COMP-304 Fall 2018

Review of Lecture 5

- Data-Driven Containers :
  - **CursorAdapter**
    - when using data from a database
  - **ArrayAdapter**
    - binds each element of the array to a single View object within the layout resource:

```
ArrayAdapter<String> adapter =
    new
    ArrayAdapter<String>(this,
    android.R.layout.simple_drop
    down_item_1line, anArray);
textView.setAdapter(adapter);
```

- **ListView control**
  - used for **full-screen menus** or lists of items from which a user selects
  - use **ListActivity as the base class** for such screens
  - To handle selection event, implement the **onListItemClick()** method within your ListActivity
- **Drawing on the Screen**
  - Create a subclass of the **View** class and write the drawing code in **onDraw()** method
  - Use **Canvas** methods to draw
  - Use **Paint** object to set painting features(*color*, *style*,etc)

Review of Lecture 5

- **Paint Gradients**:
  - LinearGradient
  - RadialGradient
  - SweepGradient
  - set the Paint gradient using the setShader() method.
- **Working with Text:**
  - Create a **Typeface** object
  - Set *text size* and *typeface* using **Paint** methods
  - Draw text using Canvas method **drawText**.
- **Drawing Bitmap Graphics on a Canvas**
  - draw bitmaps within the **onDraw()** method of a View, using one of the **drawBitmap()** methods
- **Transforming Bitmaps Using Matrixes**
  - **Matrix** class to perform tasks such as **mirroring, scaling** and **rotating** graphics
- **Using Shapes**
  - Draw primitive shapes such as rectangles and ovals using the **ShapeDrawable** class in conjunction with a variety of specialized **Shape** classes
    - Define shape in **xml**
    - Draw it on **ImageView** using **setImageResource** method.
    - define **ShapeDrawable** instances programmatically
    - Draw shape on Imageview using **setImageDrawable** method

Review of Lecture 5

- **Drawing on ImageView**
  - Create a **BitMap** as content view for the image
  - Construct a **canvas** with the specified bitmap to draw into
  - Create a **Paint** object
  - Use **Canvas methods** (**drawLine**, etc.) to draw on the image
- **Using Dialogs**
  - **Use DialogFragment**
    - In **onCreateDialog**, use an **AlertDialog** builder to create a simple **AlertDialog** with Yes/No confirmation buttons
- **Showing Time Passage with the Chronometer**
  - **start, stop,** and **setBase** methods
- **Displaying the time with DigitalClock** and **AnalogClock** controls
  - DigitalColock displays the time in a TextView
  - AnalogClock control is a dial-based clock with a basic clock face with two hands, one for the minute and one for the hour.
    - It updates automatically as each minute passes

Using Android Data and Storage APIs

Objectives:

- Create animations in Android apps
  - Frame-by-Frame animation
  - Tweened Animation
- Use Application **Preferences**
- Create and use **SQLite databases**
- Write Android apps that manipulate an **SQLite** database

Working with Animation

- The Android platform supports three types of graphics animation:
  - Animated GIF images
  - Frame-by-frame animation
  - Tweened animation
- **Animated GIFs** store the animation frames within the image, and you simply include these GIFs like any other graphic **drawable resource**.
- For **frame-by-frame animation**, the developer must provide all graphics frames of the animation.
- However, with **tweened animation, only a single graphic is needed**, upon which **transforms** can be programmatically applied.

Working with Frame-by-Frame Animation

- You can think of frame-by-frame animation as a digital flipbook in which **a series of similar images display on the screen in a sequence**, each subtly different from the last.
- When you display these images quickly, they give the **illusion of movement**.
- This technique is called frame-by-frame animation and is often used on the Web in the form of **animated GIF images**.
- Frame-by-frame animation is best used for **complicated graphics transformations** that are not easily implemented programmatically.

genie juggling example

Working with Tweened Animations

- With tweened animation, you can provide a single Drawable resource—it is a **Bitmap** graphic , a **ShapeDrawable**, a *TextView*, or **any other type of View** object - and the intermediate frames of the animation are rendered by the system.
- Android provides tweening support for several common image **transformations**, including **alpha**, **rotate**, **scale**, and **translate** animations.
- You can apply **tweened animation transformations** to any View, whether it is an ImageView with a Bitmap or ShapeDrawable, or a layout such as a TableLayout.

Defining Tweening Transformations

- You can define tweening transformations as **XML resource** files or **programmatically**.
- All tweened animations share some common properties, including when to **start**, **how long** to animate, and whether to return to the starting state upon completion.

Defining Tweened Animations as XML Resources

- We can store animation sequences as specially formatted XML files within the **/res/anim/** resource directory.
- For example, the following resource file called /res/anim/spin.xml describes a **simple five-second rotation**:

```
<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android
= "http://schemas.android.com/apk/res/android"
android:shareInterpolator="false">
        <rotate
        android:fromDegrees="0"
        android:toDegrees="360"
        android:pivotX="50%"
        android:pivotY="50%"
        android:duration="5000" />
</set>
```

Defining Tweened Animations Programmatically

- You can also programmatically define these animations.
- The different types of transformations are available as classes within the android.view.**animation** package.
- For example, you can define the aforementioned rotation animation as follows:

```
RotateAnimation rotate = new RotateAnimation(
0, 360, RotateAnimation.RELATIVE_TO_SELF, 0.5f,
RotateAnimation.RELATIVE_TO_SELF, 0.5f);
rotate.setDuration(5000);
```

Defining Simultaneous and Sequential Tweened Animations

- Animation transformations can happen simultaneously or sequentially when you set the startOffset and duration properties, which control when and for how long an animation takes to complete.
- You can **combine animations into the <set> tag** (or, programmatically using **AnimationSet**) to share properties.

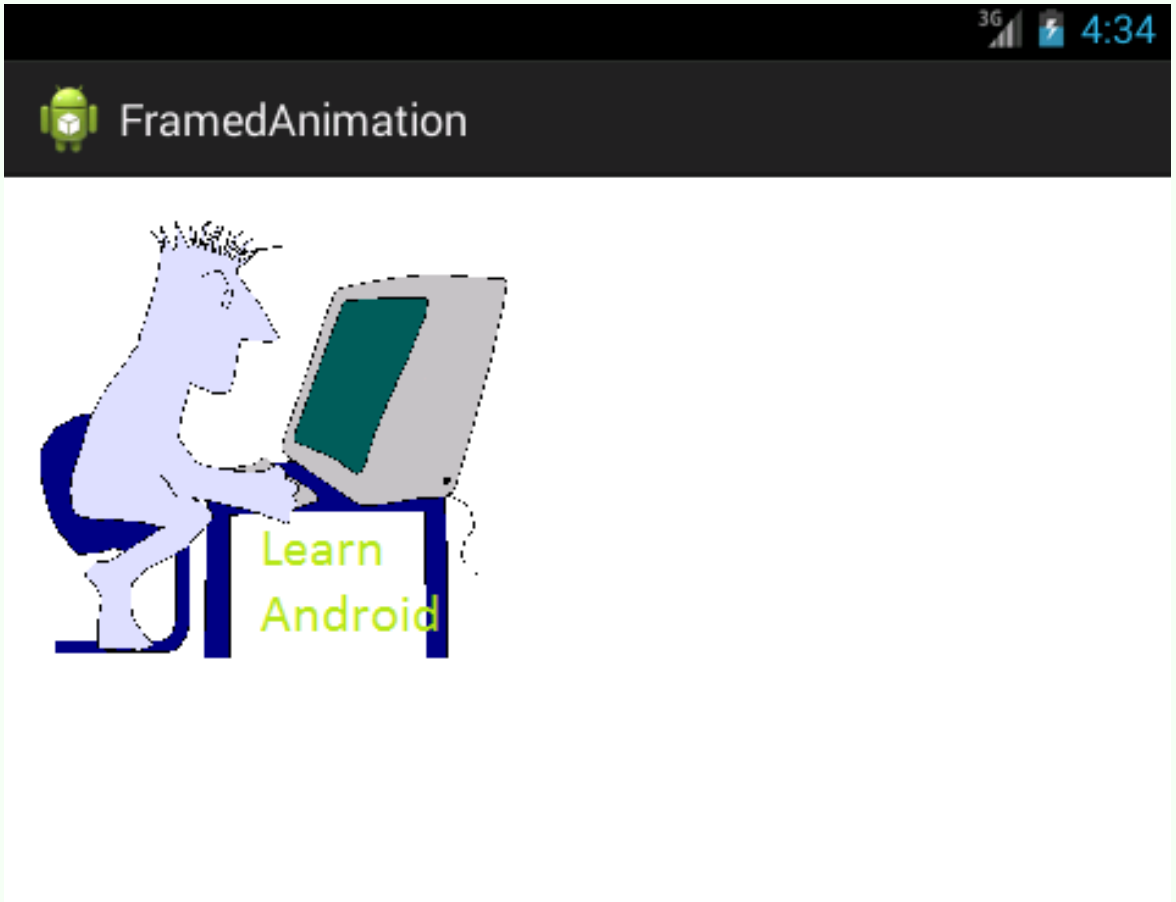
```
<set>
    <alpha
    android:fromAlpha="0.0"
    android:toAlpha="1.0"
    ……/>
    <scale
    android:duration="2500"
    android:pivotX="50%"
    …….. />
    <rotate
    android:fromDegrees="0"
    android:toDegrees="360“
    …….. />
</set>
```

Loading Animations

- Loading animations is made simple by using the **AnimationUtils** helper class.
- The following code loads an animation XML resource file called /res/anim/grow.xml and applies it to an ImageView whose source resource is a green rectangle shape drawable:

```
ImageView iView =
    (ImageView)findViewById(R.id.ImageView1);
iView.setImageResource(R.drawable.green_rect);
Animation an =
AnimationUtils.loadAnimation(this, R.anim.grow);
iView.startAnimation(an);
```

Framed Animation example



SimpleFramedAnimation

Listening for Animation events

- We can listen for Animation events, including the animation start, end, and repeat events, by implementing an AnimationListener class, such as the MyListener class shown here:

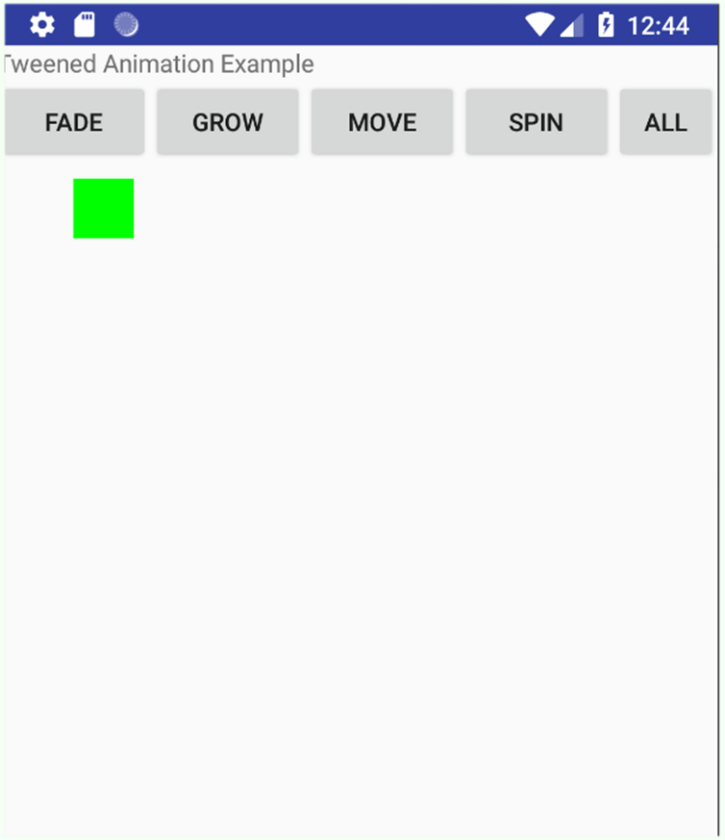
```
class MyListener implements Animation.AnimationListener {
        public void onAnimationEnd(Animation animation) {
        // Do at end of animation
        }
        public void onAnimationRepeat(Animation animation) {
        // Do each time the animation loops
        }
        public void onAnimationStart(Animation animation) {
        // Do at start of animation
        }
}
```

- You can then register your AnimationListener as follows:

```
an.setAnimationListener(new MyListener());
```

Four Different Tweening Transformations

- These types are:
  - Transparency changes (Alpha)
  - Rotations (Rotate)
  - Scaling (Scale)
  - Movement (Translate)



- AnimationsExample:

Storage Options

- **Shared Preferences**
  - Store private primitive data in **key-value** pairs.
- **Internal Storage**
  - Store private data on the device memory **– files**, **cache files**.
- **External Storage**
  - Store public data on the **shared external storage** – SD card, etc.
- **SQLite Databases**
  - Store structured data in a **private database**.
- **Network Connection**
  - Store data on the web with your own **network server**.

Working with Application Preferences

- Many applications use a **lightweight data storage** mechanism called **shared preferences** for storing:
  - **application state**
  - simple **user information**
  - **configuration options**
  - other similar information

Working with Application Preferences

- Android provides a **simple preferences** system for **storing primitive application data at the Activity level** and **preferences shared across** all of an application's activities.
- You **cannot share** preferences **outside of the package**.
- Preferences are stored as groups of **key/value** pairs.
- The following **data types** are supported as preference settings:
  - **Boolean** values
  - **Float** values
  - **Integer** values
  - **Long** values
  - **String** values

Working with Application Preferences

- To add preferences support to your application, you must take the following steps:
  1. Retrieve an instance of a **SharedPreferences** object.
  2. Create a **SharedPreferences.Editor** to modify preference content.
  3. Make changes to the preferences using the **Editor**.
  4. **Commit** your changes.

Creating Private Preferences

- Individual activities can have their own **private preferences**.
- These preferences are for the **specific Activity only** and are **not shared with other activities** within the application.
- The activity gets **only one group** of private preferences.
- The following code retrieves the activity's private preferences:

```
SharedPreferences settingsActivity =
    getPreferences(MODE_PRIVATE);
```

Creating Shared preferences

- Creating **shared preferences** is similar.
- The only two differences are that we **must name our preference set** and use a different method to get the preference instance:

SharedPreferences settings =

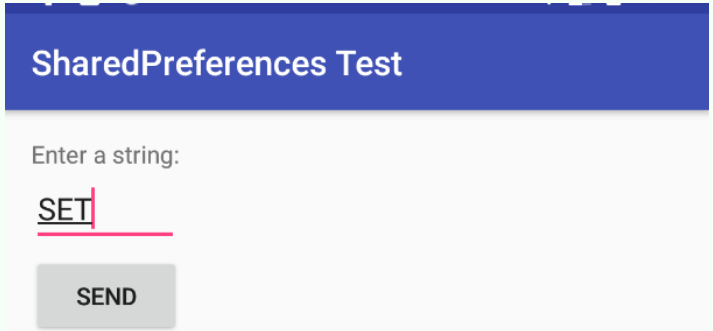
getSharedPreferences(“MyCustomSharedPreferences”, 0);

- You can access shared preferences **by name** from any activity in the application.
- There is **no limit to the number** of different shared preferences you can create.

Adding, Updating, and Deleting Preferences

- The following code retrieves the activity's private preferences, opens the preference editor, adds a long preference called *SomeLong*, and saves the change:

```
SharedPreferences settingsActivity =
    getPreferences(MODE_PRIVATE);
SharedPreferences.Editor prefEditor = settingsActivity.edit();
prefEditor.putLong("SomeLong", java.lang.Long.MIN_VALUE);
prefEditor.commit();
```



Finding Preferences Data on the Android File System

- Internally, **application preferences are stored as XML files in** on the Android file system in the following directory: **/data/data/**<package name>**/shared_prefs/**<preferences filename>**.xml**
- You can access the preferences file by selecting View -> Tool Windows -> Device File Explorer
- Example:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<map>
<string name="String_Pref">Test String</string>
<int name="Int_Pref" value="-2147483648" />
<float name="Float_Pref" value="-Infinity" />
<long name="Long_Pref" value="92233720368547758O7" />
<boolean name="Boolean_Pref" value="false" />
</map>
```

Defining Preferences in XML

- Although you can instantiate new Preference objects at runtime, you should **define your list of settings in XML** with a hierarchy of **Preference** objects.
- Using an XML file to define your collection of settings is preferred because the file provides an easy-to-read structure that's **simple to update**.
- Also, your app's settings are generally pre-determined, although you can still modify the collection at runtime.
- Each **Preference subclass** can be declared with an XML element that matches the class name, such as <CheckBoxPreference>.

Defining Preferences in XML

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory android:title="Category 1">
        <CheckBoxPreference
            android:title="Checkbox"
            android:defaultValue="false"
            android:summary="True or False"
            android:key="checkboxPref" />
        </PreferenceCategory>
    <PreferenceCategory android:title="Category 2">
        <EditTextPreference
            android:summary="Enter a string"
            android:defaultValue="[Enter a string here]"
            android:title="Edit Text"
            android:key="editTextPref"
            />
</PreferenceCategory>
</PreferenceScreen>
```

Defining Preferences in XML

- If you want to provide dividers with headings between groups of settings, place each group of Preference objects inside a **PreferenceCategory**:

```
<PreferenceCategory android:title="Category 1">
    <CheckBoxPreference
        android:title="Checkbox"
        android:defaultValue="false"
        android:summary="True or False"
        android:key="checkboxPref" />
</PreferenceCategory>
```

Loading and Reading Preferences

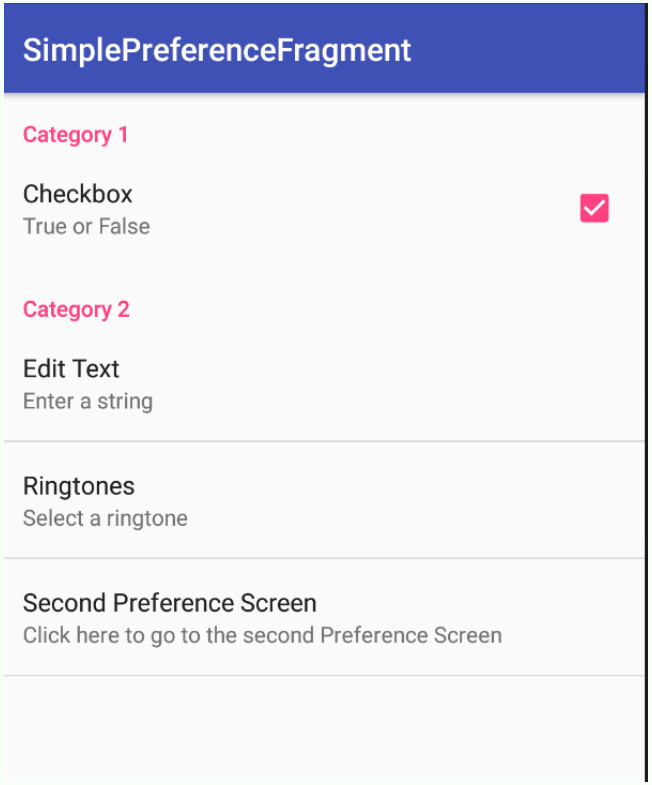
- To load the preferences from an XML file:
  addPreferencesFromResource(R.xml.*myappprefer ences);*
- To read the preferences from an XML file:

SharedPreferences appPrefs =
getSharedPreferences("appPreferences", *MODE_PRIVATE);*

DisplayText(String.*valueOf(appPrefs.**getBoolean**("checkboxPref", false**)));***

PreferenceFragment example

- For Android 3.0 (API level 11) and higher versions, **PreferenceFragment** is used to display the list of Preference objects.
- You can add a **PreferenceFragment** to any activity, no need to use *PreferenceActivity*.



Storing Data Using SQLite Databases

- **SQLite** databases are lightweight and file-based, making them ideally suited for embedded devices
- The Android SDK includes a number of useful **SQLite database management classes**.
- Many of these classes are found in the android.database.**sqlite** package.
- You can also interact directly with application's database using the **sqlite3 command-line tool** that's accessible through the ADB shell interface

Creating a SQLite Database Instance

- The simplest way to create a new SQLiteDatabase instance for your application is to use the openOrCreateDatabase() method of your **application Context**, like this:

```
SQLiteDatabase mDatabase;
mDatabase = openOrCreateDatabase(
"my_sqlite_database.db",
SQLiteDatabase.CREATE_IF_NECESSARY,
null);
```

- **SQLiteDatabase** class contains methods to manage a database (create, delete, execute SQL commands)

Application's Database File on the Device

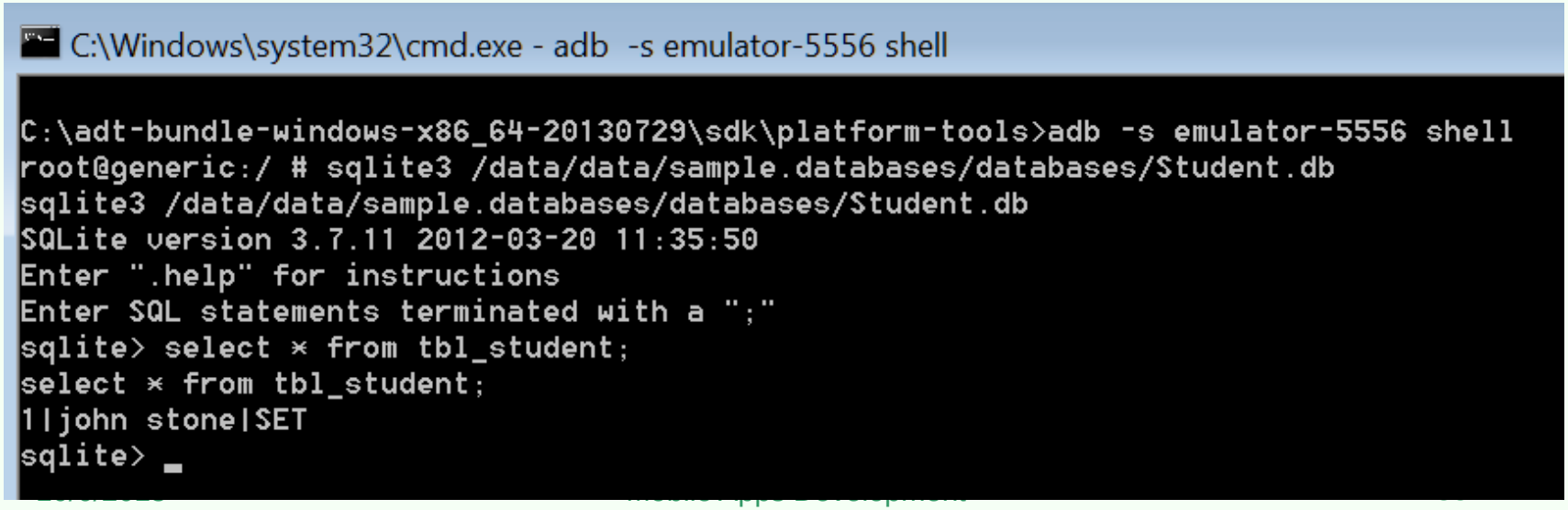
- Android applications store their databases (SQLite or otherwise) in a special application directory:

/data/data/<application package name>/**databases**/<databasename>

- For example, the path to the database would be

/data/data/sample.databases/databases/Student.db

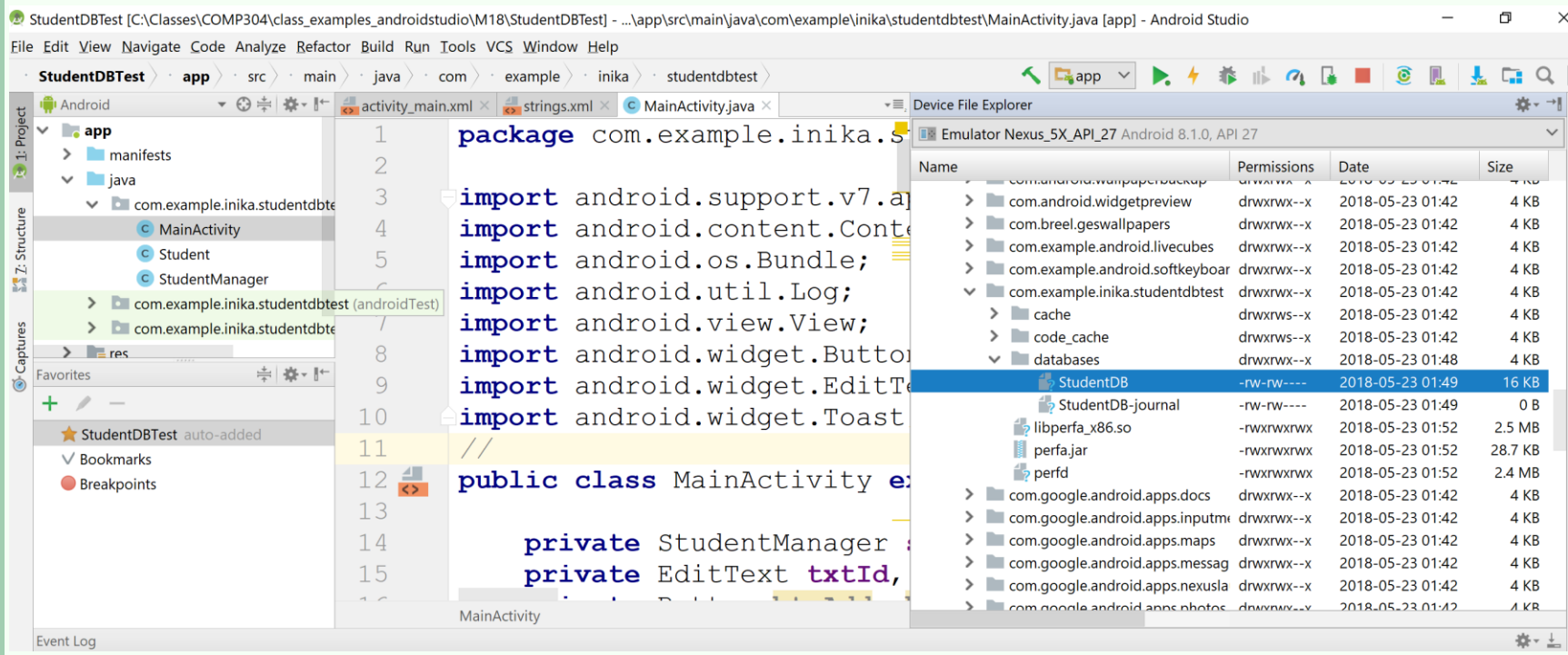
- You can access your database using the **sqlite3** command-line interface using this path:



Application's Database File on the Device

- You can access the database file by selecting:

  View -> Tool Windows -> Device File Explorer ->data/data/<application package name>/**databases**/<databasename>:



Configuring the SQLite Database Properties

- Some important database **configuration options** include **version**, **locale**, and the **thread-safe locking** feature.

```
Database.setLocale(Locale.getDefault());
Database.setLockingEnabled(true); //deprecated since API level 16
Database.setVersion(1);
```

Creating Tables

- The following is a valid CREATE TABLE SQL statement.
  - This statement creates a table called tbl_authors, with three fields: a unique **id** number, which auto-increments with each record and acts as our primary key, and **firstname** and **lastname** text fields:

```
CREATE TABLE tbl_authors (
id INTEGER PRIMARY KEY AUTOINCREMENT,
firstname TEXT, lastname TEXT);
```

Creating Tables

- You can encapsulate this CREATE TABLE SQL statement in a static final String variable (called CREATE_AUTHOR_TABLE) and then execute it on your database using the **execSQL()** method:

mDatabase.**execSQL**(CREATE_AUTHOR_TABLE);

  - This method executes a single SQL statement that is NOT a SELECT or any other SQL statement that returns data.

Other SQLite Features

- SQLite does **not enforce foreign key constraints**.
- Instead, we **must enforce them ourselves using custom SQL triggers**.

Inserting Records

- We use the *insert()* method to add new data to our tables.
- We use the *ContentValues* object **to pair the column names to the column values** for the record we want to insert.
- For example, here we insert a record into tbl_authors for J.K. Rowling:

```
ContentValues values = new ContentValues();
values.put("firstname", "J.K.");
values.put("lastname", "Rowling");
long newAuthorID = mDatabase.insert("tbl_authors", null,
    values);
```

- The **insert()** method returns the *id* of the newly created record.
- We can use this author *id* to create book records for this author

Updating Records

- You can modify records in the database using the update() method.
- The update() method takes four arguments:
  1. The **table** to update records
  2. A ContentValues object with the modified fields to update
  3. An **optional WHERE clause**, in which **?** identifies a WHERE clause argument
  4. An **array** of WHERE clause arguments, each of which is substituted in place of the ?'s from the second parameter

Updating Records

- Passing null to the WHERE clause **modifies all records within the table**, which can be useful for making sweeping changes to your database.
- Most of the time, we want to **modify individual records by their unique identifier**.
- In the following function we find the record in the table called tbl_books that corresponds with the id and update that book's title.
- ContentValues object is used to bind our column names to our data values:

```
public void updateBookTitle(Integer bookId, String newtitle) {
ContentValues values = new ContentValues();
values.put("title", newtitle);
mDatabase.update("tbl_books", values, "id=?", new String[] {
    bookId.toString() });
}
```

Deleting Records

- You can remove records from the database using the remove() method.
- The remove() method takes three arguments:
  1. The **table** to delete the record from
  2. An **optional WHERE clause**, in which ? identifies a WHERE clause argument
  3. An **array** of WHERE clause arguments, each of which is substituted in place of the ?'s from the second parameter
- Passing null to the WHERE clause deletes all records within the table.
- For example, this function call deletes all records within the table called tbl_authors:

```
mDatabase.delete("tbl_authors", null, null);
```

Working with Transactions

- You can use **SQL Transactions** to group operations together; if any of the operations fails, you can handle the error and either recover or roll back all operations.
- If the operations all succeed, you can then commit them. Here we have the basic structure for a transaction:

```
mDatabase.beginTransaction();
try {
        // Insert some records, updated others, delete a few
        // Do whatever you need to do as a unit, then commit it
    mDatabase.setTransactionSuccessful();
} catch (Exception e) {
        // Transaction failed. Failed! Do something here.
        // It's up to you.
} finally {
    mDatabase.endTransaction();
}
```

Working with Cursors

- Use a **Cursor** to access the results returned from a SQL query
  - Cursor class is in android.**database** package
  - Think of query results as a table, in which each row corresponds to a returned record.
- A Cursor object is similar to a file pointer; it allows random access to query results.
- The **Cursor** object includes helpful **methods** for determining **how many results** were returned by the query the Cursor represents and methods for **determining the column names** (fields) for each returned record.

Working with Cursors

- The columns in the query results are defined by the query, not necessarily by the database columns.
  - These might include calculated columns, column aliases, and composite columns.

```
// SIMPLE QUERY: select * from tbl_books
Cursor c =
    mDatabase.query("tbl_books",null,null,null,null,null
    ,null);
// Do something quick with the Cursor here...
c.close();
```

- public Cursor **query** (String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)

Managing Cursors as Part of the Application Lifecycle

- Cursor objects **must be managed** as part of the application lifecycle.
  - When the application pauses or shuts down, the **Cursor** must be deactivated with a call to the **deactivate()** method
  - When the application restarts, the **Cursor** should refresh its data using the **requery()** method.
  - When the **Cursor** is no longer needed, a call to **close()** must be made to release its resources.
- Handle this by implementing **Cursor** management calls within the various lifecycle callbacks, such as **onPause()**, **onResume()**, and **onDestroy()**.

Managing Cursors as Part of the Application Lifecycle

- You can hand off the responsibility of managing Cursor objects to the parent Activity by using the Activity method called startManagingCursor().
- The Activity handles the rest, deactivating and reactivating the Cursor as necessary and destroying the Cursor when the Activity is destroyed.
- You can always begin manually managing the Cursor object again later by simply calling stopManagingCursor().

Managing Cursors as Part of the Application Lifecycle

- Here we perform the same simple query and then hand over **Cursor** management to the parent Activity:

```
// SIMPLE QUERY: select * from tbl_books
Cursor c =
    mDatabase.query("tbl_books",null,null,null,null,null
    ,null);
startManagingCursor(c);
```

Navigating Query Results

- You can **use the Cursor to iterate query results**, one row at a time using various navigation methods such as moveToFirst(), moveToNext(), and isAfterLast().
- On a specific row, you can **use the Cursor to extract the data for a given column** in the query results.
- Because **SQLite is not strongly typed**, you can always pull fields out as Strings using the getString() method, but you can also use the type-appropriate extraction utility function to **enforce type safety** in your application.

Navigating Query Results

- The following method takes a valid Cursor object, prints the number of returned results, some column information (name and number of columns):

```
public void logCursorInfo(Cursor c) {
Log.i(DEBUG_TAG, "*** Cursor Begin *** " + " Results:" +
c.getCount() + " Columns: " + c.getColumnCount());
// Print column names
String rowHeaders = "|| ";
for (int i = 0; i < c.getColumnCount(); i++) {
rowHeaders = rowHeaders.concat(c.getColumnName(i) + " ||
    ");
}
```

Designing Persistent Databases

- The Android SDK provides a helper class called **SQLiteOpenHelper** to help you manage your application's database.
- To create a SQLite database for your Android application using the SQLiteOpenHelper, you need to **extend that class** and then **instantiate an instance** of it as a member variable for use within your application:

```
DatabaseManager extends SQLiteOpenHelper {
    //...
}
```

- Override the *constructor*, *onCreate*, and *onUpgrade* methods to handle the **creation** of a new database and **upgrade** to a new version, respectively.

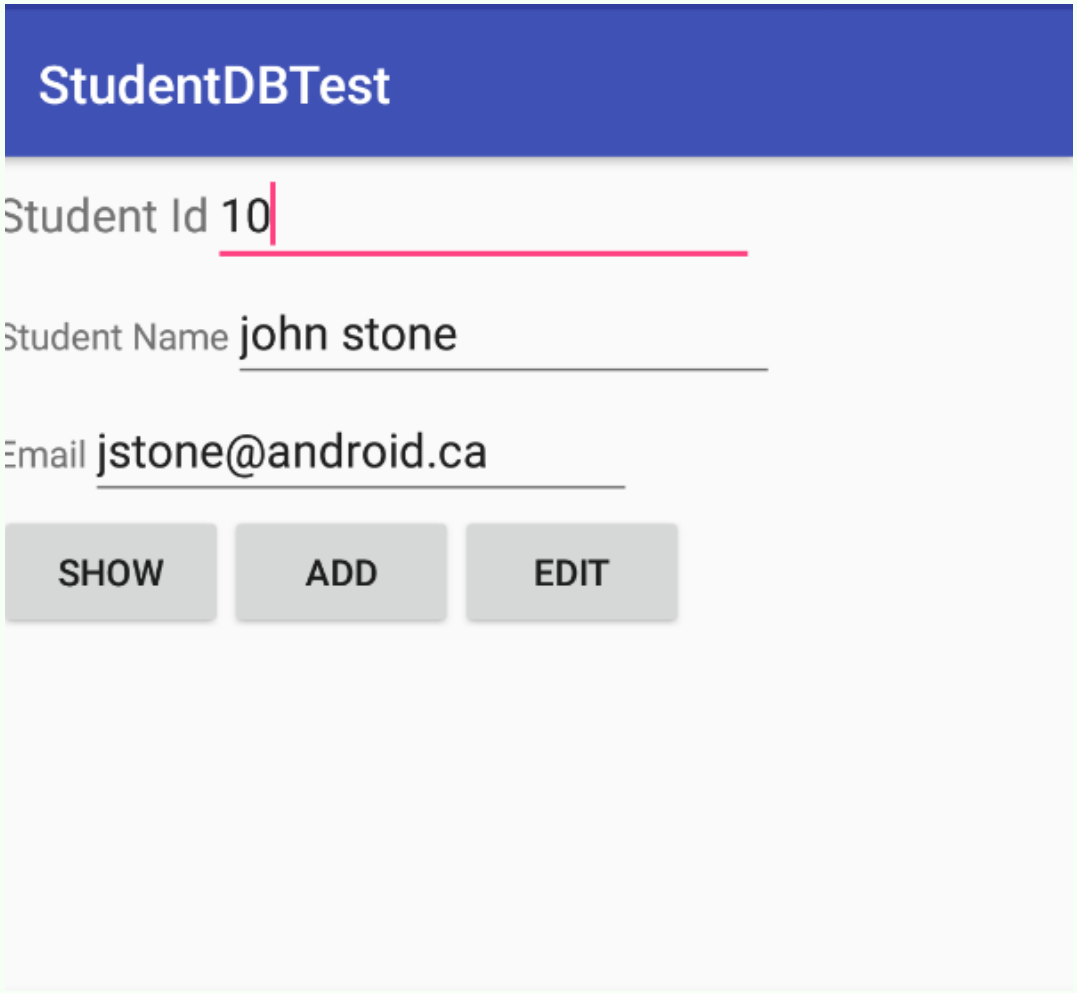
Designing Persistent Databases

- **onCreate** - called when the database is created for the first time.
  - This is where the **creation of tables and the initial population of the tables** should happen
- **onUpgrade** - called when the database needs to be upgraded.
  - The implementation should use this method to **drop tables, add tables, or do anything else it needs to upgrade to the new schema version**.
- **SQLiteOpenHelper** versions the database files - the version number is the int argument passed to the constructor.
  - **A higher version number** passed in the constructor **triggers onUpgrade execution**.

StudentDBTest example

- Insert, Edit, and Display student records:

- Student class
  describes a
  student object
- StudentManager
  class
  Defines database
  operations
- MainActivity class
  UI, creator strings,
  Event handling



References

- Textbook
- Android Documentation: http://developer.android.com/guide/topics/data/data-storage.html, http://developer.android.com/guide/topics/ui/settings.html,
- https://developer.android.com/training/data-storage/sqlite#java