

- For Database connectivity Practicals, Link 7 concepts (from your guidelines) have been explained here.
- Youtube videos Links have also been specified:

❖ Link 1: basics of sqlite

<https://www.youtube.com/watch?v=Uv0JRm7OdsW>

❖ Link 2: working with sqlite

<https://www.youtube.com/watch?v=sreuidkR4yc>

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.

Define a schema and contract

One of the main principles of SQL databases is the schema: a formal declaration of how the database is organized. The schema is reflected in the SQL statements that you use to create your database. You may find it helpful to create a companion class, known as a *contract* class, which explicitly specifies the layout of your schema in a systematic and 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.

For example, the following contract defines the table name and column names for a single table representing an RSS feed:

JAVA code

```
public final class FeedReaderContract {
    // To prevent someone from accidentally instantiating the contract
    class,
    // make the constructor private.
    private FeedReaderContract() {}

    /* Inner class that defines the table contents */
    public static class FeedEntry implements BaseColumns {
        public static final String TABLE_NAME = "entry";
        public static final String COLUMN_NAME_TITLE = "title";
        public static final String COLUMN_NAME_SUBTITLE = "subtitle";
    }
}
```

Create a database using an SQL helper

Once you have defined how your database looks, you should implement methods that create and maintain the database and tables. Here are some typical statements that create and delete a table:

JAVA

```
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedEntry.TABLE_NAME + " (" +
    FeedEntry._ID + " INTEGER PRIMARY KEY," +
    FeedEntry.COLUMN_NAME_TITLE + " TEXT," +
    FeedEntry.COLUMN_NAME_SUBTITLE + " TEXT)";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + FeedEntry.TABLE_NAME;
```

Just like files that you save on the device's internal storage, Android stores your database in your app's private folder. Your data is secure, because by default this area is not accessible to other apps or the user.

The SQLiteOpenHelper class contains a useful set of APIs for managing your database. When you use this class to obtain references to your database, the

system performs the potentially long-running operations of creating and updating the database only when needed and *not during app startup*. All you need to do is call `getWritableDatabase()` or `getReadableDatabase()`.

To use `SQLiteOpenHelper`, create a subclass that overrides the `onCreate()` and `onUpgrade()` callback methods. You may also want to implement the `onDowngrade()` or `onOpen()` methods, but they are not required.

For example, here's an implementation of `SQLiteOpenHelper` that uses some of the commands shown above:

JAVA

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the
    // database version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }

    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        // This database is only a cache for online data, so its
        // upgrade policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }

    public void onDowngrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

To access your database, instantiate your subclass of `SQLiteOpenHelper`:

JAVA

```
FeedReaderDbHelper dbHelper = new FeedReaderDbHelper(getContext());
```

Put information into a database

Insert data into the database by passing a `ContentValues` object to the `insert()` method:

JAVA

```
// Gets the data repository in write mode
SQLiteDatabase db = dbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedEntry.COLUMN_NAME_SUBTITLE, subtitle);

// Insert the new row, returning the primary key value of the new row
long newRowId = db.insert(FeedEntry.TABLE_NAME, null, values);
```

The first argument for `insert()` is simply the table name.

The second argument tells the framework what to do in the event that the `ContentValues` is empty (i.e., you did not `put` any values). If you specify the name of a column, the framework inserts a row and sets the value of that column to null. If you specify `null`, like in this code sample, the framework does not insert a row when there are no values.

The `insert()` methods returns the ID for the newly created row, or it will return -1 if there was an error inserting the data. This can happen if you have a conflict with pre-existing data in the database.

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.

JAVA

```
SQLiteDatabase db = dbHelper.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 your selection statements immune to SQL injection. For more detail about all arguments, see the [query\(\)](#) reference.

To look at a row in the cursor, use one of the [Cursor](#) move methods, which you must always call before you begin reading values. Since the cursor starts at position -1, calling [moveToNext\(\)](#) places the "read position" on the first entry in the results and returns whether or not the cursor is already past the last entry in the result set. For each row, you can read a column's value by calling one of the [Cursor](#) get methods, such as [getString\(\)](#) or [getLong\(\)](#). For each of the get methods, you must pass the index position of the column you desire, which you can get by calling [getColumnIndex\(\)](#) or [getColumnIndexOrThrow\(\)](#). When finished iterating through results, call [close\(\)](#) on the cursor to release its resources. For example, the following shows how to get all the item IDs stored in a cursor and add them to a list:

JAVA

```

List itemIds = new ArrayList<>();
while(cursor.moveToNext()) {
    long itemId = cursor.getLong(
        cursor.getColumnIndexOrThrow(FeedEntry._ID));
    itemIds.add(itemId);
}
cursor.close();

```

Delete information from a database

To delete rows from a table, you need to provide selection criteria that identify the rows to the `delete()` method. The mechanism works the same as the selection arguments to the `query()` method. It divides the selection specification into a selection clause and selection arguments. The clause defines the columns to look at, and also allows you to combine column tests. The arguments are values to test against that are bound into the clause. Because the result isn't handled the same as a regular SQL statement, it is immune to SQL injection.

JAVA

```
// Define 'where' part of query.
String selection = FeedEntry.COLUMN_NAME_TITLE + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { "MyTitle" };
// Issue SQL statement.
int deletedRows = db.delete(FeedEntry.TABLE_NAME, selection,
selectionArgs);
```

The return value for the `delete()` method indicates the number of rows that were deleted from the database.

Update a database

When you need to modify a subset of your database values, use the `update()` method.

Updating the table combines the `ContentValues` syntax of `insert()` with the `WHERE` syntax of `delete()`.

JAVA

```
SQLiteDatabase db = dbHelper.getWritableDatabase();

// New value for one column
String title = "MyNewTitle";
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the title
String selection = FeedEntry.COLUMN_NAME_TITLE + " LIKE ?";
String[] selectionArgs = { "MyOldTitle" };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
```

```
values,  
selection,  
selectionArgs);
```

The return value of the `update()` method is the number of rows affected in the database.

Persisting database connection

Since `getWritableDatabase()` and `getReadableDatabase()` are expensive to call when the database is closed, you should leave your database connection open for as long as you possibly need to access it. Typically, it is optimal to close the database in the `onDestroy()` of the calling Activity

JAVA code

```
@Override  
protected void onDestroy() {  
    dbHelper.close();  
    super.onDestroy();  
}
```

For any doubt contact 9873961590