SQLite Databases
SQLite Database

- Android applications can have application databases powered by SQLite
  - Lightweight and file-based, ideal for mobile devices
  - Databases are private for the application that creates them
  - Databases should not be used to store files
- SQLite is a lightweight database
  - Atomic
  - Stable
  - Independent
  - Enduring
  - Only several kilobytes
  - Only partly support some SQL commands such as ALTER, TABLE.
- SQLite is included as part of Android’s software stack
- More info about SQLite at http://www.sqlite.org
SQLite Databases

- **Steps for using SQLite databases:**
  1. Create a database
  2. Open the database
  3. Create a table
  4. Create and insert interface for datasets
  5. Create a query interface for datasets
  6. Close the database

- **Good practice to create a Database Adapter class to simplify your database interactions**

- **We will use the SQLite database defined in the notebook tutorial as an example**
public class NotesDbAdapter {

    public static final String KEY_TITLE = "title";
    public static final String KEY_BODY = "body";
    public static final String KEY_ROWID = "_id";

    private static final String TAG = "NotesDbAdapter";
    private DatabaseHelper mDbHelper;
    private SQLiteDatabase mDb;

    /**
     * Database creation sql statement
     */
    private static final String DATABASE_CREATE =
    "create table notes ("_id" integer primary key autoincrement, "
    + "title" text not null, body text not null);";

    private static final String DATABASE_NAME = "data";
    private static final String DATABASE_TABLE = "notes";
    private static final int DATABASE_VERSION = 2;

    private final Context mCtx;
SQLiteOpenHelper Class

- Abstract class for implementing a best practice pattern for creating, opening and upgrading databases
- To create a SQLite database, the recommended approach is to create a subclass of SQLiteOpenHelper class
- Then override its onCreate() method
  - Then execute a SQLite command to create tables in the database
- Use the onUpgrade() method to handle upgrade of the database
  - A simple way would be to drop an existing table and replace with a new definition
  - Better to migrate existing data into a new table
- Then use an instance of the helper class to manage opening or upgrading the database
  - If the database doesn’t exist, the helper will create one by calling its onCreate() handler
  - If the database version has changed, it will upgrade by calling the onUpgrade() handler
private static class DatabaseHelper extends SQLiteOpenHelper {

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

    @Override
    public void onCreate(SQLiteDatabase db) {

        db.execSQL(DATABASE_CREATE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        Log.w(TAG, "Upgrading database from version " + oldVersion + " to " + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS notes");
        onCreate(db);
    }
}
public NotesDbAdapter(Context ctx) {
    this.mCtx = ctx;
}

class NotesDbAdapter {
    public NotesDbAdapter open() throws SQLException {
        mDbHelper = new DatabaseHelper(mCtx);
        mDb = mDbHelper.getWritableDatabase();
        return this;
    }

    public void close() {
        mDbHelper.close();
    }
}
SQLite Databases

- ContentValues() objects used to hold rows to be inserted into the database
- Example:

```java
public long createNote(String title, String body) {
    ContentValues initialValues = new ContentValues();
    initialValues.put(KEY_TITLE, title);
    initialValues.put(KEY_BODY, body);

    return mDb.insert(DATABASE_TABLE, null, initialValues);
}

public boolean deleteNote(long rowId) {
    return mDb.delete(DATABASE_TABLE, KEY_ROWID + "=" + rowId, null) > 0;
}

public boolean updateNote(long rowId, String title, String body) {
    ContentValues args = new ContentValues();
    args.put(KEY_TITLE, title);
    args.put(KEY_BODY, body);

    return mDb.update(DATABASE_TABLE, args, KEY_ROWID + "=" + rowId, null) > 0;
}
```
SQLite Databases

- Database queries are returned as Cursor objects
  - Pointers to the resulting sets within the underlying data
- Cursor class provides several methods:
  - `moveToFirst`, `moveToNext`, `moveToPrevious`, `moveToPosition` used to move to a row
  - `getCount` to get the number of rows in the cursor
  - `getPosition` to get the current row position
  - `getColumnName`, `getColumnNames`, `getColumnIndexOrNull` to get info on columns
  - `startManagingCursor` and `stopManagingCursor` methods used to integrate cursor lifetime into the activity’s lifetime
public Cursor fetchAllNotes() {

    return mDb.query(DATABASE_TABLE, new String[] {KEY_ROWID, KEY_TITLE, KEY_BODY}, null, null, null, null, null);
}

public Cursor fetchNote(long rowId) throws SQLException {

    Cursor mCursor =

    mDb.query(true, DATABASE_TABLE, new String[] {KEY_ROWID, KEY_TITLE, KEY_BODY}, KEY_ROWID + "=" + rowId, null, null, null, null);

    if (mCursor != null) {
        mCursor.moveToFirst();
    }

    return mCursor;
}
SQLite Example: Notebook Tutorial

• Within the main activity, cursors returned by the Dbadapter are used as follows:

```java
private void fillData() {
    Cursor notesCursor = mDbHelper.fetchAllNotes();
    startManagingCursor(notesCursor);

    // Create an array to specify the fields we want to display in the list (only TITLE)
    String[] from = new String[]{NotesDbAdapter.KEY_TITLE};

    // and an array of the fields we want to bind those fields to (in this case just text1)
    int[] to = new int[]{R.id.text1};

    // Now create a simple cursor adapter and set it to display
    SimpleCursorAdapter notes =
        new SimpleCursorAdapter(this, R.layout.notes_row, notesCursor, from, to);
    setListAdapter(notes);
}
```
Content Providers
Content Providers

• Store and retrieve data and make it available to all applications
  – Only way to share data across applications
• Standard content providers part of Android:
  – Common data types (audio, video, images, personal contact information)
• Applications can create their own content providers to make their data public
  – Alternatively add the data to an existing provider
• Implement a common interface for querying the provider, adding, altering and deleting data
• Actual storage of data is up to the designer
• Provides a clean separation between the application layer and data layer
Accessing Content

• Applications access the content through a ContentResolver instance
  – ContentResolver allows querying, inserting, deleting and updating data from the content provider
    
    ContentResolver cr = getContentResolver();

    cr.query(People.CONTENT_URI, null, null, null, null); //querying contacts

    ContentValues newvalues = new ContentValues();
    cr.insert(People.CONTENT_URI, newvalues);

    cr.delete(People.CONTENT_URI, null, null); //delete all contacts
Content Providers

• Content providers expose their data as a simple table on a database model
  – Each row is a record and each column is data of a particular type and meaning

• Queries return cursor objects

• Each content provider exposes a public URI that uniquely identifies its data set
  – Separate URI for each data set under the control of the provider
  – URIs start with content://...
  – Typical format:
    Content://<package name>.provider.<custom provider name>/<DataPath>
Content Providers: Query

- You need three pieces of information to query a content provider:
  - The URI that identifies the provider
  - The names of the data fields you want to receive
  - The data types for those fields
- If you're querying a particular record, you also need the ID for that record
- Example:
  ```java
  import android.provider.Contacts.People;
  import android.content.ContentUris;
  import android.net.Uri;
  import android.database.Cursor;

  // Use the ContentUris method to produce the base URI for the contact with _ID == 23.
  Uri myPerson = ContentUris.withAppendedId(People.CONTENT_URI, 23);

  // Alternatively, use the Uri method to produce the base URI.
  // It takes a string rather than an integer.
  Uri myPerson = Uri.withAppendedPath(People.CONTENT_URI, "23");

  // Then query for this specific record:
  Cursor cur = managedQuery(myPerson, null, null, null, null);
  ```
import android.provider.Contacts.People;
import android.database.Cursor;

// Form an array specifying which columns to return.
String[] projection = new String[] {
    People._ID,
    People._COUNT,
    People.NAME,
    People.NUMBER
};

// Get the base URI for the People table in the Contacts content provider.
Uri contacts = People.CONTENT_URI;

// Make the query.
Cursor managedCursor = managedQuery(contacts,
    projection, // Which columns to return
    null,      // Which rows to return (all rows)
    null,      // Selection arguments (none)
    null,      // Put the results in ascending order by name
    People.NAME + " ASC");
Content Providers: Query

- Retrieving the data:
  import android.provider.Contacts.People;

  private void getColumnData(Cursor cur){
      if (cur.moveToFirst()) {
          String name;
          String phoneNumber;
          int nameColumn = cur.getColumnIndex(People.NAME);
          int phoneColumn = cur.getColumnIndex(People.NUMBER);
          String imagePath;

          do {
              // Get the field values
              name = cur.getString(nameColumn);
              phoneNumber = cur.getString(phoneColumn);

              // Do something with the values.
              ...

          } while (cur.moveToNext());
      }
  }
Content Providers: Modifying Data

• Data kept by a content provider can be modified by:
  – Adding new records
  – Adding new values to existing records
  – Batch updating existing records
  – Deleting records

• All accomplished using ContentResolver methods

• Use ContentValues() to add or update data
Content Providers: Adding Data

• Adding new records:
  import android.provider.Contacts.People;
  import android.content.ContentResolver;
  import android.content.ContentValues;

  ContentValues values = new ContentValues();

  // Add Abraham Lincoln to contacts and make him a favorite.
  values.put(People.NAME, "Abraham Lincoln");
  // 1 = the new contact is added to favorites
  // 0 = the new contact is not added to favorites
  values.put(People.STARRED, 1);

  Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
Content Providers: Adding Data

• Adding new values:

```java
Uri phoneUri = null;
Uri emailUri = null;

phoneUri = Uri.withAppendedPath(uri, People.Phones.CONTENT_DIRECTORY);

values.clear();
values.put(People.Phones.TYPE, People.Phones.TYPE_MOBILE);
values.put(People.Phones.NUMBER, "1233214567");
getContentResolver().insert(phoneUri, values);

// Now add an email address in the same way.
emailUri = Uri.withAppendedPath(uri, People.ContactMethods.CONTENT_DIRECTORY);

values.clear();
// ContactMethods.KIND is used to distinguish different kinds of
// contact methods, such as email, IM, etc.
values.put(People.ContactMethods.KIND, Contacts.KIND_EMAIL);
values.put(People.ContactMethods.DATA, "test@example.com");
values.put(People.ContactMethods.TYPE, People.ContactMethods.TYPE_HOME);
getContentResolver().insert(emailUri, values);
```
Content Providers

• Use ContentResolver.update() to batch update fields
• Use ContentResolver.delete() to delete:
  – A specific row
  – Multiple rows, by calling the method with the URI of the type of record to delete and an SQL WHERE clause defining which rows to delete
Creating Content Providers

• To create a content provider, you must:
  – Set up a system for storing the data. Most content providers store their data using Android's file storage methods or SQLite databases, but you can store your data any way you want.
  – Extend the ContentProvider class to provide access to your data
  – Declare the content provider in the manifest file for your application (AndroidManifest.xml)
Creating Content Providers

• Extending the ContentProvider class will require:
  – Implementing the following methods: query(), insert(), update(), delete(), getType(), onCreate()
  – Make sure that these implementations are thread-safe as they may be called from several ContentResolver objects in several different processes and threads

• In addition, you need to:
  – Declare a public static final URI named CONTENT_URI
  – Define the column names that the content provider will return to clients
  – Carefully document the data type of each column
Creating Content Providers

• You need to declare the content provider in the Manifest file:
• Example:
  
  <provider android:name="com.example.autos.AutoInfoProvider"
    android:authorities="com.example.autos.autoinfoprovider"
    ...
  </provider>