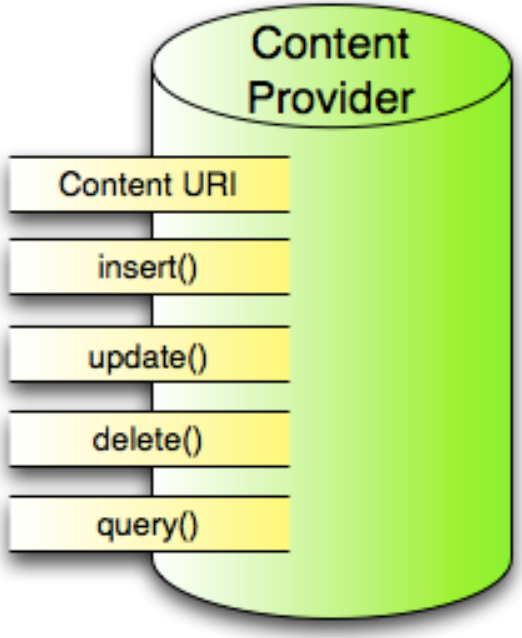
ContentProviders

Programming the Android Platform

CS 282
Principles of Operating Systems II
Systems Programming for Android

ContentProvider

- ContentProviders manage access to a central repository of structured data & can make an app's data available to other applications
- They encapsulate the data & provide mechanisms for defining data security
- Content providers are the standard interface that connects data in one process with code running in another process

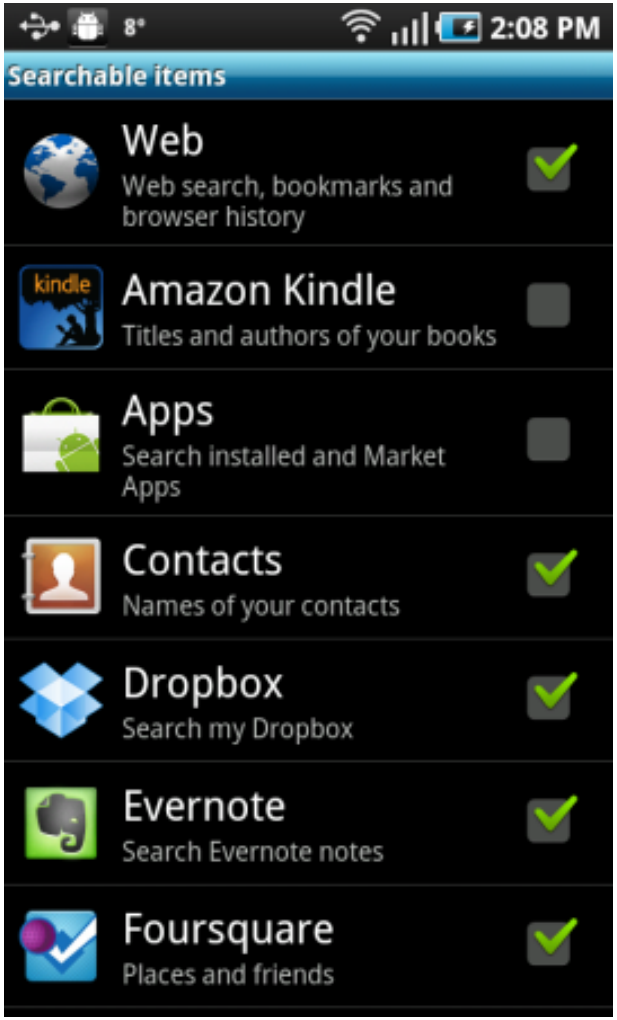


- Content providers support database "CRUD" operations (*create*, *read*, *update*, *delete*), where "*read*" is implemented as "*query*"
- Apps can provide Activities that allow users to query & modify the data managed by a provider

http://developer.android.com/guide/topics/providers/content-providers.html

Example Android ContentProviders

- Android itself includes many ContentProviders that manage data for
  - Browser – bookmarks, history
  - Call log- telephone usage
  - Contacts – contact data
  - Media – media database
  - UserDictionary – database for predictive spelling
  - Maps – previous searches
  - YouTube – previous searches
  - Many more



http://developer.android.com/reference/android/provider/package-summary.html

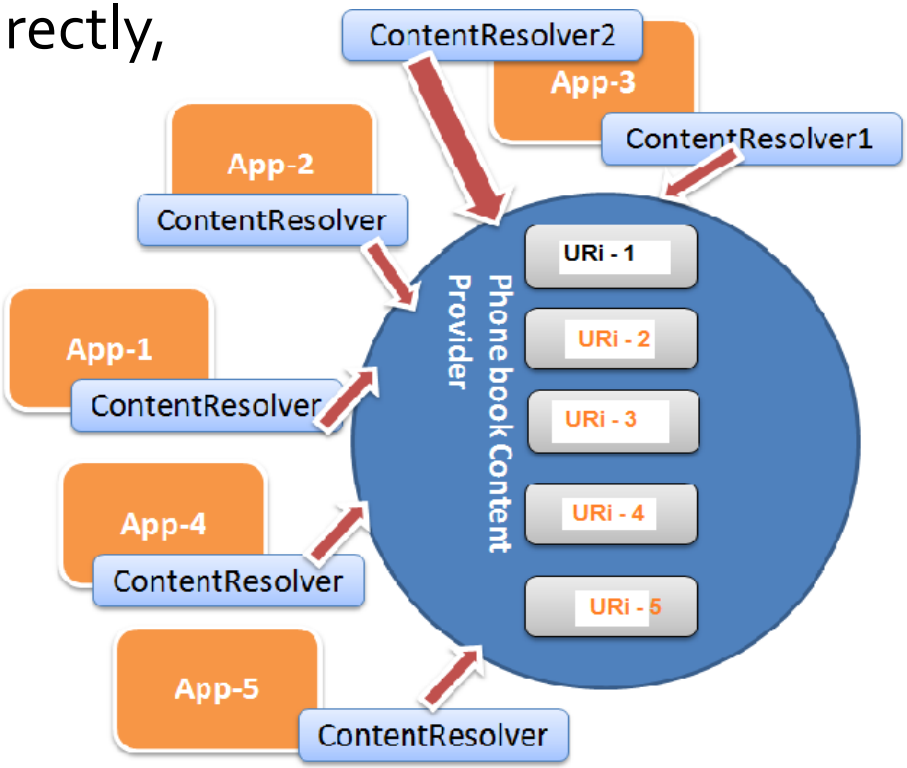
ContentProvider Data Model

- A content provider typically presents data to external applications as one or more tables
  - e.g., the tables found in a relational SQL database
- A row represents an instance of some type of data the provider collects
  - Each column in a row represents an individual piece of data collected for an instance
- e.g., one of the built-in providers in Android is the user dictionary, which stores the spellings of non-standard words that the user wants to keep

word	app id	freq	locale	_ID
mapreduce	user1	100	en_US	1
precompiler	user14	200	fr_FR	2
applet	user2	225	fr_CA	3
const	user1	255	pt_BR	4
int	user5	100	en_UK	5

ContentResolver

- ContentProvider never accessed directly, but accessed indirectly via a ContentResolver
  - ContentProvider not created until a ContentResolver tries to access it
- ContentResolvers manage & support ContentProviders
  - Enables use of ContentProviders across multiple applications
  - Provides additional services, such as change notification & IPC



- Context.getContentResolver() accesses default ContentResolver

```
ContentResolver cr = getContentResolver();
```

http://developer.android.com/reference/android/content/ContentResolver.html

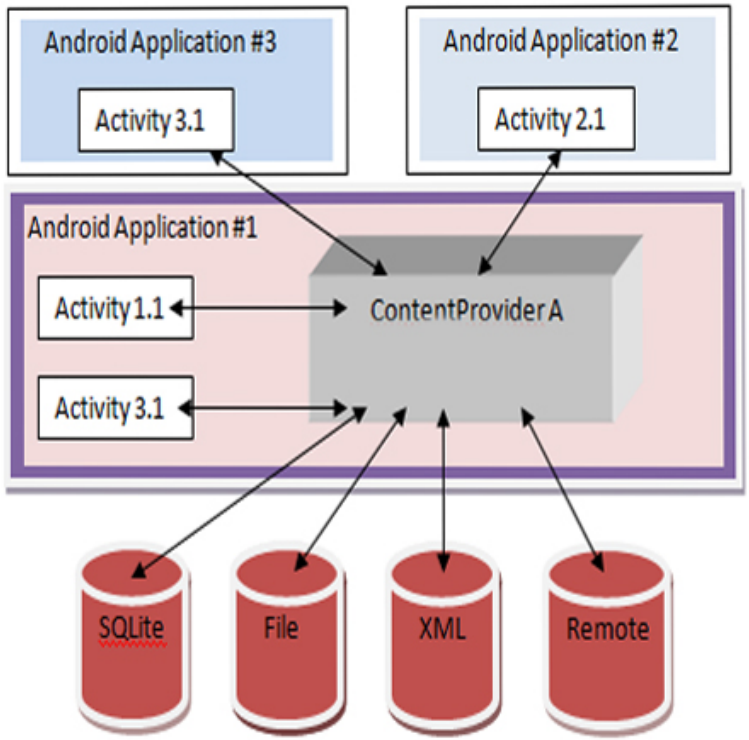
ContentResolver & ContentProvider

- When you query data via a ContentProvider, you don't communicate with the provider directly
  - Instead, you use a ContentResolver object to communicate with the provider
- The sequence of events for a query is :
  1. A call to getContentResolver().query(Uri, String, String, String, String) is made
     - This call invokes the Content Resolver's query method, *not* the ContentProvider's
  2. When the query method is invoked, the Content Resolver parses the uri argument & extracts its authority
  3. The Content Resolver directs the request to the content provider registered with the (unique) authority by calling the Content Provider's query() method
  4. When the Content Provider's query() method is invoked, the query is performed & a Cursor is returned (or an exception is thrown)
     - The resulting behavior depends on the Content Provider's implementation

http://www.androiddesignpatterns.com/search?updated-max=2012-08-21T18:14:00-04:00&max-results=4

Content URIs

- Any URI that begins with the *content://* scheme represents a resource served up by a Content Provider
- e.g., *content://authority/path/id*
  - *content* - data is managed by a ContentProvider
  - *authority* – id for the content provider
  - *path* – 0 or more segments indicating the type of data to access
  - *id* – specific record being requested



- ContentProviders are a façade that offers data encapsulation via Content Uri objects used as handles
  - The data could be stored in a SQLite database, in flat files, retrieved off a device, be stored on some server accessed over the Internet, etc.

Inserting Data

- Use ContentResolver.insert() to insert data into a ContentProvider

public final Uri insert(Uri url, ContentValues values)

- Inserts a row into a table at the given URL
- If the content provider supports transactions the insertion will be atomic

Parameters

- *url* – The URL of the table to insert into.
- *values* – The initial values for the newly inserted row. The key is the column name for the field. Passing an empty ContentValues will create an empty row.

Returns

- the URL of the newly created row.

http://developer.android.com/reference/android/content/ContentProvider.html#insert(android.net.Uri, android.content.ContentValues)

Deleting Data

- Use ContentResolver.delete() to delete data from a ContentProvider

public final int delete(Uri url, String where, String[] selectionArgs)

- Deletes row(s) specified by a content URI. If the content provider supports transactions, the deletion will be atomic.

Parameters

- *url* – The URL of the row to delete
- *where* – A filter to apply to rows before deleting, formatted as an SQL WHERE clause (excluding the WHERE itself).
- *selectionArgs* – SQL pattern args

Returns

- The number of rows deleted.

http://developer.android.com/reference/android/content/ContentProvider.html#delete(android.net.Uri, java.lang.String, java.lang.String[])

Inserting/Deleting via applyBatch()

- ContentResolver.applyBatch() can be used to insert (&delete) groups of data

public ContentProviderResult[] applyBatch (String authority,
ArrayList<ContentProviderOperation> operations)

- Applies each ContentProviderOperation object & returns array of results
- If all the applications succeed then a ContentProviderResult array with the same number of elements as the operations will be returned

Parameters

- *authority* –authority of the ContentProvider to apply this batch
- *operations* – the operations to apply

Returns

- the results of the applications

http://developer.android.com/reference/android/content/ContentProvider.html#applyBatch(java.util.ArrayList<android.content.ContentProviderOperation>)

Querying a ContentResolver

- Use ContentResolver.query() to retrieve data
  - Returns a Cursor instance for accessing results
  - A Cursor is an iterator over a result set

```
Cursor query(
        Uri uri,                    // ContentProvider Uri
        String[] projection         // Columns to retrieve
        String selection            // SQL selection pattern
        String[]                    // SQL pattern args
              selectionArgs
        String sortOrder            // Sort order
        )
```

```
Cursor c= qb.query(db.getReadableDatabase(), projection,
                        selection, selectionArgs, orderBy);
```

http://developer.android.com/reference/android/content/ContentProvider.html

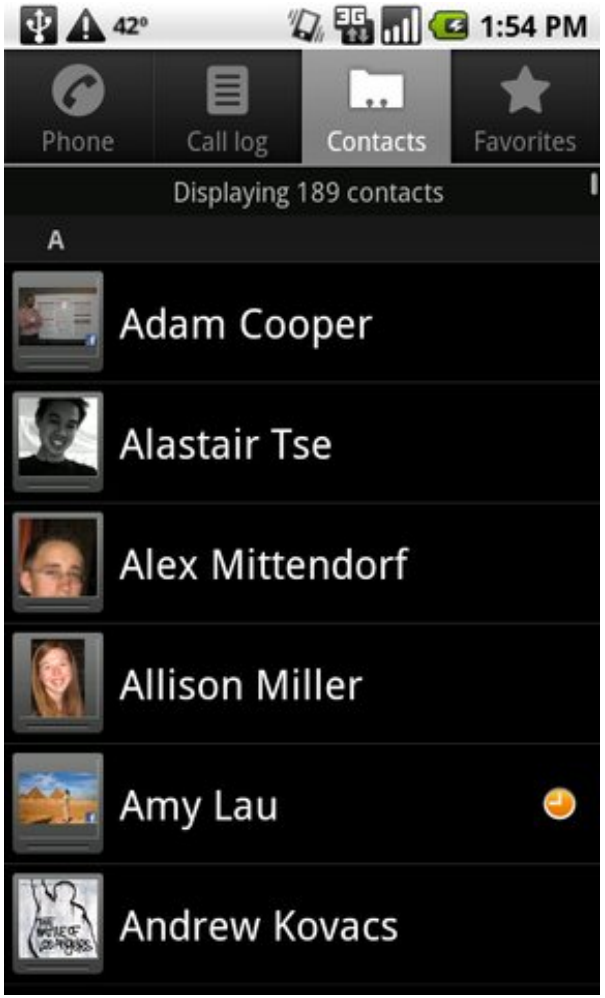
query() Compared to SQL Query

query() argument	SELECT keyword/parameter	Notes
Uri	FROM *table_name*	Uri maps to the table in the provider named *table_name*
projection	*col,col,col,…*	projection is an array of columns that should be included for each row retrieved
selection	WHERE *col =value*	selection specifies the criteria for selecting rows
selectionArgs	(No exact equivalent. Selection arguments replace ?placeholders in the selection clause.)	
sortOrder	ORDER BY*col,col,…*	sortOrder specifies the order in which rows appear in the returned Cursor

Example: Contacts

- Uri for searching Android contacts database

  static final Uri
  ContactsContract.Contacts.CONTENT_URI =
  "content://com.android.contacts/contacts/"



Contacts Example

```
public class ContactsListExample extends ListActivity {
  public void onCreate(Bundle savedInstanceState) {
    ...
    ContentResolver cr = getContentResolver();
    Cursor c = cr.query(ContactsContract.Contacts.CONTENT_URI,
          new String[] { ContactsContract.Contacts.DISPLAY_NAME },
          null,   null,   null);
    List<String> contacts = new ArrayList<String>();
    if (c.moveToFirst()) {
     do {
       // add data to contacts variable
     } while (c.moveToNext());
    }
    ... // populate list view widget
```

Using Cursor to Add Data to Contacts

- Provides an iterator for accessing query results
- Some useful methods
  - boolean moveToFirst()
  - boolean moveToNext()
  - int getColumnIndex (String columnName)
  - String getString (int columnIndex)

```
public class ContactsListExample extends
    ListActivity {
  public void onCreate(Bundle
    savedInstanceState) {
  Cursor c = // issue query
  List<String> contacts = new
    ArrayList<String>();
  if (c.moveToFirst()) {
    do {
      contacts.add(c.getString
            (c.getColumnIndex(
              ContactsContract.Contacts.
                        DISPLAY_NAME)));
    } while (c.moveToNext());
  ... // populate list view widget
```

http://developer.android.com/reference/android/database/Cursor.html

Populating the ListView Widget

```
...
    // populate list view widget
    ArrayAdapter<String> adapter =
      new ArrayAdapter<String>(this, R.layout.list_item, contacts);
    setListAdapter(adapter);
  }
}
```

- The ArrayAdapter class can handle any Java object as input & maps the data of this input to a TextView in the layout
- ArrayAdapter uses the toString() method of the data input object to determine the String which should be displayed

http://developer.android.com/reference/android/widget/ArrayAdapter.html

A More Sophisticated Query

```
public class ContactsListExample extends ListActivity {
  public void onCreate(Bundle savedInstanceState) {
    ...
    // columns to retrieve
    String columns[] = new String[] {
                    ContactsContract.Contacts._ID,
                    ContactsContract.Contacts.DISPLAY_NAME,
                    ContactsContract.Contacts.STARRED };
    // columns to display
    String disp [] = new String[] {
                          ContactsContract.Contacts.DISPLAY_NAME};
   // layout for columns to display
    int[] colResIds = new int[] { R.id.name };
    ...
```

A More Sophisticated Query (cont.)

```
    …
    ContentResolver cr = getContentResolver();
    Cursor c = cr.query(
        ContactsContract.Contacts.CONTENT_URI, columns,
        ContactsContract.Contacts.STARRED + "= 0”, null, null);

        setListAdapter(new SimpleCursorAdapter(
           this, R.layout.list_layout,  c,colsToDisplay, colResIds));
    }
}
```

Deleting Data Example

```
public class ContactsListDisplayActivity extends ListActivity {
  …
  private void deleteContact(String name) {
    getContentResolver().delete(
     ContactsContract.RawContacts.CONTENT_URI,
     ContactsContract.Contacts.DISPLAY_NAME + "=?",
     new String[] {name});
  }
  private void deleteAllContacts() {
    getContentResolver().delete(
            ContactsContract.RawContacts.CONTENT_URI, null, null);
  }
  …
}
```

Inserting Data Example

```
public class ContactsListDisplayActivity extends ListActivity {
  ...
  private void insertContact(String name) {
    ArrayList<ContentProviderOperation> ops =
                        new  ArrayList<ContentProviderOperation>();
   ...
  // create new RawContacts
   ...
    try {
       getContentResolver()
            .applyBatch(ContactsContract.AUTHORITY, ops);
    } catch (RemoteException e) {
    } catch (OperationApplicationException e) {
    }
  ...
}
```

Inserting Data Example (cont.)

```
// create new RawContact
  ops.add(ContentProviderOperation
    .newInsert(RawContacts.CONTENT_URI)
    .withValue(RawContacts.ACCOUNT_TYPE, "com.google")
    .withValue(RawContacts.ACCOUNT_NAME,
            "douglas.craig.schmidt@gmail.com")

    .build());
// add new RawContact
  ops.add(ContentProviderOperation.newInsert(Data.CONTENT_URI)
   .withValueBackReference(Data.RAW_CONTACT_ID,0)
   .withValue(Data.MIMETYPE,StructuredName.CONTENT_ITEM_TYPE)
   .withValue(StructuredName.DISPLAY_NAME, name)
   .build());
```

Creating a ContentProvider

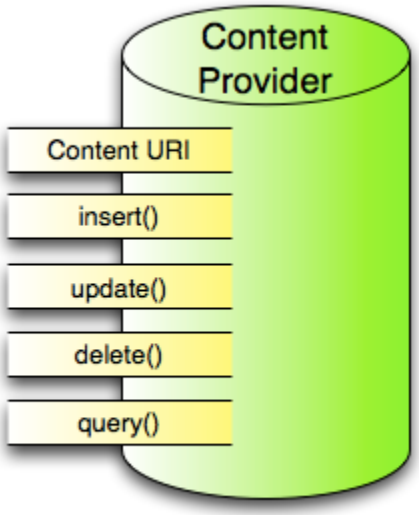
- Before making the effort to create a ContentProvider make sure you really need one!
- Some considerations include:
  - You want to offer complex data or files to other applications
  - You want to allow users to copy complex data from your app into other apps
  - You want to provide custom search suggestions using the search framework

- Three steps to creating a ContentProvider
  1. Implement a storage system for the data
     - e.g., file vs. structure data etc.
  2. Implement a provider as one or more classes in an Android application, along with <provider> element in manifest file
  3. One class implements a subclass of ContentProvider, which is the interface between your provider & other applications

http://developer.android.com/guide/topics/providers/content-provider-creating.html

Required ContentProvider Methods

- ContentProvider defines six abstract methods that must be implemented in subclass:
  - query() uses arguments to select table to query, rows & columns to return via Cursor, & sort order of result
  - insert() uses arguments to select the destination table & to get the column values to use to insert a new row
  - update() uses arguments to select the table & rows to update & to get the updated column values
  - delete() uses arguments to select the table & rows to delete
  - getType() returns MIME type corresponding to a content URI
  - onCreate() initializes a provider (called immediately after creating a provider)

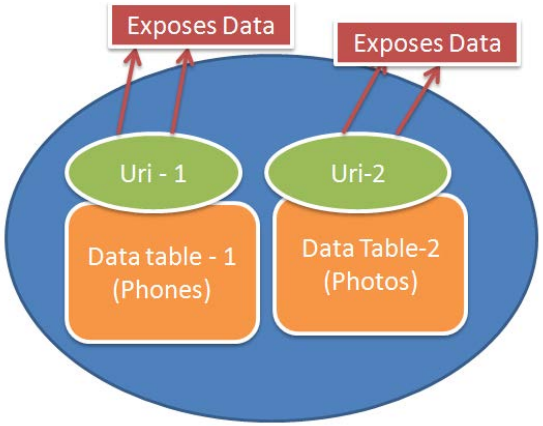


- Methods have same signature as identically named ContentResolver methods

http://developer.android.com/guide/topics/providers/content-provider-creating.html

Define Content URI(s)

- A Content URI is a URI that identifies data in a provider
  - Each ContentProvider method uses a content URI as an argument to determine which table, row, or file to access



- Content URIs include
  - The symbolic name of the entire provider (its authority)
  - A name that points to a table or file (a path)
  - An optional id part points to an individual row in a table
- Define a unique public static final Uri named CONTENT_URI, representing the full content:// URI your content provider handles
  - Use fully-qualified class name of content provider (made lowercase), e.g.:

    public static final Uri CONTENT_URI =
    Uri.parse("content://course.examples.contentproviders.mycontentprovider");
- If provider has subtables, define CONTENT_URI consts for each one

Define Column Names

- Define column names
  - Typically identical to the SQL database column names
- Also define public static String constants that clients can use to specify the columns
- Be sure to include an integer column named "_id" (with the constant _ID) for the IDs of the records
  - If you use an SQLite database, the _ID field should be of type INTEGER PRIMARY KEY AUTOINCREMENT
- Document the data type of each column so clients can read the data

```
public static final String
   _ID = “_id”, DATA= "data";

private static final String[] columns =
     new String[] { _ID, DATA};

private static final String
contentTypeSingle
    = "vnd.android.cursor.item/
        myContentProvider.data.text";

private static final String
contentTypeMultiple =
"vnd.android.cursor.dir/
    myContentProvider.data.text";
```

Define MIME Types for Tables

- The getType() method returns a String in MIME format that describes the type of data returned by the content URI argument
- For content URIs that point to a row or rows of table data, getType() should return a MIME type in Android's vendor-specific MIME format described at http://developer.android.com/guide/topics/providers/content-provider-creating.html#MIMETypes
- e.g., if a provider's authority is com.example.app.provider & it exposes a table namedtable1, the MIME type for
  - multiple rows in table1 is:

    vnd.android.cursor.**dir**/vnd.com.example.provider.table1
  - a single row of table1:

    vnd.android.cursor.**item**/vnd.com.example.provider.table1

Define MIME Types for Files

- If a provider offers files, implement getStreamTypes()
  - Returns a String array of MIME types for files your provider can return for a given content URI
  - Filter MIME types offered by MIME type filter argument, so return only MIME types client wants
- e.g., consider a provider that offers photo images as files in .jpg, .png, & .gif format
  - If an app calls ContentResolver.getStreamTypes() with filter string image/*, then ContentProvider. getStreamTypes() should return array { "image/jpeg", "image/png", "image/gif"}
  - If an app's only wants .jpg files it can call ContentResolver. getStreamTypes() with filter string *\/jpeg, & ContentProvider. getStreamTypes() should return {"image/jpeg"}

Declaring ContentProvider

- Declare ContentProvider with <provider> in AndroidManifest.xml
  - The name attribute is the fully qualified name of the ContentProvider subclass
  - The authorities attribute is part of content: URI that identifies the provider
  - e.g., if the ContentProvider subclass is MyContentProvider:

    <provider
      android:name="course.examples.ContentProviders.MyContentProvider"
      android:authorities="course.examples.contentproviders.mycontentprovider"
    . . . /> </provider>
- Authorities attribute omits path part of a content:// URI
  - e.g., Any subtables defined by MyContentProvider not defined in the manifest

    content://course.examples.contentproviders.mycontentprovider/table1
- The authority is what identifies ContentProvider, not the path
  - ContentProvider can interpret the path part of the URI in any way it chooses

    http://developer.android.com/guide/topics/manifest/provider-element.html

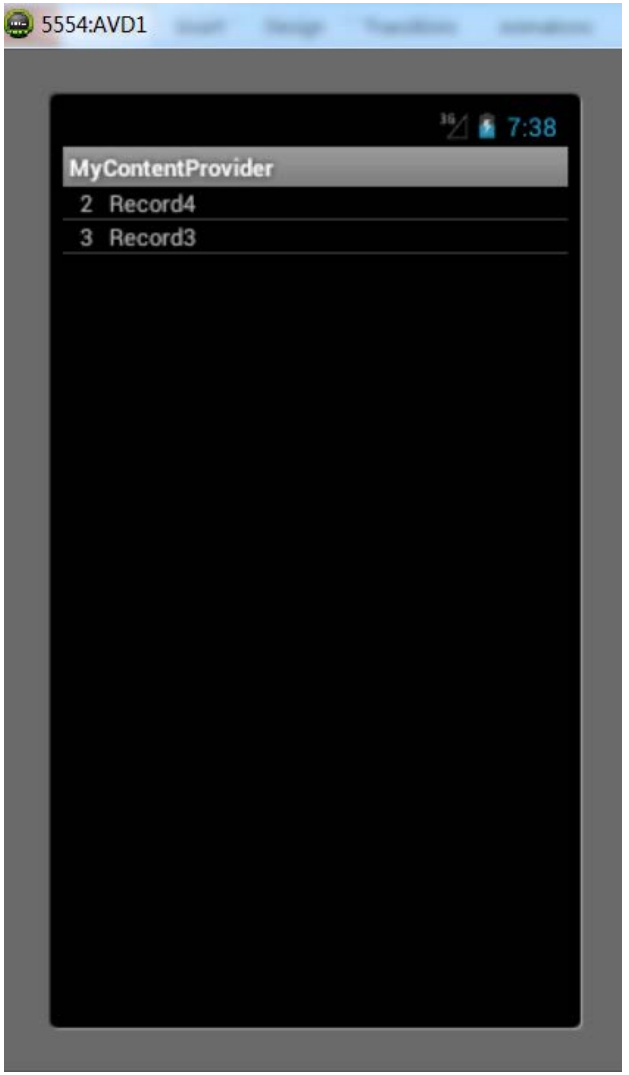
Handling Large Data in Providers

- Although the data model for ContentProvider has a blob type, it's only appropriate for small binary array (less than 50K)
  - e.g., a small icon or short audio clip
- For large binary data the application should put a content:// URI in the table
  - e.g., photographs or complete songs,

- If binary data is small (< 50K) then enter data directly into the table & read from cursor using Cursor.getBlob()
  - getBlob() returns a byte array
- If binary data is large (>= 50K) put a content:// URI into the table
  - The content:// URI specifies a file that should not be read directly by the client
  - Instead, call ContentResolver.openInputStream(Uri uri) to get an InputStream object to read the data

Sync MyContentProvider

- Illustrates how to implement a simple ContentProvider
- Stores the DataRecord objects in a HashMap
- Supports all the ContentProvider "CRUD" operations
  - All of which are implemented as synchronized Java methods
- Client Activity accesses the ContentProvider using synchronous two-way calls made via a ContentResolver



Sync MyContentProvider (cont.)

```
public class MyContentProvider extends ContentProvider {
  public static final Uri CONTENT_URI = Uri.parse(
   "content://course.examples.ContentProviders.myContentProvider/"
   );

  public static final String _ID = "_id", DATA= "data";
  private static final String[] columns = new String[] { _ID, DATA};
  private static final Map<Integer, DataRecord> db =
                                    new HashMap<Integer, DataRecord>();
  private static final String contentTypeSingle =
        "vnd.android.cursor.item/myContentProvider.data.text";
  private static final String contentTypeMultiple =
        "vnd.android.cursor.dir/myContentProvider.data.text";
```

Sync MyContentProvider (cont.)

```
public synchronized int delete(
                    Uri uri, String selection, String[] selectionArgs) {
String requestIdString = uri.getLastPathSegment();
  if (null == requestIdString) {
    for (DataRecord dr : db.values()) { db.remove(dr.get_id()); }
  } else {
    Integer requestId = Integer.parseInt(requestIdString);
    if (db.containsKey(requestId)) { db.remove(requestId);
    }
  }
  return // # of records deleted;
}
```

Sync MyContentProvider (cont.)

```
public synchronized Uri insert(Uri uri, ContentValues values) {
  if (values.containsKey(Data)) {
    DataRecord tmp =
                    new DataRecord(values.getAsString(Data));
    db.put(tmp.get_id(), tmp);
    return Uri.parse(CONTENT_URI +
                                        String.valueOf(tmp.get_id()));
  }
  return null;
}
```

Sync MyContentProvider (cont.)

```
public synchronized Cursor query(
                    Uri uri, String[] projection, String selection,
                              String[] selectionArgs, String sortOrder) {
  String requestIdString = uri.getLastPathSegment();
  MatrixCursor cursor = new MatrixCursor(columns);
  if (null == requestIdString) {
    for (DataRecord dr : db.values()) {
      cursor.addRow(new Object[] {dr.get_id(), dr.get_data()});
    }
 }
...
```

Sync MyContentProvider (cont.)

```
...
else {
    Integer requestId = Integer.parseInt(requestIdString);
    if (db.containsKey(requestId)) {
      DataRecord dr = db.get(requestId);
      cursor.addRow(new Object[] {dr.get_id(), dr.get_data() });
    }
  }
  return cursor;
}
```

ContactProviderActivity

```
public class ContactProviderActivity extends ListActivity {

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ContentResolver cr = getContentResolver();
        ContentValues values = new ContentValues();
        values.put("data", "Record1");
        cr.insert(MyContentProvider.CONTENT_URI, values);

        values.clear(); values.put("data", "Record2");
        cr.insert(MyContentProvider.CONTENT_URI, values);

        values.clear(); values.put("data", "Record3");
        cr.insert(MyContentProvider.CONTENT_URI, values);
```

ContactProviderActivity (cont.)

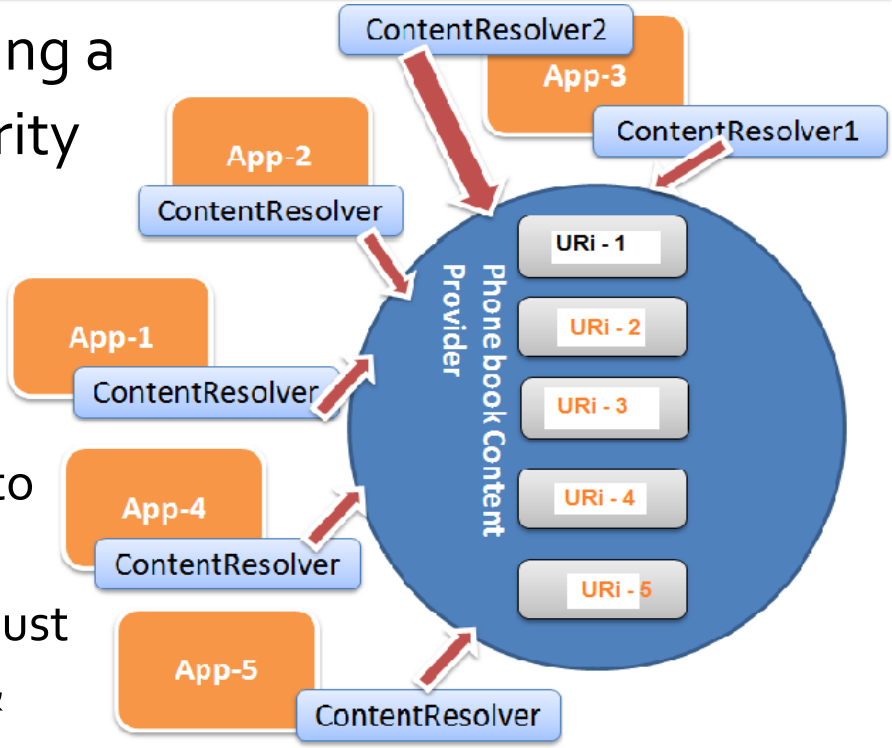
```
cr.delete(Uri.parse(MyContentProvider.CONTENT_URI + "/1"),
                (String) null, (String[]) null);

values.clear(); values.put("data", "Record4");

cr.update(Uri.parse(MyContentProvider.CONTENT_URI +
                "/2"), values,  (String) null, (String[]) null);

Cursor c = cpc.query(MyContentProvider.CONTENT_URI,
                        null, null, null, null);

setListAdapter(new SimpleCursorAdapter(this,
                        R.layout.list_layout, c, new String[] { "_id",
                        "data" }, new int[] { R.id.idString, R.id.data }));
```

MyContentProvider Manifest

```
<manifest ...
 package="course.examples.ContentProviders.myContentProvider"
   ...”>
  <application... >
    <activity android:name=".CustomContactProviderDemo" ...>
       ...
    </activity>
    <provider android:name=".MyContentProvider"
                android:authorities=
       "course.examples.contentproviders.mycontentprovider"
                  android:process=":remote">
    </provider>
  </application>
</manifest>
```

Motivating ContentProviderClient

- ContentResolver stores data providing a mapping from String contentAuthority to ContentProvider
- This mapping is expensive
  - When you call ContentResolver.query(), update(), etc., the URI is parsed apart into its components, the contentAuthority string is identified, & contentResolver must search that map for a matching string, & direct the query to the right provider



- This expensive search occurs during every call since the URI might differ from call to call, with a different contentAuthority as well

http://stackoverflow.com/questions/5084896/using-contentproviderclient-vs-contentresolver-to-access-content-provider

Querying via ContentProviderClient

- ContentResolver.acquireContentProviderClient(Uri uri) returns a ContentProviderClient associated with the ContentProvider that services the content at uri
  - The identified Content Provider is started if necessary
- ContentProviderClient is a direct link to the ContentProvider
  - Needn't constantly re-compute "which provider do I want?"
- The ContentProviderClient has essentially the same interface as ContentProvider
  - Don't forget to call release() when you're done
  - Also, the methods aren't thread-safe

http://developer.android.com/reference/android/content/ContentProviderClient.html

ContactProviderClientActivity

```
public class ContactProviderClientActivity extends ListActivity {

   public void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);

         ContentProviderClient cpc = getContentResolver()
            .acquireContentProviderClient
                                    (MyContentProvider.CONTENT_URI);
         try {
            ContentValues values = new ContentValues();

            values.put("data", "Record1");
            cpc.insert(MyContentProvider.CONTENT_URI, values);
```

ContactProviderClientActivity (cont.)

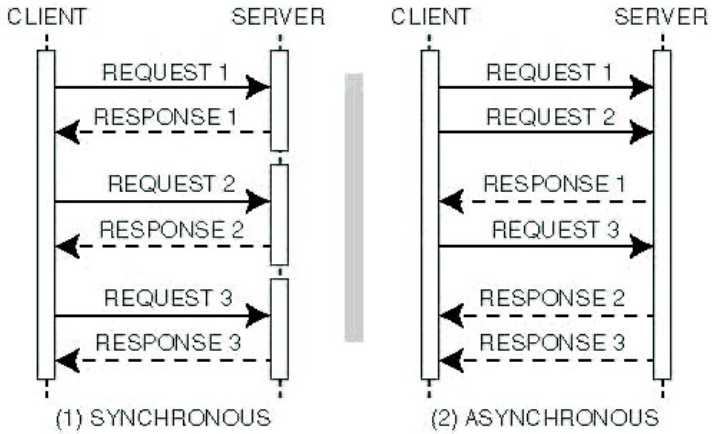
```
values.clear(); values.put("data", "Record2");
cpc.insert(MyContentProvider.CONTENT_URI, values);

values.clear(); values.put("data", "Record3");
cpc.insert(MyContentProvider.CONTENT_URI, values);


cpc.delete(Uri.parse(MyContentProvider.CONTENT_URI + "/1"),
                (String) null, (String[]) null);

values.clear(); values.put("data", "Record4");

cpc.update(Uri.parse(MyContentProvider.CONTENT_URI +
                "/2"), values,  (String) null, (String[]) null);
```

ContactProviderClientActivity (cont.)

```
 Cursor c = cpc.query(MyContentProvider.CONTENT_URI,
                                 null, null, null, null);

 setListAdapter(new SimpleCursorAdapter(this,
                         R.layout.list_layout, c, new String[] { "_id",
                         "data" }, new int[] { R.id.idString, R.id.data }));
} catch (RemoteException e) {
   // ...
}
finally {
    cpc.release();
}
```

Async Access to ContentProviders

- All Activities thus far have invoked synchronous two-way calls to query ContentResolver/Provider
  - Synchronous two-ways can block caller by performing queries on the UI thread, which is bad for lengthy operations, such as loading data



- There are several solutions to this problem
  - Use the LoaderManager & CursorLoader
    - CursorLoader queries the ContentResolver & returns a Cursor
    - Implements the Loader protocol to query cursors & perform the cursor query on a background thread so it does not block the application's UI
  - Use an AsyncQueryHandler
    - A helper class to make asynchronous ContentResolver queries easier

http://www.androiddesignpatterns.com/2012/07/understanding-loadermanager.html

LoaderManager

- Interface associated with an Activity or Fragment for managing one or more Loader instances associated with it
- This helps an application manage longer-running operations in conjunction with the Activity or Fragment lifecycle
- The most common use of LoaderManager is with a CursorLoader
  - Applications can write their own loaders for loading other types of data
- The LoaderManager API was introduced in HONEYCOMB
  - A version of the API at is also available for use on older platforms through FragmentActivity
- You can initialize the Loader with id & a callback location as follows

```
getLoaderManager().initLoader(LOADER_ID, null, mCallbacks);
```

http://developer.android.com/reference/android/app/LoaderManager.html

CursorLoader

- A loader that queries the ContentResolver & returns a Cursor
- This class implements the Loader protocol in a standard way for querying cursors, building onAsyncTaskLoader to perform the cursor query on a background thread
  - It does not block the application's UI
- A CursorLoader must be built with the full information for the query to perform, either through
  - CursorLoader(Context, Uri, String[], String, String[], String) or
  - Creating empty instance with CursorLoader(Context) & filling in desired params with setUri(Uri), setSelection(String), setSelectionArgs(String[]), setSortOrder(String), & setProjection(String[])

http://developer.android.com/reference/android/content/CursorLoader.html

LoaderManager & CursorLoader

- To use the LoaderManager & CursorLoader have your Activity implement the LoaderManager.LoaderCallbacks<Cursor> class & override the following methods:

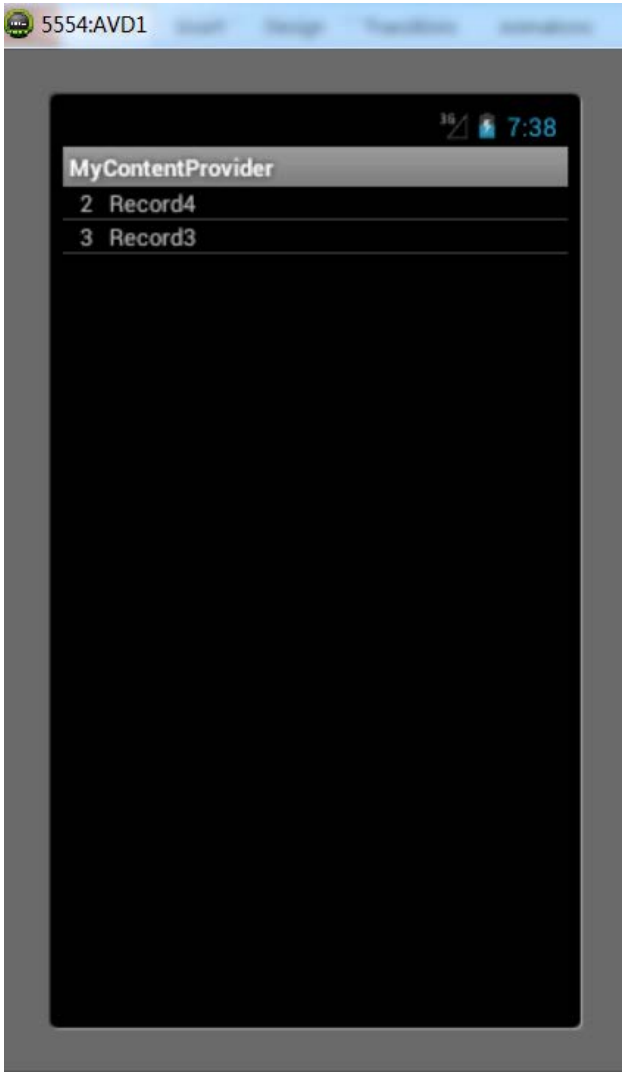
onCreateLoader(int id, Bundle args) – Instantiate & return a new Loader for the given ID
onLoadFinished(Loader<Cursor> loader, D data) – Called when a previously created loader has finished its load
onLoaderReset(Loader<Cursor> loader) – Called when a created loader is being reset, making its data unavailable

- One nice feature of the LoaderManager is that it can automatically callback to your onLoadFinished() method each time the ContentProvider's data is updated

http://developer.android.com/reference/android/app/LoaderManager.LoaderCallbacks.html

LoaderManager ContentProvider

- Illustrates how to implement a ContentProvider that is accessed asynchronously
- Stores the DataRecord objects in a HashMap
- Supports all the ContentProvider "CRUD" operations
  - All of which are implemented as synchronized Java methods
- Client Activity accesses the ContentProvider using asynchronous two-way calls made via a LoaderManager & CursorLoader



ContactProviderActivityAsync

```
public class ContactProviderActivityAsync extends ListActivity
      implements LoaderManager.LoaderCallbacks<Cursor> {
  // The loader's unique id
  private static final int LOADER_ID = 0;

  // The callbacks through which we interact with the LoaderManager
  private LoaderManager.LoaderCallbacks<Cursor> mCallbacks;

  // The adapter that binds our data to the ListView
  private SimpleCursorAdapter mAdapter;

  public void onCreate(Bundle savedInstanceState) {
      … // Initially the same as before
```

ContactProviderActivityAsync (cont.)

```
    String[] dataColumns = {"_id","data"};
    int[] viewIDs = {R.id.idString, R.id.data};

   mAdapter = new SimpleCursorAdapter(this,
        R.layout.list_layout, null, dataColumns, viewIDs, 0);

    setListAdapter(mAdapter); // Associate adapter with ListView

    // The Activity is the callbacks object  for the LoaderManager
    mCallbacks = this;

    // Initialize the Loader with id & callbacks "mCallbacks".
    getLoaderManager().initLoader(LOADER_ID, null, mCallbacks);
}
```

ContactProviderActivityAsync (cont.)

```
public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    // Create a new CursorLoader with query parameters.
     return new CursorLoader(ContentProviderActivityAsync.this,
                MyContentProvider.CONTENT_URI, null, null, null, null);
}

public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
   switch (loader.getId()) {
   case LOADER_ID:
      // Async load complete & data available for SimpleCursorAdapter
      mAdapter.swapCursor(cursor);
   }
   // The listview now displays the queried data
}
```

AsyncQueryHandler

- LoaderManager & CursorLoader only provide a way to access results of asynchronously invoked query() operations on ContentResolvers
  - Other ContentResolver operations are still synchronous
- AsyncQueryHandler invokes all ContentResolver calls asynchronously

  startDelete (int token, Object cookie, Uri uri, String selection, String[] selectionArgs) – Begins an asynchronous delete

  startInsert (int token, Object cookie, Uri uri, ContentValues initialValues) – Begins an asynchronous insert

  startQuery (int token, Object cookie, Uri uri, String[] projection, String selection, String[] selectionArgs, String orderBy) – Begins an asynchronous query

  startUpdate (int token, Object cookie, Uri uri, ContentValues values, String selection, String[] selectionArgs) – Begins an asynchronous update
- It's possible to cancel operations started asynchronously

http://developer.android.com/reference/android/content/AsyncQueryHandler.html

AsyncQueryHandler (cont.)

- AsyncQueryHandler defines callback hook methods that are invoked in a handler thread when operations invoked asynchronously on a ContentResolver have completed

  onDeleteComplete (int token, Object cookie, int result)
  – Called when an asynchronous delete is completed

  onInsertComplete (int token, Object cookie, Uri uri)
  – Called when an asynchronous insert is completed

  onQueryComplete (int token, Object cookie, Cursor cursor)
  – Called when an asynchronous query is completed

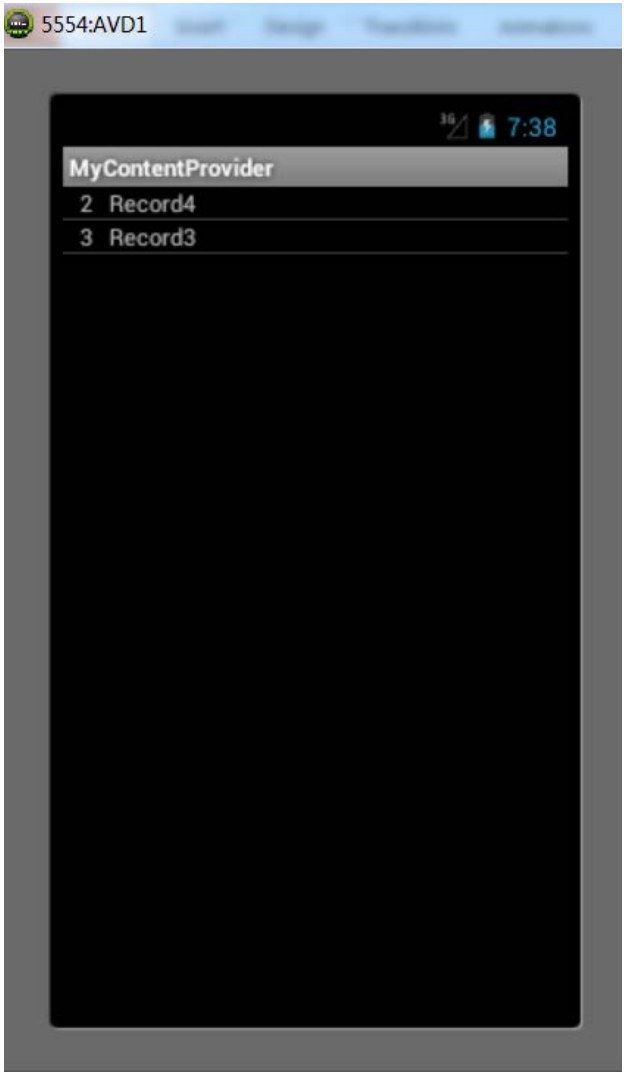
  onUpdateComplete (int token, Object cookie, int result)
  – Called when an asynchronous update is completed

- AsyncQueryHandler implements the Proactor & Asynchronous Completion Token patterns

http://developer.android.com/reference/android/content/AsyncQueryHandler.html

AsyncQueryHandler ContentProvider

- Illustrates how to implement a ContentProvider that is accessed asynchronously
- Stores the DataRecord objects in a HashMap
- Supports all the ContentProvider "CRUD" operations
  - All of which are implemented as synchronized Java methods
- Client Activity accesses the ContentProvider using asynchronous two-way calls made via an AsyncQueryHandler
  - Note the use of the Command & Asynchronous Completion Token patterns



ContactProviderActivityAsync

```
public class ContactProviderActivityAsync extends ListActivity
{
   // The adapter that binds our data to the ListView
   private SimpleCursorAdapter mAdapter;

   // This class is used to implement both the Command pattern
   // (execute) and the Asynchronous Completion Token pattern (by
   // virtue of inheriting from AsyncQueryHandler).
   abstract class CompletionHandler extends AsyncQueryHandler {
      CompletionHandler() {
         super (getContentResolver());
      }
      abstract public void execute();
   }
```

ContactProviderActivityAsync (cont.)

```
class InsertQueryHandler extends CompletionHandler {
    private Object mCommand; private String mValue = null;

    public InsertQueryHandler (String value, Object command)
    { mValue = value; mCommandCookie = commandCookie; }

     public void execute() {
        ContentValues v = new ContentValues(); v.put("data", mValue);
       startInsert(o, mCommand, MyContentProvider.CONTENT_URI, v);
     }

    public void onInsertComplete(int t, Object command, Uri uri)
    {  ((CompletionHandler) command).execute(); }
}
```

ContactProviderActivityAsync (cont.)

```
class DeleteQueryHandler extends CompletionHandler {
    private String mDeleteItem, mUpdateValue, mUpdateItem;

    DeleteQueryHandler(String dI, String uI, String uV)
    { mDeleteItem = dI= uI; mUpdateValue = uV; }

    public void execute() {
       startDelete(0, (Object) new UpdateQueryHandler(mUpdateItem,
                        mUpdateValue), Uri.parse (MyContentProvider.
              CONTENT_URI + mDeleteItem), (String) null, (String[]) null);
    }

    public void onDeleteComplete(int token, Object command, int r) {
       ((CompletionHandler) command).execute();
    }
}
```

ContactProviderActivityAsync (cont.)

```
class UpdateQueryHandler extends CompletionHandler {
    private String mItem, mValue;

    UpdateQueryHandler(String item, String value)
    { mItem = item; mValue = value; }

    public void execute() {
       ContentValues v = new ContentValues(); v.put("data", mValue);
       startUpdate(0, (Object) new QueryQueryHandler(),
          Uri.parse (MyContentProvider.CONTENT_URI + mItem), v,
                (String) null, (String[]) null);
    }

    public void onUpdateComplete(int t, Object command, int r)
   { ((CompletionHandler) command).execute(); }
}
```

ContactProviderActivityAsync (cont.)

```
class QueryQueryHandler extends CompletionHandler {

    public void execute() {
       startQuery(0, null, MyContentProvider.CONTENT_URI,
                  (String []) null, (String) null, (String[]) null, (String) null);
    }

    public void onQueryComplete(int t, Object command, Cursor c) {
       String[] cols = {"_id","data"}; int[] ids = {R.id.idString, R.id.data};
       mAdapter = new SimpleCursorAdapter(
                                          ContactProviderActivityAsync.this,
                                            R.layout.list_layout, c, cols, ids);
       setListAdapter(mAdapter);
    }
}
```

ContactProviderActivityAsync (cont.)

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Insert "Value1", "Value2", and "Value3" into the
    // ContentProvider, then delete item 1 and change the value of
    // item 2 to "Value4".
    new InsertQueryHandler("Value1",
        new InsertQueryHandler("Value2",
            new InsertQueryHandler("Value3",
                new DeleteQueryHandler("/1", "/2", "Value4")))).execute();
  }
```

Source Code Examples

- ContentProviderCustom
- ContentProviderExample
- ContentProviderWithInsertionDeletion
- ContentProviderWithSimpleCursorAdapter