http://www.android.com/

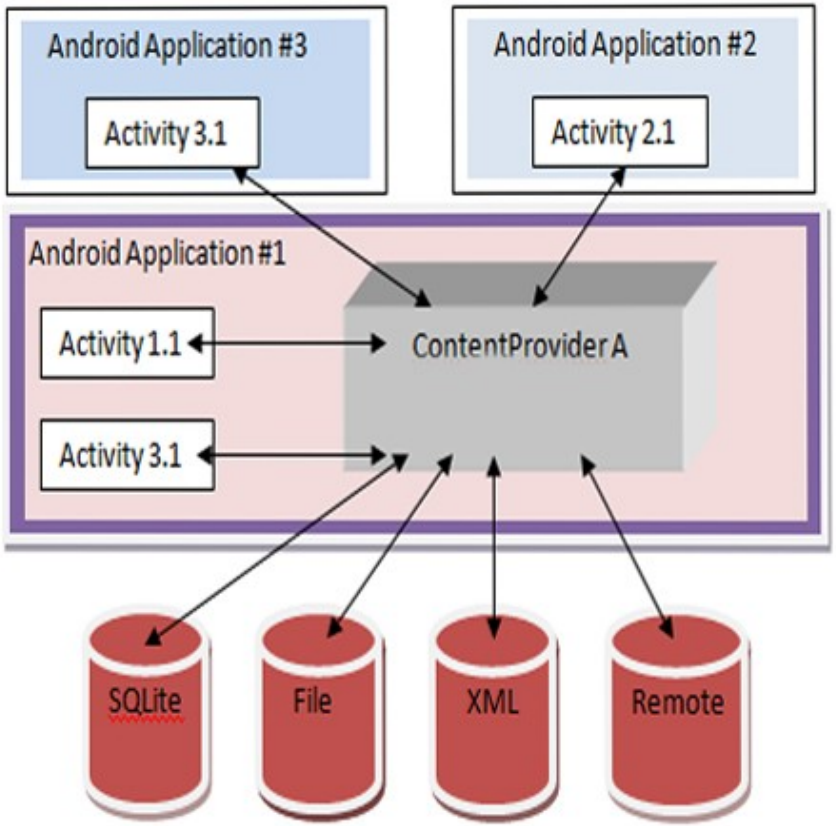
Content providers

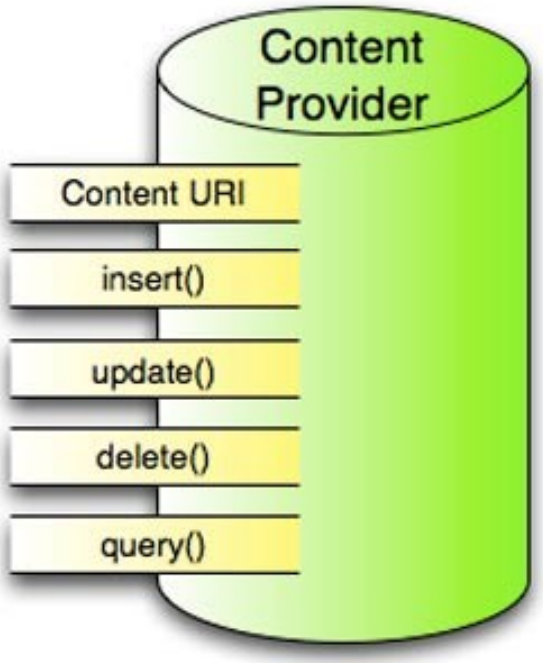
Loaders

CursorLoaders

Content Providers

- Android provides an interface called ContentProvider to act as a bridge between applications, enabling them to share and change each other's data
- Content providers can store and retrieve data, hiding the underlying data source

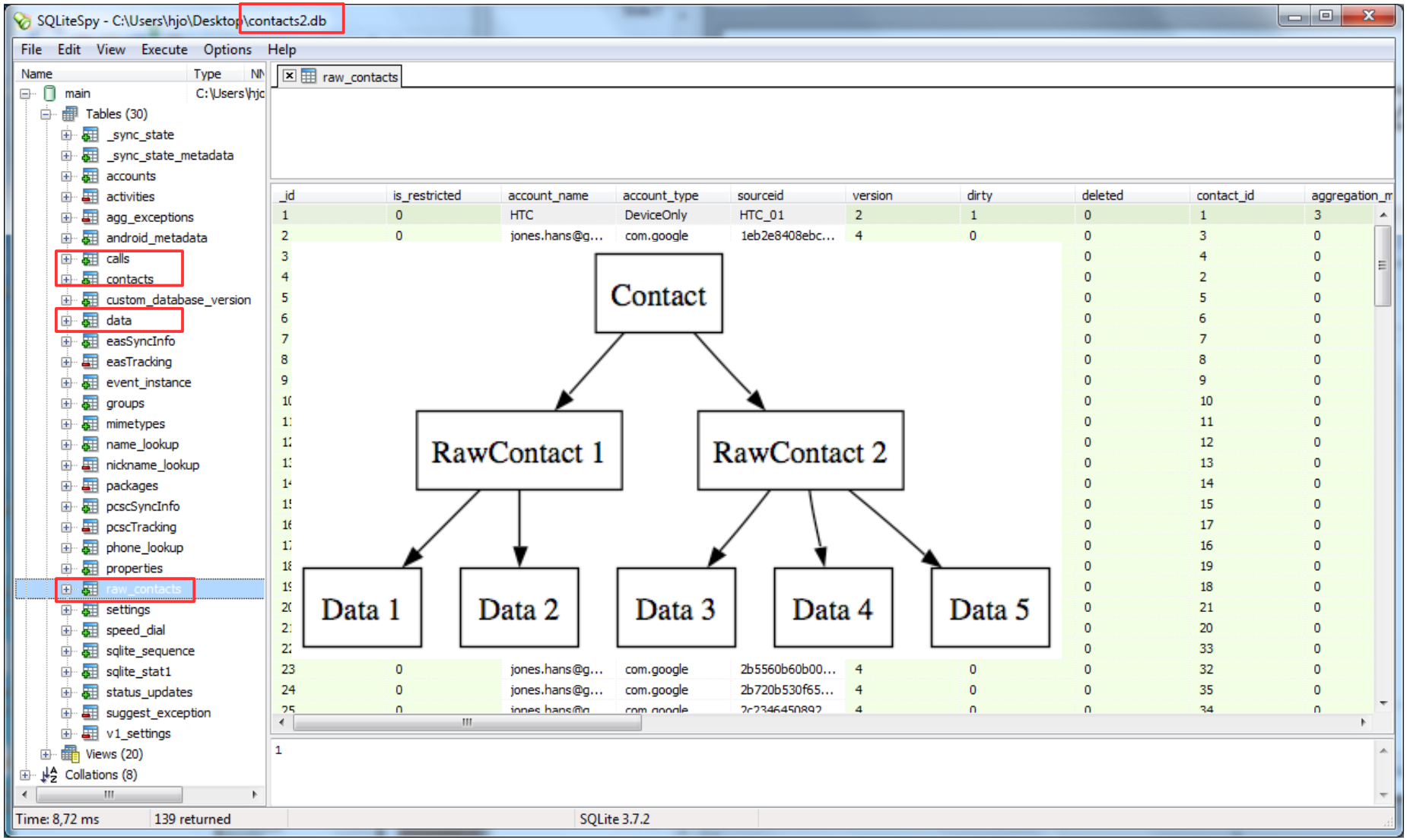




Content provider 1

- Every application has its own sandbox and cannot access data from other applications
  - Content providers make sharing possible - if permission is granted
- A content provider allows a clean separation between the application layer and data layer
- Using the content provider requires a permission setting in the AndroidManifest XML file and it can be accessed using a URI model
- Some native databases Android makes available as content providers are:
- **Contacts** - Retrieve, modify or store the personal contacts. Contact information is stored in a three-tier data model of tables under a ContactsContract object:
  - **ContactsContract.Data** - Contains all kinds of personal data. There is a predefined set of common data, such as phone numbers and email addresses, but the format of this table can be application-specific.
  - **ContactsContract.RawContacts** - Contains a set of Data objects associated with a single account or person.
  - **ContactsContract.Contacts** - Contains an aggregate of one or more Raw Contacts, presumably describing the same person.

Content provider 2



Content provider 3

- **Browser** - Read or modify bookmarks, browser history or web searches
- **CallLog** - View or update the call history
- **LiveFolders** - A special folder whose content is provided by a ContentProvider
- **MediaStore** - Access audio, video and images
- **Setting** - View and retrieve Bluetooth settings, ring tones and other device preferences
- **SearchRecentSuggestions** - Can be configured to operate with a search suggestions provider
- **SyncStateContract** - ContentProvider contract for associating data with a data array account. Providers that want to store this data in a standard way can use this
- **UserDictionary** - Provides user-defined words used by input methods during predictive text input. Applications and input methods can add words to the dictionary. Words can have associated frequency information and locale information
- **SMS, MMS and the Calendar**

Content provider URIs

- Each content provider exposes a public URI (wrapped as a Uri object) that uniquely identifies its data set
- Begins with "content:// "
- Suggested URI format
  - content://<package name>.provider.<custom ContentProvider name>/<DataPath>
- Custom ContentProvider Uri example

```
public static final Uri CONTENT_URI =
Uri.parse("content://se.du.database.BooksProvider");
```

- Android defines CONTENT_URI constants for all native providers, e.g.
  - android.provider.Contacts.Photos.CONTENT_URI
  - android.provider.CallLog.Calls.CONTENT_URI
  - android.provider.ContactsContract.***.CONTENT_URI;
  - ...

Querying a Content Provider

- Some of the methods in the abstract ContentProvider class
  - `public abstract Uri insert (Uri uri, ContentValues values)`
  - `public abstract int update (Uri uri, ContentValues values, String selection, String[] selectionArgs)`
  - `public abstract int delete (Uri uri, String selection, String[] selectionArgs)`
  - `public abstract String getType (Uri uri)`
  - `public abstract Cursor query (Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)`
- ContentProvider acts as an interface that clients use indirectly, through ContentResolver objects or Activities

```
// The ContentResolver class provides applications access to the content model.
// Return a ContentResolver instance for your application's package.
ContentResolver cr = Context.getContentResolver();
// Query the given URI, returning a Cursor over the result set.
Cursor c = cr.query(uri, ..., ..., ...);

// Wrapper around query(android.net.Uri, String[], String, String[], String) that gives the resulting
// Cursor to call startManagingCursor(Cursor) so that the activity will manage its lifecycle for you.
// use LoaderManager and a Loader instead, available via getLoaderManager()
Cursor c = Activity.managedQuery(uri, ..., ..., ...); // depricated!
```

Querying a Content Provider 1

- Retrieve a list of contact names, eventual email address, last time contacted and if they got any phone numbers

```
import android.provider.ContactsContract;
import android.provider.ContactsContract.Contacts;

// Form an array specifying which columns to return
String[] projection = new String[] {
          ContactsContract.Contacts._ID,
          ContactsContract.Contacts.LAST_TIME_CONTACTED,
          ContactsContract.Contacts.DISPLAY_NAME,
          ContactsContract.Contacts.DISPLAY_NAME_SOURCE,
          ContactsContract.Contacts.HAS_PHONE_NUMBER
};

// Get the base URI of the People table in the ContentProvider
Uri contacts = ContactsContract.Contacts.CONTENT_URI;

// Request all records ascending.
Cursor managedCursor = managedQuery(
                contacts,      // Uri
                projection,    // Which columns to return.
                null,          // WHERE clause. Selection
                null,     // WHERE clause value substitution. SelectionArgs (?s in Selection)
                ContactsContract.Contacts.DISPLAY_NAME + " ASC");   // Sort order.
```

```
// WHERE clause substitution example
String[] select = "name=? AND email=?";

String[] selArgs = {"first string",
            "second@string.com"};

Cursor cursor = cr.query(uri,
      prj, select, selArgs, null);
```

Querying a Content Provider 2

- Reading names and email using the Cursor beginning with the first row (if available) depending on sort order

```
if (managedCursor.moveToFirst())
{
    String name, nameSource;
    String nc = ContactsContract.Contacts.DISPLAY_NAME;
    String nsc = ContactsContract.Contacts.DISPLAY_NAME_SOURCE;
     int nameColumn = managedCursor.getColumnIndex(nc);
     int nameSourceColumn = managedCursor.getColumnIndex(nsc);
     ...
     do {
         // Get the field values
         name = managedCursor.getString(nameColumn);
         nameSource = managedCursor.getString(nameSourceColumn);
         // Do something with the values
         ...
     } while (managedCursor.moveToNext());
}
```

Using a content provider 1

- To access a content provider, the application needs to get a contentResolver instance to query, insert, delete, and update the data from the content provider, as shown in the following example (call log is in the contacts2.db)

```
<uses-permission android:name="android.permission.READ_CONTACTS"></uses-permission>
You can find all available permissions in the class android.Manifest.permission.***
```

```
Uri allCalls = android.provider.CallLog.Calls.CONTENT_URI;
// or Uri allCalls = Uri.parse("content://call_log/calls");
// projections controls how many columns the query returns
// Filtering (selection, selectionArgs) let you specify a SQL WHERE clause to filter the query results.
// SortOrder lets you specify a SQL ORDER BY clause to sort the query results
String[] projection = new String[] {Calls._ID, Calls.NUMBER, Calls.TYPE};
String selection = "Calls.NUMBER LIKE '5%'"; // any number that starts with 5
String sortOrder = "Calls.TYPE DESC";

Cursor c = getContentResolver().query(allCalls, null, null, null, null);
if (c.moveToFirst()) {
   do{
      String callType = "Unknown";
      switch (Integer.parseInt(c.getString(c.getColumnIndex(Calls.TYPE))))
      {
            case 1: callType = "Incoming";
               break;
            case 2: callType = "Outgoing";
               break;
            case 3: callType = "Missed";
      }
      Log.v("Content Providers", c.getString(c.getColumnIndex(Calls._ID)) + ", " +
            c.getString(c.getColumnIndex(Calls.NUMBER)) + ", " + callType) ;
   } while (c.moveToNext());
}
```

Using a content provider 2

- Content providers are not only static sources of data. They can also be used to add, update,and delete data, if the content provider application has implemented this functionality

```
<uses-permission android:name="android.permission.WRITE_CONTACTS"></uses-permission>
```

```
// Adding Records
// Using the Contacts content provider, we can, for example,
// add a new record to the contacts database
ContentValues values = new ContentValues();
values.put(Contacts.People.NAME, “Sample User”);
Uri uri = getContentResolver().insert(Contacts.People.CONTENT_URI, values);
Uri phoneUri = Uri.withAppendedPath(uri, Contacts.People.Phones.CONTENT_DIRECTORY)
values.clear();
values.put(Contacts.Phones.NUMBER, “2125551212”);
values.put(Contacts.Phones.TYPE, Contacts.Phones.TYPE_WORK);
getContentResolver().insert(phoneUri, values);

// Updating Records
ContentValues values = new ContentValues();
values.put(People.NOTES, “This is my boss”);
Uri updateUri = ContentUris.withAppendedId(Contacts.People.CONTENT_URI, rowId);
int rows = getContentResolver().update(updateUri, values, null, null);

// Deleting All Records
int rows = getContentResolver().delete(Contacts.People.CONTENT_URI, null, null);

// Deleting Specific Records
int rows = getContentResolver().delete(Contacts.People.CONTENT_URI, People.NAME +
     “=?”, new String[] {“Sample User”});
```

Note!
Old API on the URIs

Using a content provider 3

Manipulating contacts with the three tier/layer model is complicated. To update/delete data you need to get the unique ContactId first

```
// https://stackoverflow.com/questions/4075694/inserting-contacts-in-android-2-2
// Adding a contact with id, phone number and name
private void testAddingAContact1() {
        ArrayList<ContentProviderOperation> ops = new ArrayList<ContentProviderOperation>();
        int rawContactInsertIndex = ops.size();

        ops.add(ContentProviderOperation.newInsert(ContactsContract.RawContacts.CONTENT_URI)
                        .withValue(ContactsContract.RawContacts.ACCOUNT_TYPE, null)
                        .withValue(ContactsContract.RawContacts.ACCOUNT_NAME, null)
                        .build());
        ops.add(ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
                        .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, rawContactInsertIndex)
                        .withValue(ContactsContract.Data.MIMETYPE, ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE)
                        .withValue(ContactsContract.CommonDataKinds.Phone.NUMBER, "0123456789")
                        .build());
        ops.add(ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
                        .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, rawContactInsertIndex)
                        .withValue(ContactsContract.Data.MIMETYPE, ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE)
                        .withValue(ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME, "Mike Sullivan")
                        .build());
        try {
                ContentProviderResult[] res = getContentResolver().applyBatch(ContactsContract.AUTHORITY, ops);
        } catch (Exception e) {
                e.printStackTrace();
        }
}
// Get a specific contact id template, this will get the one from raw contact id but specific searches can be incorprated
public static long getContactId(Context context, long rawContactId) {
        Cursor cur = null;
        try {
                cur = context.getContentResolver().query(ContactsContract.RawContacts.CONTENT_URI,
                                new String[] { ContactsContract.RawContacts.CONTACT_ID },
                                ContactsContract.RawContacts._ID + "=" + rawContactId, null, null);
                if (cur.moveToFirst()) {
                        return cur.getLong(cur.getColumnIndex(ContactsContract.RawContacts.CONTACT_ID));
                }
        } catch (Exception e) {
                e.printStackTrace();
        } finally {
                if (cur != null) {
                        cur.close();
                }
        }
        return -11;
}
```

Create your own content provider

- To create a custom content provider in Android you must extend the abstract ContentProvider class and override the six methods defined within it. **Optionally you can disable methods by returning null**
  - **onCreate():** which is called to initialize the provider
  - **query():** Receives a request from a client. The result is returned as a Cursor object. (enable **query** and **onCreate** for a **read only provider**)
  - **insert():** Inserts a new record into the content provider.
  - **delete():** Deletes an existing record from the content provider.
  - **update():** Updates an existing record from the content provider.
  - **getType():** Returns the MIME type of the data at the given URI.
- Within your content provider, you may choose to store your data however you like: traditional file system, XML, SQLite database, or even through web services
- Register the Content Provider with it's class name in your app-providers AndroidManifest (in this case the books DataBaseTest)

```
<provider android:name=".BooksProvider"
    android:authorities="se.du.database.BooksProvider"
    android:exported="true"/>
```

Use your own content provider

- **CPExamplesLoader** project - demo of a sligtly modified content provider user class from this url: http://www.devx.com/wireless/Article/41133/1954

```
ContentResolver mCR = getContentResolver();
Uri uri = Uri.parse("content://se.du.database.BooksProvider/titles");

//---add a book title---
ContentValues values = new ContentValues();
values.put(TITLE, "C# 2008 Programmer's Reference");
values.put(ISBN, "0470285818");
values.put(PUBLISHER, "Hans Jones");
mCR.insert(uri, values);

//---print all books sorted on title ascending---
Cursor c = managedQuery(uri, null, null, null, "title ASC");
if (c.moveToFirst()) {
     do{
          Toast.makeText(this, c.getString(c.getColumnIndex(_ID)) + ", " +
          c.getString(c.getColumnIndex(TITLE)) + ", " +
          c.getString(c.getColumnIndex(ISBN)) + ", " +
          c.getString(c.getColumnIndex(PUBLISHER)), Toast.LENGTH_LONG).show();
     } while (c.moveToNext());
}
//---update a specific book title---
Uri bookuri = Uri.parse("content://se.du.database.BooksProvider/titles/2");
ContentValues editedValues = new ContentValues();
editedValues.put(TITLE, "Updated title test");
mCR.update(bookuri, editedValues, null, null);
```

Your content provider

```
public class BooksProvider extends ContentProvider {
      private SQLiteDatabase mBooksDB;
      private ContentResolver mCR;
      private static final UriMatcher sUriMatcher;
      static{
            sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
            sUriMatcher.addURI(PROVIDER_NAME, DATABASE_TABLE, BOOKS);
            sUriMatcher.addURI(PROVIDER_NAME, DATABASE_TABLE + "/#", BOOK_ID);
      }
      @Override
      public boolean onCreate() {
            mCR = getContext().getContentResolver();
            DBAdapter db = new DBAdapter(getContext());
            mBooksDB = db.getDB();
            return (mBooksDB == null)? false:true;
      }
      @Override
      public Cursor query(Uri uri, String[] projection, String selection,
                          String[] selectionArgs, String sortOrder) {
            SQLiteQueryBuilder sqlBuilder = new SQLiteQueryBuilder();
            sqlBuilder.setTables(Consts.DATABASE_TABLE);
            //---if getting a particular book---
            if (sUriMatcher.match(uri) == Consts.BOOK_ID)
                  sqlBuilder.appendWhere(Consts.KEY_ROWID + " = " + uri.getPathSegments().get(1));
            if (sortOrder==null || sortOrder=="")
                  sortOrder = Consts.KEY_TITLE;
Cursor c = sqlBuilder.query(mBooksDB, projection, selection, selectionArgs, null, null, sortOrder);
            //---register to watch a content URI for changes---
            c.setNotificationUri(mCR, uri);
            return c;
            // By default, the query result is sorted using the title field.
            // The resulting query is returned as a Cursor object.
      }
```

Content providers permissions

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

- ContentProvider permissions (applied to the <provider> tag) restrict who can access the data (default it is open?)
  - android:permission restricts who can use the associated object
  - android:readPermission and android:writePermission restricts who can read and write and have precedence over permission
  - The permissions are checked when you first retrieve a provider (if you don't have permissions, a SecurityException will be thrown)

```
<application android:name=".DownloadManagerActivity" ...
   <provider android:name=".DownloadProvider"
      android:authorities="se.du.downloadmanager"
      android:exported="true"
      android:permission="android.permission.ACCESS_DOWNLOAD_MANAGER" />
</application>
<!-- Allow access to the Download Manager -->
<permission android:name="android.permission.ACCESS_DOWNLOAD_MANAGER"
    android:label="@string/downloadManager"
    android:description="@string/downloadManager"
    android:protectionLevel="signature" />
-----------------------------------------------------------------
<!-- APPs who whiches to use the Download Manager -->
<uses-permission android:name="android.permission.ACCESS_DOWNLOAD_MANAGER" />
```

URI Permissions

http://developer.android.com/guide/topics/security/security.html

- The standard permission system described so far is often not sufficient when used with content providers
- A typical example is attachments in a mail application. Access to the mail should be protected by permissions, since this is sensitive user data
- However, if a URI to an image attachment is given to an image viewer, that image viewer will not have permission to open the attachment since it has no reason to hold a permission to access all e-mail
- The solution to this problem is per-URI permissions: when starting an activity or returning a result to an activity, the caller can set
  - Intent.FLAG_GRANT_READ_URI_PERMISSION and/or
  - Intent.FLAG_GRANT_WRITE_URI_PERMISSION
- Granting of fine-grained URI permissions require some cooperation with the content provider holding those URIs and need a declaration that they support it through the
  - android:grantUriPermissions attribute or <grant-uri-permissions> tag

Loaders

http://developer.android.com/guide/components/loaders.html

- Introduced in Android 3.0, loaders make it easy to asynchronously load data in an activity or fragment
- Loaders have these characteristics
  - They are available to every Activity and Fragment
  - They provide asynchronous loading of data
  - They monitor the source of their data and deliver new results when the content changes
  - They automatically reconnect to the last loader's cursor when being recreated after a configuration change. Thus, they don't need to re-query their data
- For a background and a very good description of Loaders you **should** view and listen to the ”Android Loaders RELOADED” and the ”Introduction to Loaders and the LoaderManager” presentions
  - See references at last slide

Loader API

- LoaderManager
  - An abstract class associated with an Activity or Fragment for managing one or more Loader instances
- LoaderManager.LoaderCallbacks
  - A callback interface for a client to interact with the LoaderManager
- Loader
  - An abstract class that performs asynchronous loading of data. This is the base class for a loader
- AsyncTaskLoader
  - Abstract loader that provides an AsyncTask to do the work
- CursorLoader
  - A subclass of AsyncTaskLoader that queries the ContentResolver and returns a Cursor. Does not block the UI thread

Using Loaders 1

- An application that uses loaders typically includes the following
  - An **Activity** or **Fragment**.
  - An instance of the **LoaderManager**.
  - A **CursorLoader** to load data backed by a **ContentProvider**.
    - Alternatively, you can implement your own subclass of **Loader** or **AsyncTaskLoader** to load data from some other source.
  - An implementation for **LoaderManager.LoaderCallbacks**. This is where you create new loaders and manage your references to existing loaders.
  - A way of displaying the loader's data, such as a **SimpleCursorAdapter**.
  - A **data source** - such as a ContentProvider, when using a CursorLoader.
- A loader has 3 states (started, stopped and reset) which the loadermanager automatically change to according to the Activity or Fragment state

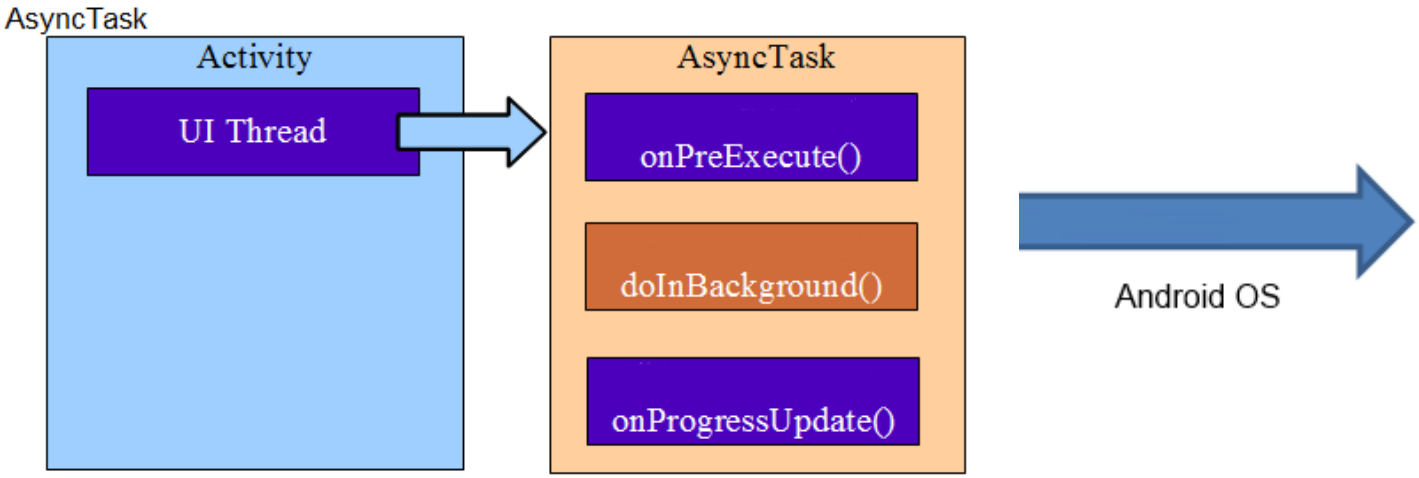
Using Loaders 2

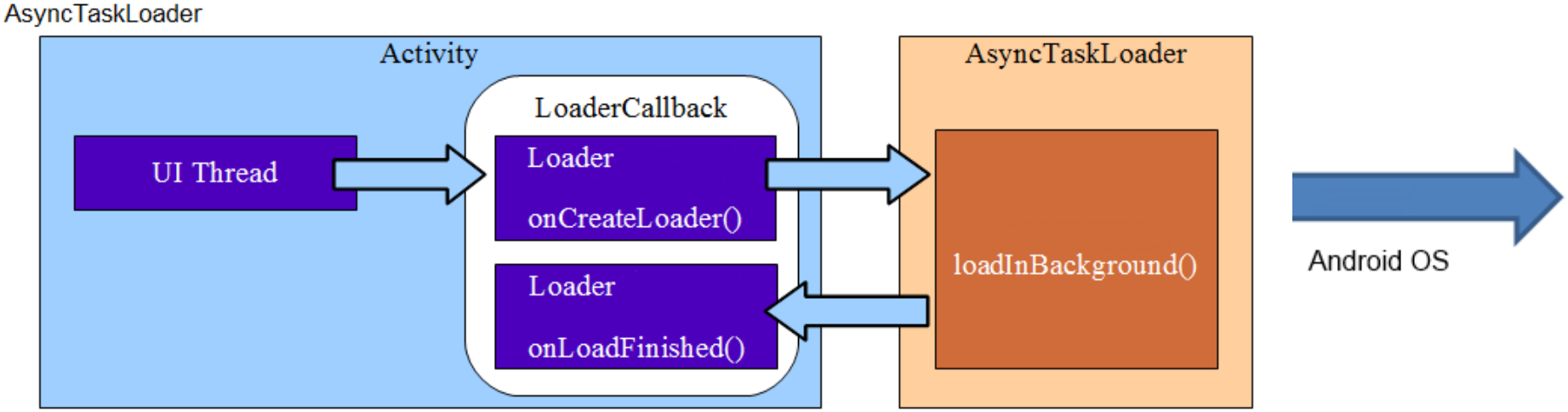
- Starting a loader
  - getLoaderManager().**initLoader**(int LOADER_ID, Bundle arg, LoaderManager.LoaderCallbacks cb_implementation);
  - getLoaderManager().**restartLoader**(int LOADER_ID, Bundle arg, LoaderManager.LoaderCallbacks cb_implementation);
  - **forceLoad**() should only be called when the loader is started and a previous loaded data set is to be ignored
    - `getLoaderManager().initLoader(0, null, this).forceLoad();`
- Handling the LoaderManager.LoaderCallbacks
  - **onCreateLoader()** - Instantiate and return a new Loader for the given LOADER_ID
  - **onLoadFinished()** - Called when a previously created loader has finished its load
  - **onLoaderReset()** - Called when a previously created loader is being reset, thus making its data unavailable
  - Loaders, in particular CursorLoader, are expected to retain their data after being stopped (no reload of data neccessary)

A basic Loader

- The mandatory loader callbacks to implement
  - **onStartLoading()** - Handles a request to start the Loader
  - **onStopLoading()** - Handles a request to stop the Loader
  - **onReset()** - Handles a request to completely reset the Loader
  - **onForceLoad()** from Loader or **loadInBackground()** from AsyncTaskLoader - This is where the bulk of our work is done. This function is called in a background thread and should generate a new set of data to be published by the loader
- Optional callbacks
  - **deliverResult()** [override] - Called when there is new data to deliver to the client. The super class will take care of delivering it

AsyncTask vs. AsyncTaskLoader





Monitoring data

- Loaders need an observer to receive notifications when the data source have changed to be correctly implemented
  - A BroadcastReciver or ContentObserver etc.
- Two Loader methods to help
  - onContentChanged()
    - Used to notify the loader that content has changed
    - If the Loader is started: will call forceLoad()
    - If the Loader is stopped: will set a flag
  - takeContentChanged()
    - Returns the flag value and clears the flag

```
// Handles a request to start the Loader, see the two helper classes in the AsyncTaskLoader example
 @Override
 protected void onStartLoading() {
     if (mApps != null) {
         deliverResult(mApps); // If we currently have a result available, deliver it immediately.
     }

     // Has something interesting in the configuration changed since we last built the app list?
     boolean configChange = mLastConfig.applyNewConfig(getContext().getResources());

     // Take the current flag indicating whether the loader's content had changed while it was stopped.
     // If it had, true is returned and the flag is cleared. OnContentChanged() is set in PackageIntentReceiver
     if (takeContentChanged() || mApps == null || configChange) {
         // If the data has changed since the last time it was loaded or is not currently available, start a load.
         forceLoad();
     }
 }
```

AsyncTaskLoader<D> 1

- A simple **Activity** receiving a List<String> callback from the loader

```
public class MainActivity extends ListActivity implements LoaderManager.LoaderCallbacks<List<String>> {
	private static final String TAG ="AsyncTaskLoader2";

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		getLoaderManager().initLoader(0, null, this);
	}

	// Instantiate and return a new Loader for the given ID.
	@Override
	public Loader<List<String>> onCreateLoader(int id, Bundle args) {
		Log.d(TAG, "onCreateLoader()");
		return (new AppListLoader(this));
	}

	// Called when a previously created loader has finished its load.
	@Override
	public void onLoadFinished(Loader<List<String>> loader, List<String> apps) {
		Log.d(TAG, "onLoadFinished()");
		// fill the list
		setListAdapter(new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, apps));
	}

	// Called when a previously created loader is being reset, thus making its data unavailable.
	@Override
	public void onLoaderReset(Loader<List<String>> loader) {
		Log.d(TAG, "onLoaderReset()");
	}
}
```

See the example project AsyncTaskLoader

AsyncTaskLoader<D> 2

- The important methods in the **AsyncTaskLoader** class

```
public class AppListLoader extends AsyncTaskLoader<List<String>> {
   // Helper for determining if the configuration has changed in an interesting way so we need to rebuild the app list.
   private final InterestingConfigChanges mLastConfig = new InterestingConfigChanges();
   private List<String> mApps;

   // Handles a request to start the Loader
    @Override
    protected void onStartLoading() {
        if (mApps != null) {
            deliverResult(mApps); // If we currently have a result available, deliver it immediately.
        }

        // Has something interesting in the configuration changed since we last built the app list?
        boolean configChange = mLastConfig.applyNewConfig(getContext().getResources());

        // Take the current flag indicating whether the loader's content had changed while it was stopped.
        // If it had, true is returned and the flag is cleared.
        if (takeContentChanged() || mApps == null || configChange) {
            // If the data has changed since the last time it was loaded or is not currently available, start a load.
            forceLoad();
        }
    }

     // This is where the bulk of our work is done.  This function is called in a background
     // thread and should generate a new set of data to be published by the loader.
    @Override
    public List<String> loadInBackground() {
      List<String> entries = new ArrayList<String>();
      entries.add("..."); // Do the work! In this case we retrieve all installed apps and content providers
      // And when done!
      return entries;
    }
```

AsyncTaskLoader<D> 3

- The important methods in the **AsyncTaskLoader** cont.

```
// Called when there is new data to deliver to the client. The super class will take care of delivering it
@Override
public void deliverResult(List<String> apps) {
    if (isReset()) {
        // An async query came in while the loader is stopped. We don't need the result.
        if (apps != null) {
            onReleaseResources(apps);
        }
    }

    List<String> oldApps = apps;
    mApps = apps;

    if (isStarted()) {
        // If the Loader is currently started, we can immediately deliver its results.
        super.deliverResult(apps);
    }
    // At this point we can release the resources associated with 'oldApps' if needed;
    // now that the new result is delivered we know that it is no longer in use.
    if (oldApps != null) {
        onReleaseResources(oldApps);
    }
}

// Handles a request to stop the Loader.
@Override
protected void onStopLoading() {
    // Attempt to cancel the current load task if possible.
    cancelLoad();
}
```

AsyncTaskLoader<D> 4

- The important methods in the **AsyncTaskLoader** cont.

```
// Handles a request to cancel a load.
@Override
public void onCanceled(List<String> apps) {
    super.onCanceled(apps);
    // At this point we can release the resources associated with 'apps' if needed.
    onReleaseResources(apps);
}

// Handles a request to completely reset the Loader.
@Override
protected void onReset() {
    super.onReset();

    // Ensure the loader is stopped
    onStopLoading();

    // At this point we can release the resources associated with 'apps' if needed.
    if (mApps != null) {
        onReleaseResources(mApps);
        mApps = null;
    }
}

// Helper function to take care of releasing resources associated with an actively loaded data set.
protected void onReleaseResources(List<String> apps) {
    // For a simple List<> there is nothing to do. For something like a Cursor, we would close it here.
}
```

AsyncTaskLoader<D> 5

- Observer/monitor helper classes in **AsyncTaskLoader**

```
// Helper class for determining if the configuration has changed in an interesting way so we need to rebuild the app list.
    public static class InterestingConfigChanges {
        final Configuration mLastConfiguration = new Configuration();
        int mLastDensity;
        boolean applyNewConfig(Resources res) {
            int configChanges = mLastConfiguration.updateFrom(res.getConfiguration());
            boolean densityChanged = mLastDensity != res.getDisplayMetrics().densityDpi;
            if (densityChanged || (configChanges&(ActivityInfo.CONFIG_LOCALE
                    | ActivityInfo.CONFIG_UI_MODE | ActivityInfo.CONFIG_SCREEN_LAYOUT)) != 0) {
                mLastDensity = res.getDisplayMetrics().densityDpi;
                return true;
            }
            return false;
        }
    }
// Helper class to look for interesting changes to the installed apps so that the loader can be updated.
    public static class PackageIntentReceiver extends BroadcastReceiver {
        final AppListLoader mLoader;
        public PackageIntentReceiver(AppListLoader loader) {
            mLoader = loader;
            IntentFilter filter = new IntentFilter(Intent.ACTION_PACKAGE_ADDED);
            filter.addAction(Intent.ACTION_PACKAGE_REMOVED);
            filter.addAction(Intent.ACTION_PACKAGE_CHANGED);
            filter.addDataScheme("package");
            mLoader.getContext().registerReceiver(this, filter);
            // Register for events related to sdcard installation.
            IntentFilter sdFilter = new IntentFilter();
            sdFilter.addAction(Intent.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE);
            sdFilter.addAction(Intent.ACTION_EXTERNAL_APPLICATIONS_UNAVAILABLE);
            mLoader.getContext().registerReceiver(this, sdFilter);
        }
       @Override
       public void onReceive(Context context, Intent intent) {
            // Tell the loader about the change.
            mLoader.onContentChanged();
        }
    }
```

CursorLoader

- CursorLoader is a Loader dedicated to querying ContentProviders
  - It returns a database **Cursor** as result
  - It performs the database query on a background thread (it inherits from **AsyncTaskLoader**)
  - It replaces **Activity.startManagingCursor(Cursor c)**
  - It manages the Cursorlifecycle according to the Activity Lifecycle → **Never call close()**
  - It monitors the database and returns a new cursor when data has changed → **Never call requery()**
  - **You do not need to create any AsyncTaskLoader, the background work is managed automatically by the system**

CursorLoader 1

- CursorLoader with a ListView and SimpleCursorAdapter

```
public class MyListActivity extends ListActivity implements LoaderManager.LoaderCallbacks<Cursor> {
	private SimpleCursorAdapter mAdapter;

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

	    int flags = 0;
	    // Now create a new list adapter bound to the cursor. SimpleCursorAdapter is designed for binding to a Cursor.
	    mAdapter = new SimpleCursorAdapter(this, // Context.
	            android.R.layout.two_line_list_item, // Specify the row template to use (here, two
	                                    // columns bound to the two retrieved cursor rows).
	            null, // Pass in the cursor to bind to. We pass in a null cursor (since no data has been loaded yet)
	            // Array of cursor columns to bind to.
	            new String[] { ContactsContract.Contacts._ID, ContactsContract.Contacts.DISPLAY_NAME },
	            // Parallel array of which template objects to bind to those columns.
	            new int[] { android.R.id.text1, android.R.id.text2 }, flags);
	        // Bind to our new adapter.
	        setListAdapter(mAdapter);
	        getLoaderManager().initLoader(Consts.LOADER_ID2, null, this);
	}

	@Override
	protected void onResume() {
	        super.onResume();
	        // Starts a new or restarts an existing Loader in this manager if not started already
	        if(!getLoaderManager().getLoader(Consts.LOADER_ID2).isStarted())
	                getLoaderManager().restartLoader(Consts.LOADER_ID2, null, this);
	}

	@Override
	protected void onPause() {
	        getLoaderManager().destroyLoader(Consts.LOADER_ID2);
	        super.onPause();
	}
```

See the example SimpleCursorAdapterLoader

CursorLoader 2

- CursorLoader with a ListView and SimpleCursorAdapter cont.

```
	// Instantiate and return a new Loader for the given ID.
	@Override
	public Loader<Cursor> onCreateLoader(int id, Bundle args) {
		Log.v(Consts.TAG, "onCreateLoader");
		CursorLoader cursorLoader = null;

		switch (id) {
		case Consts.LOADER_ID2:
		   Uri uri = ContactsContract.Contacts.CONTENT_URI;
		   String[] projection = new String[] { ContactsContract.Contacts._ID, ContactsContract.Contacts.DISPLAY_NAME };
		   String selection = ContactsContract.Contacts.IN_VISIBLE_GROUP + " = '" + ("1") + "'";
		   String[] selectionArgs = null;
		   String sortOrder = ContactsContract.Contacts.DISPLAY_NAME + " COLLATE LOCALIZED ASC";
		   cursorLoader = new CursorLoader(MyListActivity.this, uri, projection, selection, selectionArgs, sortOrder);
		   break;
		}
		return cursorLoader;
	}

	// Called when a previously created loader has finished its load
	@Override
	public void onLoadFinished(Loader<Cursor> loader, Cursor c) {
		Log.v(Consts.TAG, "onLoadFinished");
		mAdapter.swapCursor(c);
	}

	// Called when a previously created loader is being reset, and thus making its data unavailable.
	@Override
	public void onLoaderReset(Loader<Cursor> loader) {
		Log.v(Consts.TAG, "onLoaderReset");
		mAdapter.swapCursor(null);
	}
}
```

CursorLoader without Uri

- Custom CursorLoader with a custom ListView used against SQLite with all CursorLoaders benefits
  - Using the same DBAdapter code as earlier against Books DB
  - When we press list items cursor moves to current entry and display extra data
- BookCursorAdapter extends CursorAdapter
  - Adapter that exposes data from a Cursor to a ListView widget
- BookCursorLoader extends SimpleCursorLoader
  - Override the loadInBackground() method from SimpleCursorLoader
- SimpleCursorLoader
  - Based on the API CursorLoaders class: http://stackoverflow.com/questions/7182485/usage-cursorloader-without-contentprovider
- See the DatabaseTest example project
  - DatabaseCursorLoaderActivity

When NOT to use Loaders

- You shouldn’t use Loaders if you need your background tasks to complete
  - Android destroys Loaders together with the Activities/Fragments they belong to
  - You should use services for this kind of stuff instead!
- Keep in mind that Loaders are special components to help you create responsive UIs and to asynchronously load data that this UI component needs
- That’s the reason why Loaders are tied to the lifecycle of their creating components. Do not try to abuse them for anything else!

Loader resources

- Loading Data in the Background tutorial
  - https://developer.android.com/training/load-data-background/index.html
- Android Loaders RELOADED
  - http://www.slideshare.net/cbeyls/android-loaders-reloaded
- How to use Loaders in Android
  - https://www.grokkingandroid.com/using-loaders-in-android/
- Introduction to Loaders and the LoaderManager
  - http://www.androiddesignpatterns.com/2012/07/loaders-and-loadermanager-background.html
  - https://play.google.com/store/apps/details?id=com.adp.loadercustom
- Loader – vogella.com
  - http://www.vogella.com/articles/AndroidBackgroundProcessing/article.html#loader
- CommonsWare Android Components (CWAC)
  - cwac-loaderex: http://commonsware.com/cwac