COM4510/6510
Software Development for Mobile Devices

Lecture 5: Persisting Data (Part 1)

Temitope (Temi) Adeosun
The University of Sheffield
t.adeosun@sheffield.ac.uk

Lecture Overview

- Week 2: Intro to Kotlin
- Week 3: Lifecycle and Layouts
- Week 4: (Architectural) Design Patterns
- **Week 5: Persisting Data**
- Week 6: Architectural Components
- Week 7: Sensing in Android
- Week 8: Background and Foreground Services
- Week 10: Context
- Week 11: Releasing Apps
- Week 12: Guest lecture

Lecture Overview

- Part 1:
  - Persisting Data
  - Internal/External Storage
- Part 2:
  - ROOM
  - Keeping Your App Responsive
- Lab tutorial :
  - Using ROOM to retrieve data

Persisting Data

- Android provides **four ways** of persisting data:
  - **Key-Value sets**
    - for simple cases such as **user preferences** or for remembering that some events have happened
  - **Files**
    - to store data that will be accessed sequentially
  - **Rooms**
    - an object orientated abstraction of a relational database
      - to be preferred to a plain SQLite DB
  - **SQLite databases**

Persisting Data

- The solution you choose depends on your specific needs
  - **How much space does your data require ?**
  - **How reliable does data access need to be ?**
  - **What kind of data do you need to store ?**
  - **Should the data be private to your app ?**

SharedPreferences

- For a relatively small collection of key-values to save
  - use the **SharedPreferences APIs**.
- A **SharedPreferences** object points to a file containing key-value pairs
  - It provides simple methods to **read** and **write** them.
- Each **SharedPreferences** file is managed by the framework
  - It can be **private** or **shared**.

Handles

- Create a new shared preference file or access an existing one
  - **getSharedPreferences**(String fileName)
    - if multiple shared preference files identified by name are needed
    - You can call this from any **Context** in your app
    - It requires the preference file name
- **getPreferences()**
  - to have just one shared preference file for **an activity**
    - Note: different activities in your app will have different preferences
  - No need to supply a name as it is **shared by the activity**

```
val sharedPref = this@MainActivity.getSharedPreferences("com.example.myapp.PREF_KEY",
Context.MODE_PRIVATE)
```

- it is suggested that the name contains your **package name**, e.g:
  - com.example.myapp.PREFERENCE_FILE_KEY
  - Note however that it is bad practice to insert the string directly in the code. So use:

```
R.string.preference_file_key
```

- and set it under string.xml as
  - “com.example.myapp.PREFERENCE_FILE_KEY"

- If you have just one preference file:

```
this@MainActivity.getSharedPreferences("com.example.myapp.PREF_KEY", Context.MODE_PRIVATE)
```

editor.commit()

Writing Preferences

- **To write**:
  - get the preference file

```
val sharedPref = this@MainActivity.getSharedPreferences("com.example.myapp.PREF_KEY", Context.MODE_PRIVATE)
```

  - create a **SharedPreferences.Editor** by calling *edit()* on your SharedPreferences

```
val editor = sharedPref.edit()
```

- **Pass the keys and values** as you would do to a HashSet but specifying a type,
  - e.g. as putInt() and putString()

```
editor.putInt(R.string.image_title_label.toString(), value_to_store)
```

- Then call **commit()** to save the changes

```
editor.commit()
```

All together

```
val sharedPref =
this@MainActivity.getSharedPreferences("com.example.myapp.PREF_KEY",
Context.MODE_PRIVATE)

val editor = sharedPref.edit()
editor.putInt(R.string.image_title_label.toString(), value_to_store)
editor.commit()
```

(never forget the commit or the file content will not be changed)

Retrieval

- Use methods such as **getInt()** and **getString()**
  - Provide the key
  - and optionally a default value to return if the key isn't present.

```
val sharedPref = this@MainActivity.getPreferences(Context.MODE_PRIVATE)
val defaultValue = 10
val highScore = sharedPref.getInt("AGE_KEY", defaultValue)
```

More elegantly:

```
var sharedPref = this@MainActivity.getPreferences(Context.MODE_PRIVATE)
var defaultValue = resources.getIntArray(R.string.saved_age_default)
val highScore = sharedPref.getInt(R.string.saved_age.toString(), defaultValue);
```

Files

- Android uses a **file system** that's similar to disk-based file systems on other platforms.
- A **File object** is suited to reading or writing **large amounts** of data in start-to-finish order without skipping around
- For example, it's good for **image files** or anything exchanged over a network

Internal/External Storage

- All Android devices have two file storage areas: "**internal**" and "**external**" storage
- These names come from the early days of Android, when most devices offered
  - built-in non-volatile memory (internal storage),
  - a removable storage medium such as a micro SD card (external storage).
- Some devices still divide the **permanent storage space** into "internal" and "external" partitions (even without a removable storage medium)

External/Internal?

- **Internal**
  - Always available
  - Files are accessible by only your app
  - Uninstalling your app, remove all your app's files
  - It is best when be sure neither the users or other apps can access your files.
- **External**
  - Not always available
  - World-readable, may outside of your control
  - Uninstalling your app, system only removes files if you save them in the directory from **getExternalFileDir()**
  - It is best when not require access restrictions and can share with other apps

Permissions

Permissions and access to external storage

Android defines the following storage-related permissions: `READ_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE`, and `MANAGE_EXTERNAL_STORAGE`.

On earlier versions of Android, apps needed to declare the `READ_EXTERNAL_STORAGE` permission to access any file outside the app-specific directories on external storage. Also, apps needed to declare the `WRITE_EXTERNAL_STORAGE` permission to write to any file outside the app-specific directory.

More recent versions of Android rely more on a file's purpose than its location for determining an app's ability to access, and write to, a given file. In particular, if your app targets Android 11 (API level 30) or higher, the `WRITE_EXTERNAL_STORAGE` permission doesn't have any effect on your app's access to storage. This purpose-based storage model improves user privacy because apps are given access only to the areas of the device's file system that they actually use.

Android 11 introduces the `MANAGE_EXTERNAL_STORAGE` permission, which provides write access to files outside the app-specific directory and `MediaStore`. To learn more about this permission, and why most apps don't need to declare it to fulfill their use cases, see the guide on how to manage all files on a storage device.

https://developer.android.com/training/data-storage

Permissions – Scoped Storage

Scoped storage

To give users more control over their files and to limit file clutter, apps that target Android 10 (API level 29) and higher are given scoped access into external storage, or *scoped storage*, by default. Such apps have access only to the app-specific directory on external storage, as well as specific types of media that the app has created.

> ★ **Note:** If your app requests a storage-related permission at runtime, the user-facing dialog indicates that your app is requesting broad access to external storage, even when scoped storage is enabled.

Use scoped storage unless your app needs access to a file that's stored outside of an app-specific directory and outside of a directory that the `MediaStore` APIs can access. If you store app-specific files on external storage, you can make it easier to adopt scoped storage by placing these files in an app-specific directory on external storage. That way, your app maintains access to these files when scoped storage is enabled.

To prepare your app for scoped storage, view the storage use cases and best practices guide. If your app has another use case that isn't covered by scoped storage, file a feature request. You can temporarily opt-out of using scoped storage.

https://developer.android.com/training/data-storage

Saving files

Save a File on Internal Storage

When saving a file to internal storage, you can acquire the appropriate directory as a `File` by calling one of two methods:

`getFilesDir()` Kotlin - fileDir

> Returns a `File` representing an internal directory for your app.

`getCacheDir()` Kotlin - catchDir

> Returns a `File` representing an internal directory for your app's temporary cache files. Be sure to delete each file once it is no longer needed and implement a reasonable size limit for the amount of memory you use at any given time, such as 1MB. If the system begins running low on storage, it may delete your cache files without warning.

To create a new file in one of these directories, you can use the `File()` constructor, passing the `File` provided by one of the above methods that specifies your internal storage directory. For example:

```
var file = File(this.filesDir, "filename")
```

Alternatively, you can call `openFileOutput()` to get a `FileOutputStream` that writes to a file in your internal directory. For example, here's how to write some text to a file:

```
String filename = "myfile";
String string = "Hello world!";
FileOutputStream outputStream;

try {
  outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
  outputStream.write(string.getBytes());
  outputStream.close();
} catch (Exception e) {
  e.printStackTrace();
}
```

Or, if you need to cache some files, you should instead use `createTempFile()`. For example, the following method extracts the file name from a `URL` and creates a file with that name in your app's internal cache directory:

```
public File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName = Uri.parse(url).getLastPathSegment();
        file = File.createTempFile(fileName, null, context.getCacheDir());
    } catch (IOException e) {
        // Error while creating file
    }
    return file;
}
```

Delete a File

You should always delete files that you no longer need. The most straightforward way to delete a file is to have the opened file reference call `delete()` on itself.

```
myFile.delete();
```

If the file is saved on internal storage, you can also ask the `Context` to locate and delete a file by calling `deleteFile()`:

```
myContext.deleteFile(fileName);
```

> **Note:** When the user uninstalls your app, the Android system deletes the following:
>
> - All files you saved on internal storage
> - All files you saved on external storage using `getExternalFilesDir()`.
>
> However, you should manually delete all cached files created with `getCacheDir()` on a regular basis and also regularly delete other files you no longer need.

The University Of Sheffield.

Delete a File

You should always delete files that you no longer need. The most straightforward way to delete a file is to have the opened file reference call `delete()` on itself.

```
myFile.delete();
```

If the file is saved on internal storage, you can also ask the `Context` to locate and delete a file by calling `deleteFile()`:

```
myContext.deleteFile(fileName);
```

- Note: When the user uninstalls your app, the Android System deletes the following:
  - All files you saved on your internal storage
  - All files you saved to scoped storage

Room

An entity oriented abstraction of SQLite databases

For Room, add:

- `implementation "android.arch.persistence.room:runtime:1.0.0"`
- `annotationProcessor "android.arch.persistence.room:compiler:1.0.0"`
- For testing Room migrations, add:
  - `testImplementation "android.arch.persistence.room:testing:1.0.0"`
- For Room RxJava support, add:
  - `implementation "android.arch.persistence.room:rxjava2:1.0.0"`

```
dependencies {
  def room_version = "2.2.5"

  implementation "androidx.room:room-runtime:$room_version"
  annotationProcessor "androidx.room:room-compiler:$room_version"

  // optional - RxJava support for Room
  implementation "androidx.room:room-rxjava2:$room_version"

  // optional - Guava support for Room, including Optional and ListenableFutur
  implementation "androidx.room:room-guava:$room_version"

  // optional - Test helpers
  testImplementation "androidx.room:room-testing:$room_version"
}
```

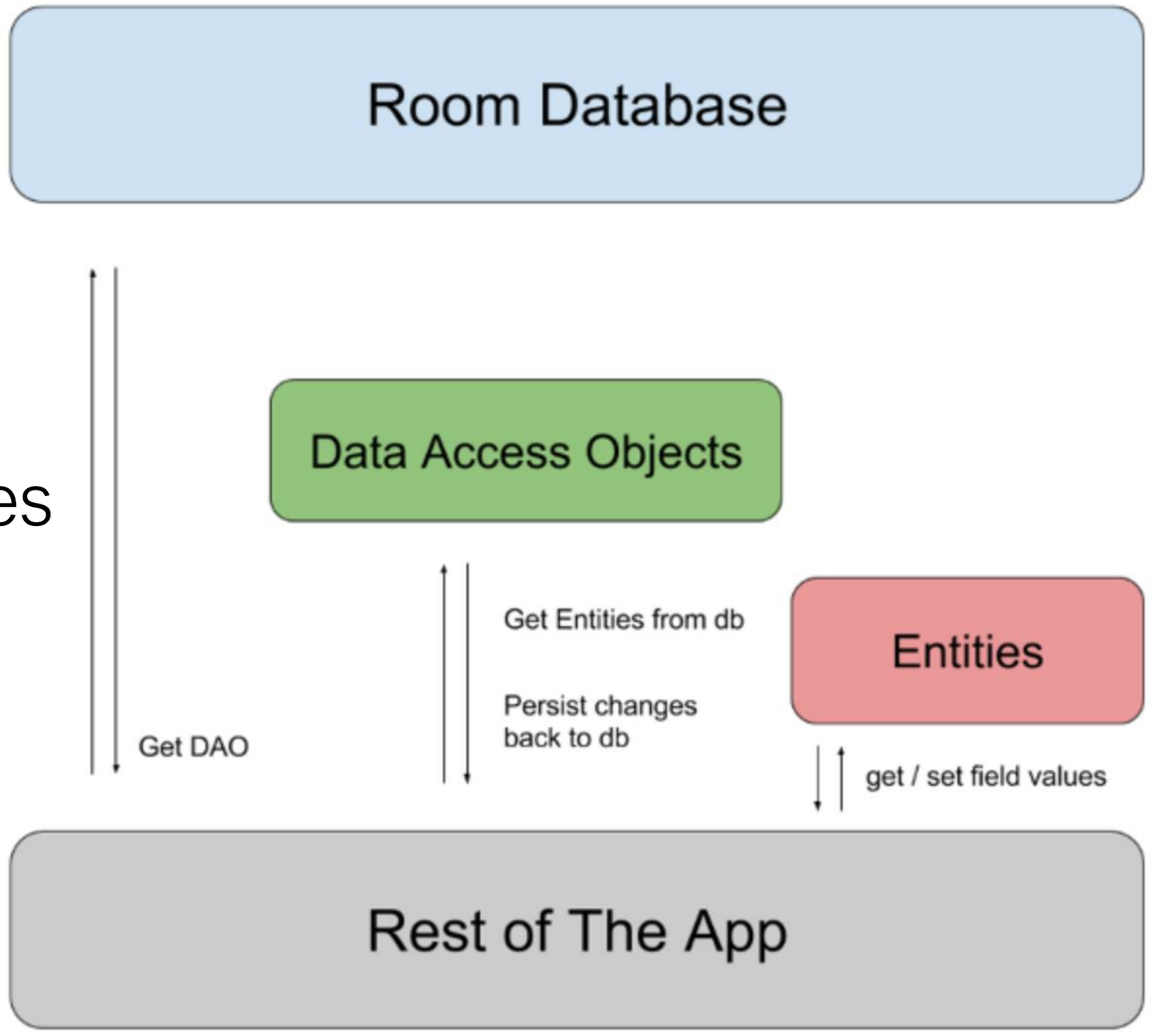
Room

- Room provides an **abstraction layer** over **SQLite**
  - to allow fluent database access while harnessing the full power of SQLite.
  - it enables writing code that is leaner than with standard SQL
- Apps that handle **non-trivial amounts of structured data** can benefit greatly from persisting that data locally
- The most common use case is to **cache relevant pieces of data**
- That way, when the device **cannot access the network**, the user can still browse that content while they are **offline**.
- Any user-initiated content changes are then **synced** to the server after the device is back online.

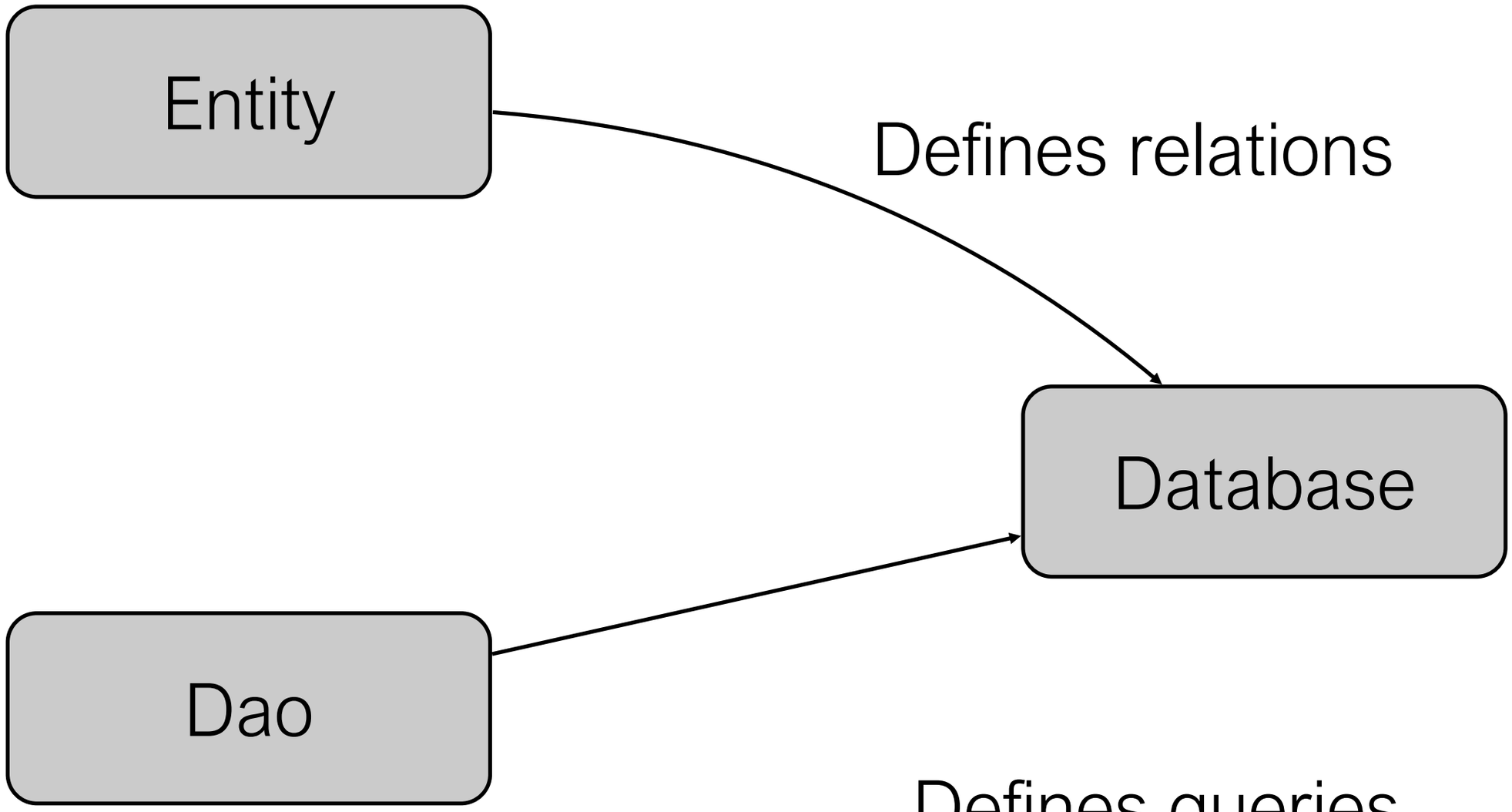
- There are 3 **major components** in Room:
- **Database**:
  - the database holder
  - it serves as the main access point for the underlying connection to the relational data
  - The class that's annotated with **@Database** should satisfy the following conditions:
    - **Be an abstract class that extends RoomDatabase**
    - **Include the list of entities associated with the database within the annotation**
    - **Contain an abstract method that has 0 arguments and returns the class that is annotated with @Dao**
  - At runtime, you can acquire an instance of Database by calling Room.databaseBuilder() or Room.inMemoryDatabaseBuilder()
- **Entity**: Represents a **table** within the database.
- **DAO**: Contains the **methods** used for accessing the database.

Interaction with app

queries



The University Of Sheffield.



Entity

- It is a normal class annotated with Room annotations

The following code snippet contains a sample database configuration with 1 entity and 1 DAO:

`User.java`

```
@Entity
public class User {
    @PrimaryKey
    private int uid;

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

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

    // Getters and setters are ignored for brevity,
    // but they're required for Room to work.
}
```

@ColumnInfo allows for different field names in entity and DB

Unique elements

- Sometimes, certain fields or groups of fields in a database must **be unique**.

```
@Entity(indices = {@Index(value = {"first_name", "last_name"},
        unique = true)})
class User {
    @PrimaryKey
    public int id;

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

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

    @Ignore
    Bitmap picture;
}
```

You can enforce this uniqueness property by setting the unique property of an @Index annotation to true

DAO

- It specifies the **data queries** and it maps the results to the data
- it must be defined as an **interface**

UserDao.java

```
@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND "
           + "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);

    @Insert
    void insertAll(User... users);

    @Delete
    void delete(User user);
}
```

it is a query

it returns a list of User

it returns a single User

it inserts a list of User

it deletes a User

DAOs

- To access your app's data using ROOM, you work with **data access objects, DAOS**.
- By accessing a database using a DAO class instead of query builders or direct queries, you can separate different components of your database architecture.
- DAOs allow you to easily mock database access as you test you app

DAOs

- Can be either an **interface** or an **abstract class**.
- If it is an abstract class, it can optionally have a constructor that takes a **RoomDatabase** as its only parameter.
- Room creates each DAO implementation at compile time.
- **Room does not support database access on the main thread** unless you have called ***allowMainThreadQueries()***

Database

- It must be Abstract
- it must extend RoomDatabase
- It declares an abstract Dao

AppDatabase.java

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

After creating the files above, you get an instance of the created database using the following code:

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database-name").build();
```

https://developer.android.com/training/data-storage/room/index.html

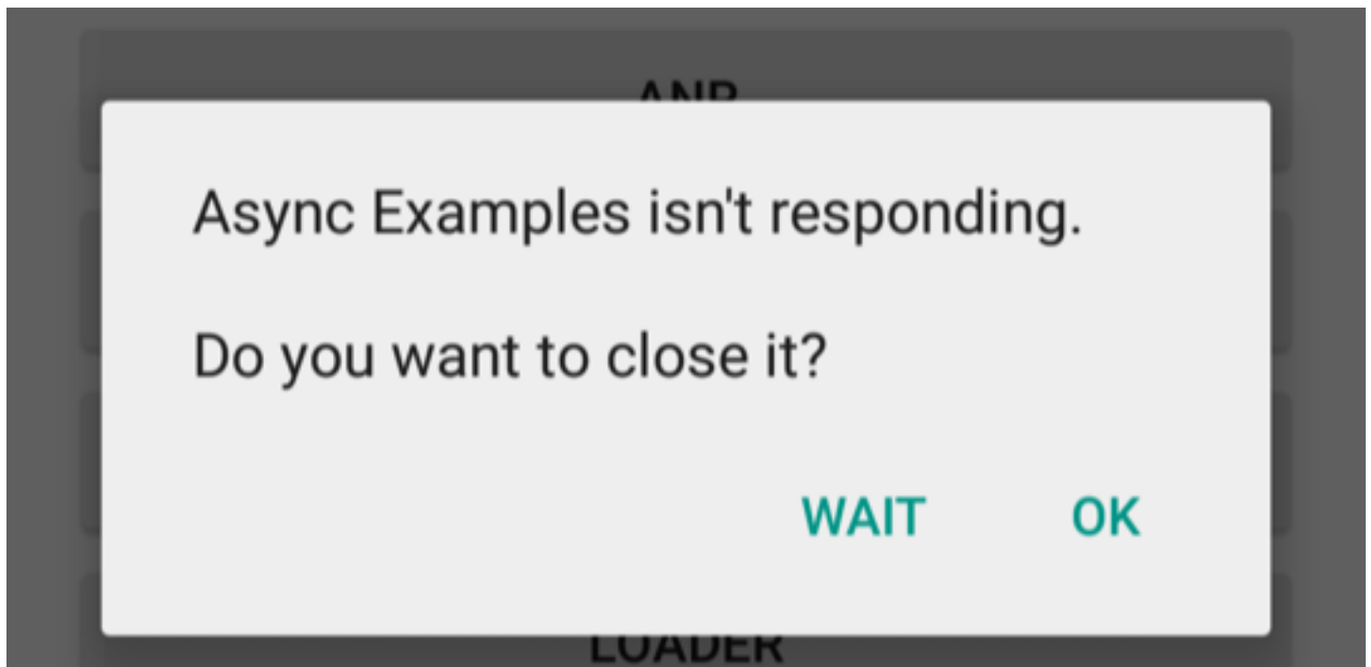
The University Of Sheffield.

Keeping Your App Responsive

https://developer.android.com/training/articles/perf-anr.html

Sluggish Response and ANRs

- It's possible to write code that wins every performance test in the world
  - but still feels sluggish, hangs or freezes for significant periods, or takes too long to process input.
- The worst thing that can happen to your app's responsiveness is an **"Application Not Responding" (ANR)** dialog



The University Of Sheffield.

- It's critical to design **responsiveness** into your application
  - so the system never displays an ANR dialog to the user
- Android displays an ANR if an application cannot respond to user input
  - e.g. if an application blocks on some I/O operation (e.g. a network access) on the UI thread
    - so the system can't process incoming user input events
  - e.g. the app spends time building an elaborate in-memory structure on the UI thread
- These computations must be efficient, but even the most efficient code still takes time to run!!

- You should NEVER perform a potentially **lengthy operation** on the UI thread
  - Create a worker thread and do most of the work there.
  - This keeps the UI thread (which drives the user interface event loop) running and prevents the system from concluding that your code has frozen
- Android will display the ANR dialog if:
  - No response to an input event (such as key press or screen touch events) **within 5 seconds**
  - A BroadcastReceiver hasn't finished executing **within 10 seconds**

Do not overuse the UI Thread

- Android applications normally run entirely on a **single thread**
  - by default the "UI thread" or "main thread"
- Anything your application is doing in the UI thread that takes a long time can trigger the ANR dialog
- Any method running in the UI thread should do **as little work as possible**
  - Activities should do **as little as possible** to set up in key life-cycle methods such as **onCreate() and onResume()**

Async processing

- You should always use a worker thread for potentially **long running operations**
  - such as network or database operations,
  - computationally expensive calculations
    - e.g. resizing bitmaps
- To create a worker thread for longer operations use the **AsyncTask class**
  - Extend AsyncTask and implement the **doInBackground()** method to perform the work
  - To post progress to the user interface, call **publishProgress()**, which invokes the **onProgressUpdate()** callback method
- From your implementation of onProgressUpdate() (which runs on the UI thread), you can notify the user

Running on UI Thread

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    // Do the long-running work in here
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    // This is called each time you call publishProgress()
    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    // This is called when doInBackground() is finished
    protected void onPostExecute(Long result) {
        showNotification("Downloaded " + result + " bytes");
    }
}
```

Runs on background Thread (cannot write on UI)

Run on UI Thread (can write on UI)

The University Of Sheffield.

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    // Do the long-running work in here
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    // This is called each time you call publishProgress()
    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    // This is called when doInBackground() is finished
    protected void onPostExecute(Long result) {
        showNotification("Downloaded " + result + " bytes");
    }
}
```

The University Of Sheffield.

Running the Async process

To execute this worker thread, simply create an instance and call `execute()`:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Updating the interface

- What is a thread ?
  - A thread defines a process running
- What is UIThread
  - Main thread of execution for your application
  - Most of your application code will run here onCreate, onPause, onDestroy, onClick, etc.
  - So simply Anything that causes the UI to be updated or changed HAS to happen on the UI thread
- When you explicitly spawn a new thread or an async task to do work in the background
  - the code is NOT run on the UIThread, so YOU cannot modify the UI

https://stackoverflow.com/questions/41632607/difference-between-android-runonuithread-and-simple-code-in-java

The University Of Sheffield.

Running on UI Thread

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    // Do the long-running work in here
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    // This is called each time you call publishProgress()
    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    // This is called when doInBackground() is finished
    protected void onPostExecute(Long result) {
        showNotification("Downloaded " + result + " bytes");
    }
}
```

Runs on background Thread (cannot write on UI)

Run on UI Thread (can write on UI)

Accessing UI from Thread

- What if you must access the UI Thread from a background task?
  - You normally **should not**!
  - Typically it means you are doing something wrong
- You must use runOnUiThread() when you want to update your UI from a Non-UI Thread

```
runOnUiThread(new Runnable() {
                    @Override
                    public void run() {

                        //do something
                    });
```

SUMMARY

- Part 1:
  - Persisting Data
  - Internal/External Storage
- Part 2:
  - ROOM
  - Keeping Your App Responsive
- Lab tutorial :
  - Using ROOM to retrieve data