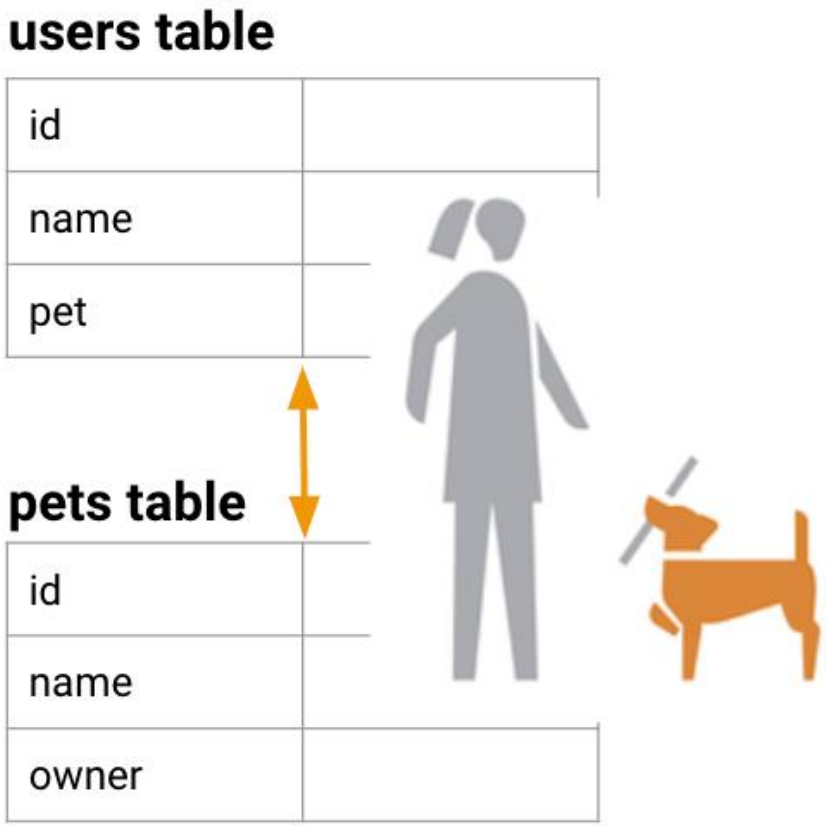
Use @Relation annotation to define related entities

Queries fetch all the returned object's relations

***Many more annotations***

For more annotations, see

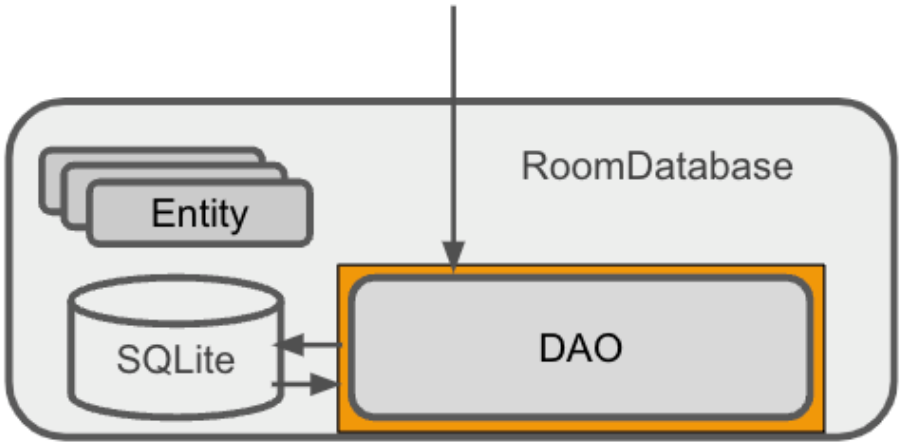
Room package summary reference



# 5.2.2.2. Data access object (DAO)

Use data access objects, or DAOs, to access app data using the Room persistence library

- DAO methods provide abstract access to the app's database
- The data source for these methods are entity objects
- DAO must be interface or abstract class
- Room uses DAO to create a clean API for your code



# Example DAO

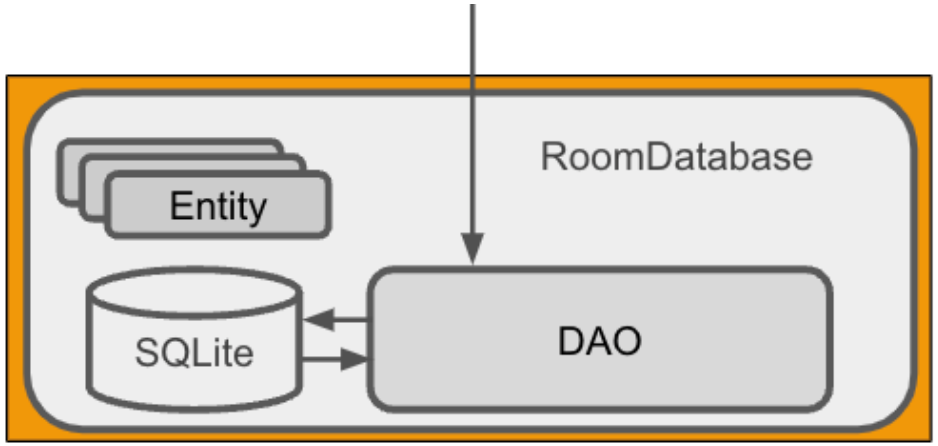
```
@Query("DELETE FROM word_table")
void deleteAll();

@Query("SELECT * from word_table ORDER BY word ASC")
List<Word> getAllWords();

@Query("SELECT * FROM word_table WHERE word LIKE :word ")
public List<Word> findWord(String word);
```

# Example queries


```
@Query("DELETE FROM word_table")
void deleteAll();

@Query("SELECT * from word_table ORDER BY word ASC")
List<Word> getAllWords();

@Query("SELECT * FROM word_table WHERE word LIKE :word ")
public List<Word> findWord(String word);
```

# 5.2.2.3. Database

- Room is a robust SQL object mapping library
- Generates SQLite Android code
- Room works with DAO and Entities
- Entities define the database schema
- DAO provides methods to access database



# Creating Room database

- Create public abstract class extending RoomDatabase
- Annotate as @Database
- Declare entities for database schema and set version number

```
@Database(entities = {Word.class}, version = 1)
public abstract class WordRoomDatabase extends RoomDatabase
```

References
https://developer.android.com/reference/android/arch/persistence/room/Database

# Room class example

```
@Database(entities = {Word.class}, version = 1)
public abstract class WordRoomDatabase
                      extends RoomDatabase {

    public abstract WordDao wordDao();

    private static WordRoomDatabase INSTANCE;

    // ... create instance here
}
```

*Entity defines DB schema*

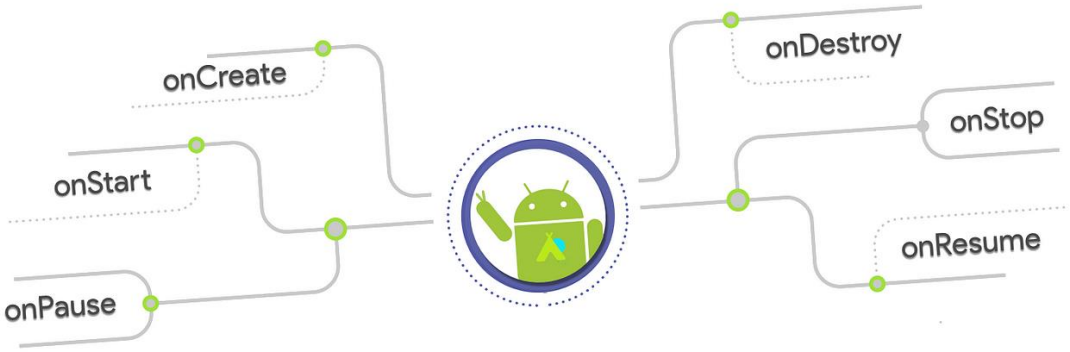
*DAO for database*

*Create database as singleton instance*

# 5.3. Lifecycle-aware Components

Instead of managing lifecycle-dependent components in the activity's lifecycle methods, `onStart(), onStop(),` and so on, you can make any class react to lifecycle events

- Lifecycle-aware components perform actions in response to a change in the lifecycle status of another component
- For example, a listener could start and stop itself in response to an activity starting and stopping



# 5.3.1. Usecases and Lifecycle library

## Use cases

- Switch between coarse and fine-grained location updates depending on app visibility
- Stop and start video buffering
- Stop network connectivity when app is in background
- Pause and resume animated drawables

## Lifecycle library

- Import the android.arch.lifecycle package
- Provides classes and interfaces that let you build lifecycle-aware components that automatically adjust their behavior based on lifecycle state of activity or fragment
- See Handling Lifecycles with Lifecycle-Aware Components

# 5.3.2. Lifecycle Events and Observers

## LifecycleObserver interface

- LifecycleObserver has an Android lifecycle.
- It does not have any methods, instead, uses OnLifecycleEvent annotated methods.

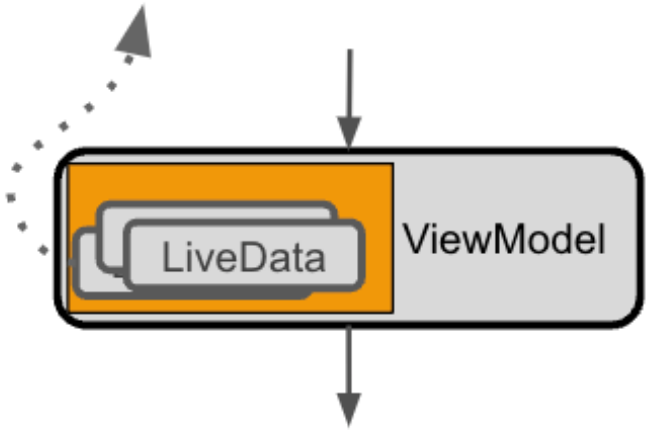
## @OnLifecycleEvent

```
@OnLifecycleEvent indicates life cycle methods

        @OnLifecycleEvent(Lifecycle.Event.ON_START)
        public void start() {...}

        @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
        public void start() {...}
```

See Lifecycle.event reference for more lifecycle events

# 5.3.3. LiveData

- LiveData is a data holder class that is aware of lifecycle events. It keeps a value and allows this value to be observed.
- Use LiveData to keep your UI up to date with the latest and greatest data.
- LiveData is observable data
- Notifies observer when data changes
- Is lifecycle aware: knows when device rotates or app stops



# Use LiveData to keep UI up to date

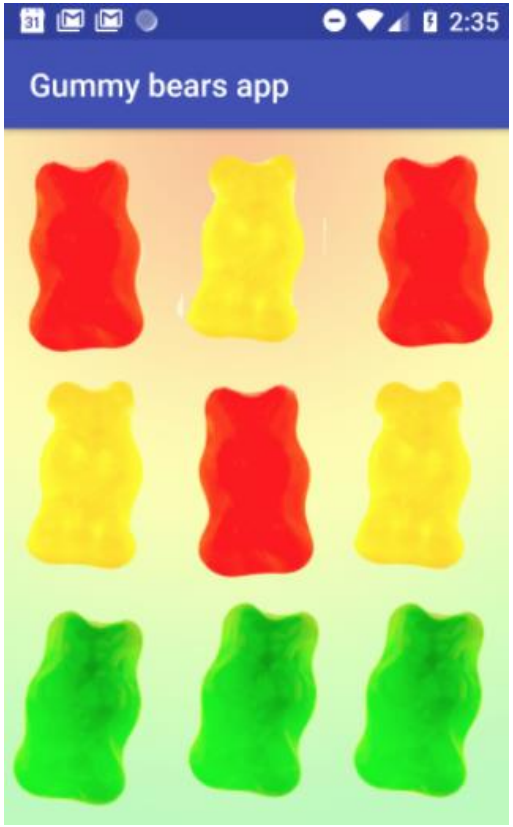
- Create an observer that observes the LiveData
- LiveData notifies Observer objects when the observed data changes
- Your observer can update the UI every time the data changes

## Creating LiveData

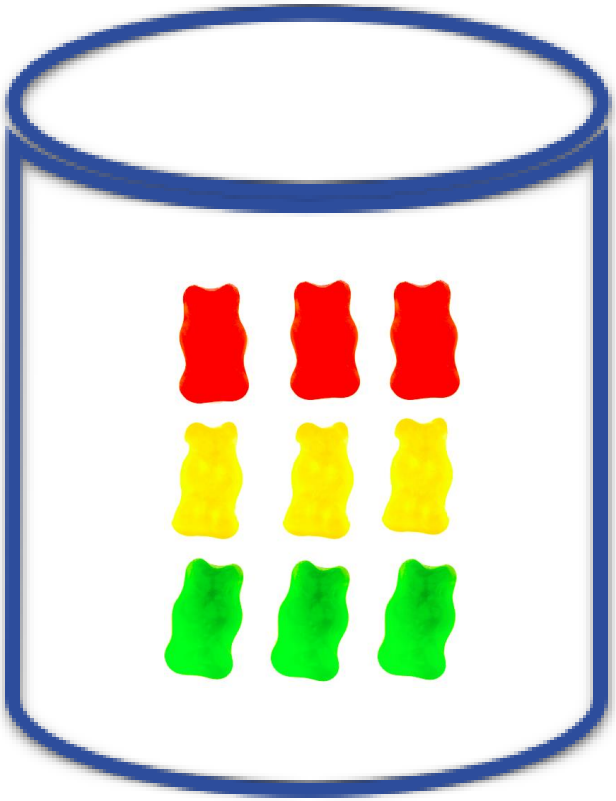
*To make data observable, return it as LiveData:*

```
@Query("SELECT * from word_table)
LiveData<List<Word>> getAllWords();
```

# Using LiveData with Room




Room generates all the code to update the LiveData when the database is updated



# Passing LiveData through layers

When you pass live data through the layers of your app architecture, from a Room database to your UI, that data must be LiveData in all layers:

- DAO
- ViewModel
- Repository

# Passing LiveData through layers

- DAO:

```
@Query("SELECT * from word_table")
LiveData<List<Word>> getAllWords();
```

- Repository:

```
LiveData<List<Word>> mAllWords =
                    mWordDao.getAllWords();
```

- ViewModel:

```
LiveData<List<Word>> mAllWords =
                    mRepository.getAllWords();
```

# Observing LiveData

- Create the observer in `onCreate()` in the Activity
- Override `onChanged()` in the observer to update the UI when the data changes

When the LiveData changes, the observer is notified and its `onChanged()` is executed

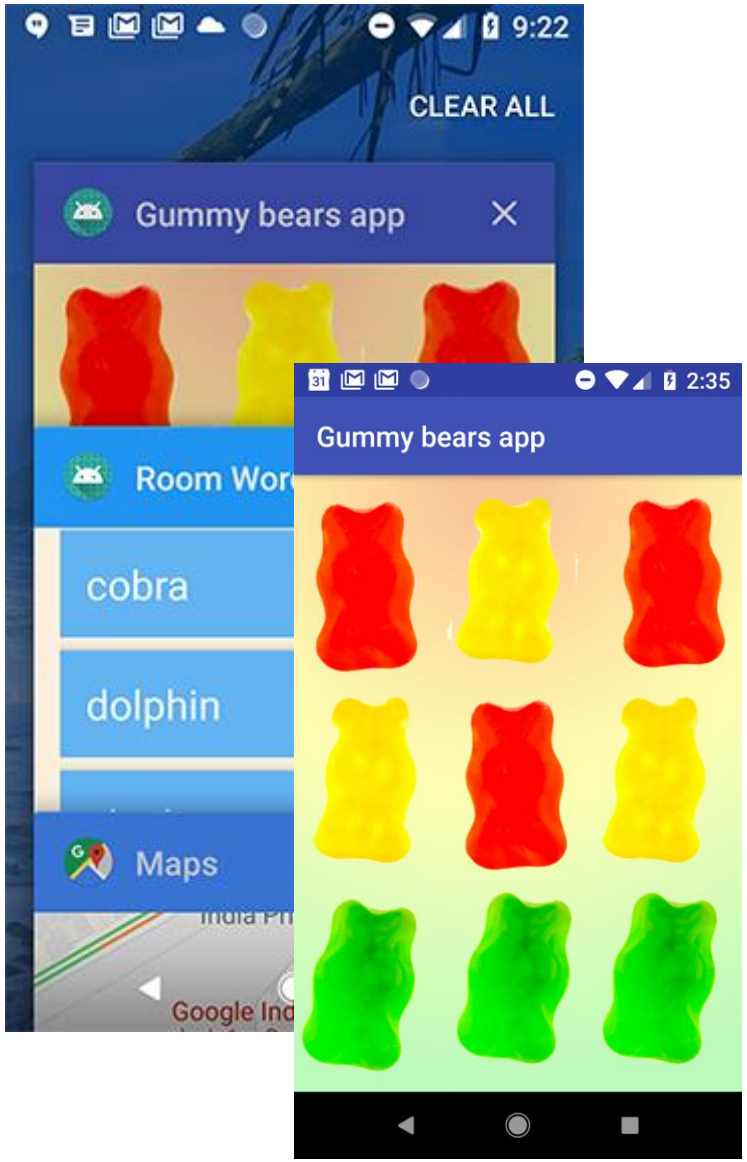
# No memory leaks

•Observers are bound to Lifecycle objects
which are objects that have an Android Lifecycle

•Observers clean up after themselves when their associated lifecycle is destroyed

# LiveData is always up to date

- If a lifecycle object becomes inactive, it gets the latest data when it becomes active again
  - Example: an activity in the background gets the latest data right after it returns to the foreground



# LiveData handles configuration changes

If an activity or fragment is re-created due to a configuration change such as device rotation, the activity or fragment immediately receives the latest available data

# Share resources

- You can extend a LiveData object using the singleton pattern, for example for services or a database
- The LiveData object connects to the system service once, and then any observer that needs the resource can just watch the LiveData object
- See Extend LiveData

References for diagrams – https://developer.android.com/courses/fundamentals-training/overview-v2