Content Providers

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Objectives

In this module, you will learn how to:

- Write client code that can read and modify the data managed by a content provider
- Implement a basic content provider to expose structured data to other applications
- Use loaders to retrieve a `Cursor` from a content provider without blocking your application’s main thread

2 Using a Content Provider

2.1 Objectives

After this section, you will be able to:

- Write client code that can read and modify the data managed by a content provider
- Find and use *contract classes* documenting the constants exposed by system content providers
- Use batch access to perform more efficient interaction with a content provider

2.2 Content Provider Overview

A *content provider* is an application component that shares data with other applications.

- Various system content providers manage the user’s contacts, call log, calendar, and other collections of information.
- User-installed applications can expose their own custom data collections.

Typically, a content provider presents data as one or more tables, similar to tables in a database.

- Each row represents one record, such as a single calendar event.
- Each column represents a particular attribute of the records, such as an event start time.

Occasionally, a content provider might expose file data.

- For example, the system contacts content provider can share a contact’s photo.

2.3 Accessing a Content Provider

A client application accesses the data from a content provider with a `ContentResolver` object.

- The `ContentResolver` object provides `query()`, `insert()`, `update()`, and `delete()` methods for accessing data from a content provider.
- The `ContentResolver` object invokes identically-named methods on an instance of a concrete subclass of `ContentProvider`, which typically resides in a separate application process.
- The `ContentProvider` acts as an abstraction layer between its data store and the external presentation of data.
- The `ContentResolver` object and the `ContentProvider` object automatically handle the details of inter-process communication.

Note
In most cases, the `ContentProvider` does not reside in the same application process as the client's `ContentResolver`. However, if you implement a content provider for your application, other components in your application can access it through a `ContentResolver` in exactly the same way as they would a content provider in a different application.

For example, to get a list of the words and their locales from the User Dictionary Provider, you call `ContentResolver.query()`:

```
// Queries the user dictionary and returns results
Cursor cursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,   // The content URI of the words table
    projection,                        // The columns to return for each row
    selectionClause                    // Selection criteria
    selectionArgs,                     // Selection criteria
    sortOrder);                        // The sort order for the returned rows
```

2.4 Content URIs

A *content URI* is a URI that identifies data in a provider. It consists of:

- The *scheme*, which is always `content://` for a content URI
- The *authority*, which is a unique string identifying the specific content provider
- The *path*, which identifies a particular record or collection of records managed by the provider

In the preceding example, the full URI for the "words" table is:

```
content://user_dictionary/words
```

- `content://` — the scheme identifying this as a content URI
- `user_dictionary` — the authority of the system user dictionary provider
- `words` — the path corresponding to the “words” table

The `ContentResolver` uses the authority to identify the content provider to contact.

- An application implementing a `ContentProvider` specifies the provider's authority in the application manifest.

The `ContentProvider` uses the path to choose the table to access.

- A provider usually has a path for each table it exposes.
- Many providers allow you to access a single row in a table by appending an ID value to the end of the URI.
- For example, to retrieve a row whose `_ID` is 4 from user dictionary, you can use this content URI:

  ```
  Uri singleUri = ContentUri.withAppendedId(UserDictionary.Words.CONTENT_URI,4);
  ```

Note
The `Uri` and `Uri.Builder` classes contain convenience methods for constructing well-formed `Uri` objects from strings. The `ContentUris` class contains convenience methods for appending id values to a URI. The previous snippet uses `withAppendedId()` to append an id to the UserDictionary content URI.

2.5 Contract Classes

A *contract class* defines constants that help applications work with the content URIs, column names, and other features of a content provider.

- Contract classes are not included automatically with a provider
- The provider's developer has to define them and then make them available to other developers.

Many of the providers included with the Android platform have corresponding contract classes in the package `android.provider`.

- For example, the User Dictionary Provider has a contract class `UserDictionary` containing content URI and column name constants.
- The content URI for the "words" table is defined in the constant `UserDictionary.Words.CONTENT_URI`. The `UserDictiona` class also contains column name constants.

2.6 Requesting Access Permission

Many content providers require clients to hold a custom access permission to access data from the provider.

- Your client application's manifest must include a `<uses-permission>` element with the permission name defined by the provider.
- The provider may define separate permissions for "read access" (queries) and "write access" (inserts, updates, and deletes).
- For example, the User Dictionary Provider defines the permission `android.permission.READ_USER_DICTIONARY` for applications that want to retrieve data, and a separate `android.permission.WRITE_USER_DICTIONARY` permission for inserting, updating, or deleting data.

2.7 Constructing a Query

The `ContentResolver.query()` method requires several arguments:

uri
The URI, using the `content://` scheme, for the content to retrieve.

projection
A list of which columns to return. Passing `null` returns all columns, which can be inefficient.

selection
A filter declaring which rows to return, formatted as an SQL `WHERE` clause (excluding the `WHERE` itself). Passing `null` returns all rows for the given URI.

selectionArgs
You may include `?s` in selection, which are replaced by the values from *selectionArgs*, in the order that they appear in the selection.

sortOrder
How to order the rows, formatted as an SQL `ORDER BY` clause (excluding the `ORDER BY` itself). Passing `null` uses the default sort order, which may be unordered.

The `ContentResolver.query()` client method always returns a `Cursor` containing the columns specified by the query's projection for the rows that match the query's selection criteria.

Note
If an internal error occurs, the results of the query depend on the particular provider. It may choose to return `null`, or it may throw an `Exception`.

Some `Cursor` implementations automatically update the object when the provider's data changes, or trigger methods in an observer object when the `Cursor` changes, or both.

2.8 A Query Example

```
// A "projection" defines the columns that will be returned for each row
String[] projection =
{
    UserDictionary.Words._ID,    // Contract class constant for the _ID column name
    UserDictionary.Words.WORD,   // Contract class constant for the word column name
    UserDictionary.Words.LOCALE  // Contract class constant for the locale column name
};

// Defines a string to contain the selection clause
String selectionClause = null;

// An array to contain selection arguments
String[] selectionArgs = null;

// Gets a word from the UI
String searchString = mSearchWord.getText().toString();

// Remember to insert code here to check for invalid or malicious input.

// If the word is the empty string, get everything. Otherwise...
if (!TextUtils.isEmpty(searchString)) {
    // Construct a selection clause that matches the word that the user entered.
    selectionClause = UserDictionary.Words.WORD + " = ?";

    // Use the user's input string as the (only) selection argument.
    selectionArgs = new String[]{ searchString };
}

// An ORDER BY clause, or null to get results in the default sort order
String sortOrder = null;

// Does a query against the table and returns a Cursor object
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,  // The content URI of the words table
    projection,                       // The columns to return for each row
    selectionClause                   // Either null, or the word the user entered
    selectionArgs,                    // Either empty, or the string the user entered
    sortOrder);                       // The sort order for the returned rows

// Some providers return null if an error occurs, others throw an exception
if (null == mCursor) {
     // Insert code here to handle the error.
} else if (mCursor.getCount() < 1) {
        // If the Cursor is empty, the provider found no matches
} else {
    // Insert code here to do something with the results
}
```

This query is analogous to the SQL statement:

```
SELECT _ID, word, locale FROM words WHERE word = <userinput> ORDER BY word ASC;
```

Directly concatenating external untrusted data into raw SQL statements can lead to *SQL injection* attacks.

For example, in the following selection clause:

```
// Constructs a selection clause by concatenating the user's input to the column name
String selectionClause =  "var = " + userInput;
```

the user could enter `"nothing; DROP TABLE *;"` for `mUserInput`, which would result in the selection clause `var = nothing; DROP TABLE *;`. Since the selection clause is treated as an SQL statement, this might cause the provider to erase all of the tables in the underlying SQLite database (unless the provider is set up to catch SQL injection attempts).

When incorporating untrusted data — such as user input — into a query, you should always use a selection clause that with `?` as a replaceable parameter and a separate array of selection arguments. The selection argument is incorporated into the query as a single argument rather than being directly concatenated into the selection string.

2.9 Inserting Data

To insert data into a provider, call the `ContentResolver.insert()` method.

- This method inserts a new row into the provider and returns a content URI for that row.

This example shows how to insert a new word into the User Dictionary Provider:

```
// Defines a new Uri object that receives the result of the insertion
Uri newUri;

// Defines an object to contain the new values to insert
ContentValues newValues = new ContentValues();

// Sets the values of each column and inserts the word.
newValues.put(UserDictionary.Words.APP_ID, "example.user");
newValues.put(UserDictionary.Words.LOCALE, "en_US");
newValues.put(UserDictionary.Words.WORD, "insert");
newValues.put(UserDictionary.Words.FREQUENCY, "100");

newUri = getContentResolver().insert(
    UserDictionary.Word.CONTENT_URI,   // the user dictionary content URI
    newValues                          // the values to insert
);
```

The content URI returned in `newUri` identifies the newly-added row, with the following format:

```
content://user_dictionary/words/<id_value>
```

Tip
To get the value of `_ID` from the returned `Uri`, call `ContentUris.parseId()`.

2.10 Updating Data

To update one or more rows, use a `ContentValues` object with the updated value and selection criteria.

- Invoke `ContentResolver.update()` to perform the update.
- You need to add values to the `ContentValues` object for only the columns you're updating.
- If you want to clear the contents of a column, set the value to `null`.

For example, the following snippet changes all the rows whose locale has the language "en" to a have a locale of `null`:

```
// Defines an object to contain the updated values
ContentValues updateValues = new ContentValues();

// Defines selection criteria for the rows you want to update
String selectionClause = UserDictionary.Words.LOCALE +  "LIKE ?";
String[] selectionArgs = {"en_%"};

// Defines a variable to contain the number of updated rows
int rowsUpdated = 0;

// Sets the updated value and updates the selected words.
updateValues.putNull(UserDictionary.Words.LOCALE);

rowsUpdated = getContentResolver().update(
    UserDictionary.Words.CONTENT_URI,  // the user dictionary content URI
    updateValues                       // the columns to update
    selectionClause                    // the column to select on
    selectionArgs                      // the value to compare to
);
```

2.11 Deleting Data

Deleting rows is similar to retrieving row data.

- Invoke `ContentResolver.delete()` to perform the update.
- Specify selection criteria for the rows you want to delete.
- The method returns the number of rows deleted.
- For example, the following snippet deletes rows whose `appid` matches "user".

```
// Defines selection criteria for the rows you want to delete
String selectionClause = UserDictionary.Words.APP_ID + " LIKE ?";
String[] selectionArgs = {"user"};

// Defines a variable to contain the number of rows deleted
int rowsDeleted = 0;

// Deletes the words that match the selection criteria
rowsDeleted = getContentResolver().delete(
    UserDictionary.Words.CONTENT_URI,   // the user dictionary content URI
    selectionClause                    // the column to select on
    selectionArgs                      // the value to compare to
);
```

2.12 Batch Access

Batch access allows you to perform multiple operations with a content provider in a single `ContentResolver` call.

- The request is much more efficient, as it requires only one IPC call.
- Depending on the specific content provider implementation, a content provider might implement a batch access as a single atomic transaction.

To access a content provider in batch mode:

1. Create an `ArrayList` of `ContentProviderOperation` objects.
2. Invoke `ContentResolver.applyBatch()` to send the operations to the specified content provider, supplying the provider's *authority* string and the `ContentProviderOperation` array as arguments.
3. The return value is an array of `ContentProviderResult` objects, each one representing the result of the corresponding `ContentProviderOperation` request.

2.13 Batch Access, the `ContentProviderOperation` and `ContentProviderResult` Classes

The `ContentProviderOperation` class has a set of static methods that return *builders* for each type of operation.

- To create a `ContentProviderOperation` object:
  1. Obtain an appropriate builder.
  2. Use the builder methods to configure the parameters of the operation.
  3. Invoke the `build()` method to create the final `ContentProviderOperation` object.

The `ContentProviderResult` object has two public fields, one of which is set depending on the corresponding operation:

`Integer count`
The count of rows affects by a delete or update operation

`Uri uri`
The `Uri` of a newly inserted row

2.14 Batch Access, Example

This shows an example of batch inserts into the User Dictionary Provider:

```
// Declare the operations ArrayList
ArrayList<ContentProviderOperation> batchOps = new ArrayList<ContentProviderOperation>();

// Declare an array of new terms
String[] words = {"foo", "bar", "wibble"};

// Declare a new array will contain the Uris of the new records
Uri newUris[words.length];

// Create a set of insert ContentProviderOperations
for (int index; words.length; index++) {
        batchOps.add(ContentProviderOperation.newInsert(UserDictionary.Word.CONTENT_URI)
                .withValue(UserDictionary.Words.APP_ID, "example.user")
                .withValue(UserDictionary.Words.LOCALE, "en_US")
                .withValue(UserDictionary.Words.WORD, words[index])
                .withValue(UserDictionary.Words.FREQUENCY, "100")
                .build());
}

// Invoke the batch insertion
ContentProviderResult[] opResults
        = getContentResolver().applyBatch(UserDictionary.AUTHORITY, batchOps);

// Extract the Uris of the new records
for (int index; opResults.length; index++) {
        newUris[index] = opResults[index].uri;
}
```

2.15 Topic Summary

You should now be able to:

- Write client code that can read and modify the data managed by a content provider
- Find and use *contract classes* documenting the constants exposed by system content providers
- Use batch access to perform more efficient interaction with a content provider

3 Accessing Contact Information

3.1 Objectives

After this section, you will be able to:

- Describe the primary tables exposed by the system contacts content provider and the data they contain
- Perform basic queries on the system contacts content provider

3.2 Android Contact Management Overview

The Android system contacts content provider manages an extensible database of contact-related information.

- Android applications can define accounts, which can store contact information — including custom contact information — in the system contacts provider.
- The system contacts provider aggregates the information from all accounts to present unified contact information.

For example, Facebook, Twitter, Google applications, and other apps can all define accounts that store contact information in the system contacts provider.

- The system contact provider aggregates common contact information from different accounts, so a single Android contact might contain email addresses from a Google account, a Twitter username, phone numbers from Facebook, etc.

3.3 Contacts Content Provider Access

The `ContactsContract` class and related inner classes and interfaces expose access to the system contacts provider.

The Android system contacts provider manages several tables of contact information, including:

`ContactsContract.RawContacts`
A row in the `ContactsContract.RawContacts` table represents a set of data describing a person and associated with a single account (for example, one of the user's Gmail accounts).

`ContactsContract.Contacts`
A row in the `ContactsContract.Contacts` table represents an aggregate of one or more raw contacts presumably describing the same person. When data in or associated with the RawContacts table is changed, the affected aggregate contacts are updated as necessary.

`ContactsContract.Data`
A row in the `ContactsContract.Data` table store a single piece of contact information (such as a phone number) and its associated metadata (such as whether it is a work or home number) for a raw contact. The set of data kinds that can be stored in this table is open-ended. There is a predefined set of common kinds, defined by subclasses of `ContactsContract.CommonDataKinds`, but any application can add its own data kinds.

Note
You application needs the `android.permission.READ_CONTACTS` permission to perform contact queries, and the `android.permission.WRITE_CONTACTS` permission to insert, update, or delete contact data.

4 Basic Contact Query Example

The following example executes a query returning all contacts that have at least one phone number associated with them:

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

// ...

String[] projection = {Contacts._ID, Contacts.DISPLAY_NAME};
String selection = Contacts.HAS_PHONE_NUMBER + "=1";
String[] selectionArgs = null;
String orderBy = Contacts.DISPLAY_NAME + " ASC";

Cursor contactsCursor = getContentResolver()
        .query(Contacts.CONTENT_URI, projection, selection, selectionArgs, orderBy);
```

Note
The tables managed by the contacts provider contain many columns of information. Therefore, when querying the contacts provider, you should always provide a projection argument (a `String` array of column names to return) to minimize the amount of data copied.

4.1 Finding a Contact by Partial Name

You can search for contacts based on a partial contact name.

- This can be handy for "type-to-filter" functionality.

For filtered queries, use `ContactsContract.Contacts.CONTENT_FILTER_URI` as the base URI for the query.

- The filter argument should be passed as an additional path segment after this URI.

For example:

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

// ...

Uri lookupUri;
if (TextUtils.isEmpty(filter)) {
        lookupUri = Contacts.CONTENT_URI;
} else {
        lookupUri = Uri.withAppendedPath(Contacts.CONTENT_FILTER_URI,
                        Uri.encode(filter));
}

String[] projection = {Contacts._ID, Contacts.DISPLAY_NAME};
String selection = Contacts.IN_VISIBLE_GROUP + "=1";
String[] selectionArgs = null;
String orderBy = Contacts.DISPLAY_NAME + " ASC";

Cursor contactsCursor = getContentResolver()
        .query(lookupUri, projection, selection, selectionArgs, orderBy);
```

4.2 Finding a Contact by Phone Number

The ContactsContract.PhoneLookup table represents the result of looking up a phone number

- This query is highly optimized for use cases like caller ID.

To perform a lookup you must append the number you want to find to ContactsContract.PhoneLookup.CONTENT_FILTER_U
For example:

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

// ...

String phoneNumber = "415-555-1234";
Uri lookupUri = Uri.withAppendedPath(PhoneLookup.CONTENT_FILTER_URI,
                Uri.encode(phoneNumber));

String[] projection = {Contacts._ID, Contacts.DISPLAY_NAME};
String selection = Contacts.IN_VISIBLE_GROUP + "=1";
String[] selectionArgs = null;
String orderBy = Contacts.DISPLAY_NAME + " ASC";

Cursor contactsCursor = getContentResolver()
        .query(lookupUri, projection, selection, selectionArgs, orderBy);
```

4.3 Retrieving Contact Data

Given a contact ID, you can retrieve information for that contact by querying the ContactsContract.Data table.

- ContactsContract.Data is a generic table that can hold any kind of contact data.
- The kind of data stored in a given row is specified by the row's ContactsContract.Data.MIMETYPE column, which determines the meaning of the generic columns DATA1 through DATA15.

ContactsContract defines several pre-defined data kinds, e.g. ContactsContract.CommonDataKinds.Phone, ContactsContract.CommonDataKinds.Email, etc.

- As a convenience, these classes define data kind specific aliases for DATA1 etc.
- For example, ContactsContract.CommonDataKinds.Phone defines ContactsContract.CommonDataKinds.Phon as an alias for ContactsContract.Data.DATA1.
- These data kind classes also define CONTENT_URI and sometimes CONTENT_FILTER_URI fields as a convenience to do queries on the ContactsContract.Data.

4.4 Example, Retrieving Contact Phone Numbers

The following example shows how to retrieve all of the phone numbers associated with a given contact:

```
Uri baseUri = ContactsContract.CommonDataKinds.Phone.CONTENT_URI;
String[] projection = {
        CommonDataKinds.Phone.TYPE,
        CommonDataKinds.Phone.NUMBER
};
long contactID = 123;
String selection = Data.CONTACT_ID + "=" + contactID;
String[] selectionArgs = null;
```

```
String orderBy = null;

Cursor phoneCursor = getContentResolver()
        .query(baseUri, projection, selection, selectionArgs, orderBy);

int typeIdx = phoneCursor.getColumnIndex(CommonDataKinds.Phone.TYPE);
int phoneIdx = phoneCursor.getColumnIndex(CommonDataKinds.Phone.NUMBER);

while (dataCursor.moveToNext()) {
        int phoneTypeResource = CommonDataKinds.Phone.getTypeLabelResource(dataCursor. ↩
            getInt(typeIdx));
        String phoneType = getString(phoneTypeResource);
        String phoneNumber = dataCursor.getString(phoneIdx);
        builder.append(phoneType)
                .append(": ")
                .append(phoneNumber)
                .append("\n\t");
}
```

4.5 Topic Summary

You should now be able to:

- Describe the primary tables exposed by the system contacts content provider and the data they contain
- Perform basic queries on the system contacts content provider

5 Creating a Content Provider

5.1 Content Provider Overview

A content provider manages access to a central repository of data.

- You implement a provider as a subclass `ContentProvider`, which is the interface between your provider and other applications.
- You can define other related classes, such as a *contract class* to define public constants such as field names to help other applications access the data from your content provider.

In addition to the content provider, you can implement activities in your application that allow the user to query and modify the data managed by your provider.

- You can expose these activities to other applications by registering implicit intent filters in your application manifest to launch appropriate activities.
- Some examples of implicit intent actions you might want to support are:

  `Intent.ACTION_VIEW`
  Launch an activity to view a single record

  `Intent.ACTION_EDIT`
  Launch an activity to edit a single record

  `Intent.ACTION_PICK`
  Launch an activity to select a record from the collection stored in the provider

- In addition to these actions, your intent filters should typically include a MIME type to identify the type of data managed by your provider. The data MIME types are discussed later in this section.

5.2 Why Implement a Content Provider?

You should implement a content provider if you want to:

- Offer complex data or files to other applications
- Allow users to copy complex data from your app into other apps
- Provide custom search suggestions using the search framework
- Expose data collections for use in application widgets

You don't need a content provider to manage access to an SQLite database if the use is entirely within your own application. However, even for use within a single application, you might consider implementing a content provider to:

- Encapsulate access to a data source
- Use the `CursorLoader` class to take advantage of the loader framework added in Honeycomb, which automatically uses a worker thread to query a content provider and return a `Cursor`

5.3 Steps to Implementing a Content Provider

Follow these steps to build your provider:

1. Design the raw storage for your data. A content provider can offer data in two ways:

   "Structured" data
   Data that normally goes into a database, array, or similar structure.

   - Store the data in a form that's compatible with tables of rows and columns.
   - A row represents an entity, such as a person or an item in inventory.
   - A column represents some data for the entity, such a person's name or an item's price.
   - A common way to store this type of data is in an SQLite database, but you can use any type of persistent storage.

   File data
   Data that normally goes into files, such as photos, audio, or videos.

   - Store the files in your application's private space.
   - In response to a request for a file from another application, your provider can offer a handle to the file.

2. Define the provider's external interface.

   - This includes its authority string, its content URIs, and column names.
   - Also define the permissions that you will require for applications that want to access your data.
   - Consider defining all of these values as constants in a separate *contract class* that you can expose to other developers.

3. Define a concrete implementation of the `ContentProvider` class and its required methods.

   - This class is the interface between your data and the rest of the Android system.

5.4 Designing Data Storage

You can store the data in any form you like, and then design the interface to read and write the data as necessary. Options include:

- For table-oriented data, Android includes an SQLite database API that Android's own providers use to store table-oriented data. The `SQLiteDatabase` class is the base class for accessing databases, the `SQLiteOpenHelper` class helps you create databases and handle upgrade scenarios, and the `SQLiteQueryBuilder`

Note
Remember that you don't have to use a database to implement your repository. A provider appears externally as a set of tables, similar to a relational database, but this is not a requirement for the provider's internal implementation.

- For storing file data, Android has a variety of file-oriented APIs. For example, if you're designing a provider that offers media-related data such as music or videos, you can have a provider that combines table data and files.
- For working with network-based data, use classes in `java.net` and `android.net`. You can also synchronize network-based data to a local data store such as a database, and then offer the data as tables or files.

5.5 Data Design Considerations

Table data should always have a *primary key* column that the provider maintains as a unique numeric value for each row.

- You can use this value to link the row to related rows in other tables (using it as a *foreign key*).
- Although you can use any name for this column, using `BaseColumns._ID` is the best choice, because linking the results of a provider query to a `ListView` requires one of the retrieved columns to have the name `_ID`.

If you want to provide bitmap images or other large pieces of file-oriented data, store the data in a file and then provide it indirectly rather than storing it directly in a table.

- If you do this, you need to tell users of your provider that they need to use a `ContentResolver` file method to access the data.

Use the *Binary Large OBject* (*BLOB*) data type to store data that varies in size or has a varying structure.

- For example, you can use a BLOB column to store a small icon or a JSON structure.
- You can also use a BLOB to implement a schema-independent table.
  - In this type of table, you define a primary key column, a MIME type column, and one or more generic columns as BLOB data.
  - The meaning of the data in the BLOB columns is indicated by the value in the MIME type column.
  - This allows you to store different row types in the same table.
  - The contacts provider's "data" table, `ContactsContract.Data`, is an example of a schema-independent table.

5.6 Designing Content URIs

A *content URI* is a URI that identifies data in a provider, consisting of:

An *authority*
: The symbolic name of the entire provider

A path
: A name pointing to a table or file

An ID (optional)
: The last path component, pointing to an individual row in a table

Every data access method of `ContentProvider` receives a content URI as an argument.

- This allows you to determine the table, row, or file to access.

5.7 Content URIs: Selecting an Authority

A provider usually has a single *authority*, which is a unique string that serves as its Android-internal name.

- To avoid conflicts with other providers, you should use Internet domain ownership (in reverse) as the basis of your provider authority.
- Because this recommendation is also true for Android package names, you can define your provider authority as an extension of the name of the package containing the provider.
- For example, if your Android package name is `com.example.<appname>`, you should give your provider the authority `com.example.<appname>.provider`.

5.8 Content URIs: Designing the Path Structure

Typically you should create content URIs from the authority by appending paths that point to individual tables.

- For example, if you have two tables `table1` and `table2`, you combine the authority from the previous example to yield the content URIs `com.example.<appname>.provider/table1` and `com.example.<appname>.provider/table2`.

Paths aren't limited to a single segment, and you don't need to have a table for each level of the path.

5.9 Content URIs: Handling IDs

By convention, providers offer access to a single row in a table by accepting a content URI with an ID value for the row at the end of the URI.

- Also by convention, providers match the ID value to the table's `_ID` column, and perform the requested access against the row that matches.

This convention facilitates a common design pattern for apps accessing a provider.

- The app does a query against the provider and displays the resulting `Cursor` in a `ListView` using a `CursorAdapter`.
- The definition of `CursorAdapter` **requires** one of the columns in the `Cursor` to be `_ID`

The user then picks one of the displayed rows from the UI in order to look at or modify the data.

- The app gets the corresponding row from the `Cursor` backing the `ListView`, gets the `_ID` value for this row, appends it to the content URI, and sends the access request to the provider.
- The provider can then do the query or modification against the exact row the user picked.

5.10 Content URI Pattern Matching with `UriMatcher`

The `UriMatcher` can simply the common task of validating and parsing URIs.

- After instantiating a `UriMatcher` object, you can register URI patterns to map to integer values.
- Then you can test URIs against the registered patterns, and `switch` on the corresponding integer value returned.

When you instantiate a `UriMatcher`, you provide an integer value to return if a given URI matches none of the registered patterns.

- Typically, you should use `UriMatcher.NO_MATCH` for this values.

Use the method `UriMatcher.addURI(String authority, String path, int code)` to map URI patterns to return values.

- In addition to static paths, the `UriMatcher` supports two wildcard characters in patterns:

*	Matches a single path component of any valid characters
#	Matches a single path component consisting solely of numeric characters

The `match(Uri)` method returns the integer corresponding to the pattern matched by the given `Uri` object.

5.11 Implementing ContentProvider MIME Types

A content provider must be able to return MIME types identifying the type of data it provides by implementing the following methods defined by the `ContentProvider` class:

`String getType(Uri)`
: Returns the MIME type of the data at the given URI. All content providers must implement this method.

`String[] getStreamTypes(Uri, String)`
: Returns the supported MIME types matching the String MIME type filter for the data stream specified by `Uri`. Only providers that can return file data need to implement this method; the default implementation returns `null`.

5.12 MIME types for tables

The `getType(Uri)` method returns a String in MIME format that describes the type of data returned by the content URI argument.

- The Uri argument can be a pattern rather than a specific URI; in this case, you should return the type of data associated with content URIs that match the pattern.

For common types of data such as as text, HTML, or JPEG, `getType()` should return the standard MIME type for that data.

- A full list of these standard types is available on the IANA MIME Media Types website.

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:

- Type part
  - `vnd.android.cursor.item/` if the URI pattern is for a single row
  - `vnd.android.cursor.dir/` if the URI pattern is for more than one row
- Provider-specific part
- `vnd.<name>.<type>`
  - The `<name>` value should be globally unique, and the `<type>` value should be unique to the corresponding URI pattern.
  - A good choice for `<name>` is your company's name or some part of your application's Android package name.
  - A good choice for the `<type>` is a string that identifies the table associated with the URI.

For example, if a provider's authority is `com.example.app.provider`, and it exposes a table named `table1`, the MIME type for multiple rows in `table1` is:

```
vnd.android.cursor.dir/vnd.com.example.provider.table1
```

For a single row of `table1`, the MIME type is:

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

5.13 MIME Types for Files

If your provider offers files, implement `String [] getStreamTypes(Uri, String)`.

- The method returns a `String` array of MIME types for the files your provider can return for a given content URI.
- You should filter the MIME types you offer by the MIME type filter argument, so that you return only those MIME types that the client wants to handle.

For example, consider a provider that offers photo images as files in `.jpg`, `.png`, and `.gif` format.

- If an application calls `ContentResolver.getStreamTypes()` with the filter string `image/*` (something that is an "image"), then the `ContentProvider.getStreamTypes()` method should return the array:

  ```
  { "image/jpeg", "image/png", "image/gif"}
  ```

- If the app is only interested in `.jpg` files, then it can call `ContentResolver.getStreamTypes()` with the filter string `*\/jpeg`, and `ContentProvider.getStreamTypes()` should return:

  ```
  {"image/jpeg"}
  ```

If your provider doesn't offer any of the MIME types requested in the filter string, `getStreamTypes()` should return `null`.

5.14 Implementing a Contract Class

A contract class is a public final class that contains constant definitions for the URIs, column names, MIME types, and other meta-data that pertain to the provider.

- The class establishes a contract between the provider and other applications by ensuring that the provider can be correctly accessed even if there are changes to the actual values of URIs, column names, and so forth.

A contract class also helps developers because it usually has mnemonic names for its constants, so developers are less likely to use incorrect values for column names or URIs.

- Since it's a class, it can contain Javadoc documentation.
- Integrated development environments such as Eclipse can auto-complete constant names from the contract class and display Javadoc for the constants.
- Developers can't access the contract class's class file from your application, but they can statically compile it into their application from a `.jar` file you provide.

Tip
The `ContactsContract` class and its nested classes are examples of contract classes.

5.15 Example Contract Class

```
public final class StatusContract {

        private StatusContract() {}

    /** The authority for the contacts provider */
        public static final String AUTHORITY = "com.marakana.android.yamba.provider";
```

```
    /** A content:// style uri to the authority for this table */
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "/status ↩
            ");

    /** The MIME type of {@link #CONTENT_URI} providing a directory of status messages. */
    public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.marakana.status";

    /** The MIME type of a {@link #CONTENT_URI} a single status message. */
    public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.marakana. ↩
        status";

        /**
         * Column definitions for status information.
         */
        public final static class Columns implements BaseColumns {
                private Columns() {}

                /**
         * The name of the user who posted the status message
         * <P>Type: TEXT</P>
                 */
                public static final String USER = "user";

                /**
         * The status message content
         * <P>Type: TEXT</P>
                 */
                public static final String MESSAGE = "message";

                /**
         * The date the message was posted, in milliseconds since the epoch
         * <P>Type: INTEGER (long)</P>
                 */
                public static final String CREATED_AT = "createdAt";

        /**
         * The default sort order for this table
         */
        public static final String DEFAULT_SORT_ORDER = CREATED_AT + " DESC";

        }
}
```

5.16 Implementing permissions

By default, all applications can read from or write to your provider — even if the underlying data is private — because by default your provider does not have permissions set.

- To change this, set permissions for your provider in your manifest file, using attributes or child elements of the `<provider>` element.
- You can set permissions that apply to the entire provider, or to certain tables, or even to certain records, or all three.

You define permissions for your provider with one or more `<permission>` elements in your manifest file. * To make the permission unique to your provider, use Java-style scoping for the android:name attribute. * For example, name the read permission `com.example.app.provider.permission.READ_PROVIDER`.

You can specify several types of provider permissions, with more fine-grained permissions taking precedence over ones with larger scope:

Single read-write provider-level permission
: One permission that controls both read and write access to the entire provider, specified with the `android:permission` attribute of the `<provider>` element.

Separate read and write provider-level permission
: A read permission and a write permission for the entire provider.

  - You specify them with the `android:readPermission` and `android:writePermission` attributes of the `<provider>` element.
  - They take precedence over the permission required by `android:permission`.

Path-level permission
: Read, write, or read/write permission for a content URI in your provider.

  - You specify each URI you want to control with a `<path-permission>` child element of the `<provider>` element.
  - For each content URI you specify, you can specify a read/write permission, a read permission, or a write permission, or all three.
  - The read and write permissions take precedence over the read/write permission.
  - Also, path-level permission takes precedence over provider-level permissions.

5.17 Implementing the `ContentProvider` Class

The abstract class `ContentProvider` defines six abstract methods that you must implement as part of your own concrete subclass. All of these methods except `onCreate()` are called by a client application that is attempting to access your content provider:

`onCreate()`
: Initialize your provider. The Android system calls this method immediately after it creates your provider. Note that the system doesn't create your provider until a client tries to access it through a `ContentResolver`.

`query()`
: Retrieve data from your provider. Use the arguments to select the table to query, the rows and columns to return, and the sort order of the result. Return the data as a `Cursor` object.

`insert()`
: Insert a new row into your provider. Use the arguments to select the destination table and to get the column values to use. Return a content URI for the newly-inserted row.

`update()`
: Update existing rows in your provider. Use the arguments to select the table and rows to update and to get the updated column values. Return the number of rows updated.

`delete()`
: Delete rows from your provider. Use the arguments to select the table and the rows to delete. Return the number of rows deleted.

`getType()`
: Return the MIME type corresponding to a content URI.

5.18 Content Provider Implementation Notes

All of these methods except `onCreate()` can be called by multiple threads at once, so they must be thread-safe.

- When a client in a different application invokes a provider method through a `ContentResolver` instance, the system automatically selects a thread from an interprocess thread pool to execute the method in the provider.

- If a client component in the same process as the content provider invokes one of these methods through a `ContentResolver` instance, the system invokes the provider method in the **same thread** as the client component method call.

Avoid doing lengthy operations in `onCreate()`.

- The system automatically invokes `onCreate()` in the application process's main thread when in needs to initialize the provider.
- Therefore, defer initialization tasks until they are actually needed to avoid blocking your application's main thread.

Although you must implement these methods, your code does not have to do anything except return the expected data type.

- For example, you may want to prevent other applications from inserting data into some tables. To do this, you can ignore the call to `insert()` and return `null`.

5.19 Implementing the `onCreate()` method

The Android system calls `onCreate()` when it starts up the provider.

- You should perform only fast-running initialization tasks in this method, and defer database creation and data loading until the provider actually receives a request for the data.
- If you do lengthy tasks in `onCreate()`, you block your application's main thread, potentially causing an Application Not Responding (ANR) condition if your application contains other components.

For example, if you are using an SQLite database:

- You can create a new `SQLiteOpenHelper` object in `ContentProvider.onCreate()`.
- You can place the calls to `getWritableDatabase()` or `getReadableDatabase()` in the implementation of the provider's data access methods
- As a result, any actual database creation or upgrade is deferred until the first data access, and that creation or upgrade takes place in the interprocess communication thread selected to process the call.

Assuming that you are using a `SQLiteOpenHelper` subclass, the following could be sufficient to initialize your provider:

```
public boolean onCreate() {
        Context context = getContext();
        dbHelper = new TimelineHelper(context);
        return (dbHelper == null) ? false : true;
}
```

5.20 Implementing the `query()` Method

The `ContentProvider.query()` method must return a `Cursor` object, or if it fails, throw an `Exception`.

- If you are using an SQLite database as your data storage, you can simply return the `Cursor` returned by one of the `query()` methods of the `SQLiteDatabase` class.
- If the query does not match any rows, return a `Cursor` instance whose `getCount()` method returns `0`.
- Return `null` only if an internal error occurred during the query process.

If you aren't using an SQLite database as your data storage, use one of the concrete subclasses of `Cursor`.

- For example, the MatrixCursor class implements a cursor in which each row is an array of Object. With this class, use addRow() to add a new row.

Remember that the Android system must be able to communicate the Exception across process boundaries. Android can do this for the following exceptions that may be useful in handling query errors:

+ IllegalArgumentException:: You might choose to throw this if your provider receives an invalid content URI UnsupportedO
You might choose to throw this is you decide not to support a method, such as delete() NullPointerException:: This type of runtime exception will be reflected to the client if not caught and handled in the provider

5.21 The SQLiteQueryBuilder Helper Class

Implementing the query() method can be particularly complex.

- The method supports several arguments, almost all of which are optional.
- You often need to parse the Uri argument to include path components such as the ID in the query.
- You might have default query parameters to use if the client doesn't specify them.
- You might have additional query parameters "hard-coded" into your provider implementation that you need to merge with arguments provided by the client.
- You might need to perform joins or other manipulation of the underlying data to present a synthesized view to the client.

To simplify the creation of a complex query, Android provides the SQLiteQueryBuilder class. It supports features such as:

- Assembling query parameters incrementally
- Merging multiple query parameters, such as several portion of a WHERE clause
- Performing joins
- Performing *projection mapping*, to map column names that the caller passes into query to database column names

The example implementation of a query() shown next provides an example of using SQLiteQueryBuilder.

5.22 Example of Implementing a query() Method

```
public Cursor query(Uri uri, String[] projection,
                                        String selection, String[] selectionArgs,
                                        String sort) {

        SQLiteDatabase db = dbHelper.getWritableDatabase();

        // A convenience class to help build the query
        SQLiteQueryBuilder qb = new SQLiteQueryBuilder();

        qb.setTables(T_TIMELINE);

        switch (uriMatcher.match(uri)) {
        case STATUS_DIR:
                break;
        case STATUS_ITEM:
                // If this is a request for an individual status, limit the result set to ↩
                    that ID
                qb.appendWhere(StatusContract.Columns._ID + "=" + uri.getLastPathSegment()) ↩
                    ;
                break;
        default:
                throw new IllegalArgumentException("Unsupported URI: " + uri);
        }

        // Use our default sort order if none was specified
        String orderBy = TextUtils.isEmpty(sort)
                                ? StatusContract.Columns.DEFAULT_SORT_ORDER
                                : sort;

        // Query the underlying database
        Cursor c = qb.query(db, projection, selection, selectionArgs, null, null, orderBy);

        // Notify the context's ContentResolver if the cursor result set changes
        c.setNotificationUri(getContext().getContentResolver(), uri);

        // Return the cursor to the result set
        return c;
}
```

5.23 Implementing the `insert()` Method

The `insert()` method adds a new row to the appropriate table, using the values in the `ContentValues` argument.

- If a particular column name is not provide in the `ContentValues` argument, you might decide to provide a default value for it either in your provider code or in your database schema.

The `insert()` method should return the content URI for the new row.

- To construct this, append the new row's `_ID` (or other primary key) value to the table's content URI, using `withAppendedId()`.
- You can return `null` if the insertion fails, or if you decide not to support this operation.

5.24 Example of Implementing an `insert()` Method

```
public Uri insert(Uri uri, ContentValues initialValues) {
        Uri result = null;

        // Validate the requested Uri
    if (uriMatcher.match(uri) != STATUS_DIR) {
        throw new IllegalArgumentException("Unsupported URI: " + uri);
    }

    SQLiteDatabase db = dbHelper.getWritableDatabase();
    long rowID = db.insert(T_TIMELINE, null, initialValues);

    if (rowID > 0) {
        // Return a URI to the newly created row on success
        result = ContentUris.withAppendedId(StatusContract.CONTENT_URI, rowID);

        // Notify the Context's ContentResolver of the change
        getContext().getContentResolver().notifyChange(result, null);
    }
    return result;
}
```

5.25 Implementing the `delete()` method

The `delete()` method should delete rows from your provider.

- Use the selection arguments to select the table and the rows to delete.
- Return the number of rows deleted.
- You might decide always to return `0` if you choose not to support this operation, or throw an `UnsupportedOperationExceptior`

```
public int delete(Uri uri, String where, String[] whereArgs) {
    SQLiteDatabase db = dbHelper.getWritableDatabase();
        int count;

        switch (uriMatcher.match(uri)) {
        case STATUS_DIR:
                count = db.delete(T_TIMELINE, where, whereArgs);
                break;
        case STATUS_ITEM:
                String segment = uri.getLastPathSegment();
                String whereClause = StatusContract.Columns._ID + "=" + segment
                                                   + (!TextUtils.isEmpty(where) ? " AND ("  ↩
                                                     + where + ')' : "");
                count = db.delete(T_TIMELINE, whereClause, whereArgs);
                break;
        default:
                throw new IllegalArgumentException("Unsupported URI: " + uri);
        }

        if (count > 0) {
        // Notify the Context's ContentResolver of the change
        getContext().getContentResolver().notifyChange(uri, null);
        }
        return count;
}
```

5.26 Implementing the `update()` method

The `update()` method updates existing rows in your provider.

- It accepts the same `ContentValues` argument used by `insert()`, and the same selection arguments used by `delete()` and `query()`.
- This might allow you to re-use code between these methods.
- Return the number of rows deleted.
- You might decide always to return `0` if you choose not to support this operation, or throw an `UnsupportedOperationExceptior`

```
public int update(Uri uri, ContentValues values, String where, String[] whereArgs) {
    SQLiteDatabase db = dbHelper.getWritableDatabase();
        int count;

        switch (uriMatcher.match(uri)) {
        case STATUS_DIR:
                count = db.update(T_TIMELINE, values, where, whereArgs);
                break;
        case STATUS_ITEM:
                String segment = uri.getLastPathSegment();
                String whereClause = StatusContract.Columns._ID + "=" + segment
                                                      + (!TextUtils.isEmpty(where) ? " AND (" ↩
                                                          + where + ')' : "");
                count = db.update(T_TIMELINE, values, whereClause, whereArgs);
                break;
        default:
                throw new IllegalArgumentException("Unsupported URI: " + uri);
        }

        if (count > 0) {
        // Notify the Context's ContentResolver of the change
        getContext().getContentResolver().notifyChange(uri, null);
        }
        return count;
}
```

5.26.1 Registering the Provider in the Application Manifest

Your content provider must be registered in your application's manifest by including a `<provider>` element within the `<application>` element.

`android:name`
Required. The class name implementing your provider

`android:authorities`
Required. The URI authorities identifying your provider. Typically, you provide only one, but you can provide a semicolon-separated list of authorities.

`android:permission`
Optional. The name of a permission that clients must have to read or write the content provider's data. If provided, `android:readPermission` or `android:writePermission` take precedence over this attribute.

`android:readPermission`
Optional. The name of a permission that clients must have to read the content provider's data.

`android:writePermission`
Optional. The name of a permission that clients must have to write the content provider's data.

For example:

```
<provider
        android:name="StatusProvider"
        android:authorities="com.marakana.android.yamba.provider"
        android:writePermission="com.marakana.android.yamba.permission.MODIFY_STATUS_DATA"
        />
```

6 Loaders

6.1 What's a Loader?

Loaders make it easy to load data asynchronously 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.

Loaders were introduced in Honeycomb (API 11).

- The Android Support Package includes support for loaders. By including the support package in your application, you can use loaders even if your application for a `minSdkVersion` of 4 or later.

6.2 The Loader Classes and Interfaces

There are several classes and interfaces that implement the loader functionality:

`LoaderManager`
An abstract class associated with an `Activity` or `Fragment` for managing one or more `Loader` instances. There is only one `LoaderManager` per activity or fragment. But a `LoaderManager` can have multiple loaders.

`LoaderManager.LoaderCallbacks`
A callback interface for a client to interact with the `LoaderManager`.

`Loader`
An abstract class that performs asynchronous loading of data.

`AsyncTaskLoader`
Abstract loader class that provides an `AsyncTask` to do the work.

`CursorLoader`
A concrete subclass of `AsyncTaskLoader` that queries the `ContentResolver` and returns a `Cursor`. This class implements the `Loader` protocol in a standard way for querying cursors, building on `AsyncTaskLoader` to perform the cursor query on a background thread so that it does not block the application's UI.

Note
As of API 15, the only concrete `Loader` implementation is the `CursorLoader`, which can query only a content provider; it cannot run a query directly against a SQLite database.

6.3 Using Loaders in an Application

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.
- A data source, such as a `ContentProvider`, when using a `CursorLoader`.
- An implementation for `LoaderManager.LoaderCallbacks`. This is where you create new loader instances and manage your references to existing loaders.
- A way of displaying the loader's data, such as a `SimpleCursorAdapter`.

6.4 Accessing the `LoaderManager`

To use loaders in an activity or fragment, you need an instance of `LoaderManager`.

- There is only one `LoaderManager` per activity or fragment.

If you're using the standard APIs, invoke `getLoaderManager()` as provided in the `Activity` and `Fragment` classes.

If you're using the Support Package, you must use `android.support.v4.app.FragmentActivity` as the base class for your activity.

- Invoke `FragmentActivity.getSupportLoaderManager()` to obtain a `LoaderManager` for the activity.
- Invoke `android.support.v4.app.Fragment.getLoaderManager()` to obtain a `LoaderManager` for a fragment.

6.5 Starting a Loader

Invoke `FragmentManager.initLoader()` to initialize a specified `Loader`:

```
// Prepare the loader.  Either re-connect with an existing one,
// or start a new one.
getLoaderManager().initLoader(0, null, this);
```

The `initLoader()` method takes the following parameters:

- A unique integer ID that identifies the loader. In this example, the ID is `0`.
- Optional arguments in the form of a `Bundle` to supply to the loader at construction (`null` in this example).
- A `LoaderManager.LoaderCallbacks` implementation, which the `LoaderManager` calls to report loader events. In this example, the local class implements the `LoaderManager.LoaderCallbacks` interface, so it passes a reference to itself, `this`.

The `initLoader()` call ensures that a loader is initialized and active. It has two possible outcomes:

- If the loader specified by the ID already exists, the last created loader is reused.
- If the loader specified by the ID does not exist, `initLoader()` triggers the `LoaderManager.LoaderCallbacks` method `onCreateLoader()`. This is where you implement the code to instantiate and return a new loader.

You typically initialize a `Loader` within an activity's `onCreate()` method, or within a fragment's `onActivityCreated()` method.

6.6 Restarting a Loader

If you want to discard the the old data returned by a `Loader` and have it load fresh data, invoke the `FragmentManager.resetLoad`
method.

```
// Refresh the loader with new data
getLoaderManager().resetLoader(0, null, this);
```

The `resetLoader()` method accepts the same arguments as `initLoader()`:

- The integer ID of a `Loader`
- Optional arguments in the form of a `Bundle`
- A `LoaderManager.LoaderCallbacks` implementation

6.7 Destroying a Loader

If you want to destroy a `Loader` and have it discard the data that it loaded, invoke the `FragmentManager.destroy()` method.

```
// Destroy a loader and release its data
getLoaderManager().destroyLoader(0);
```

The `resetLoader()` method accepts only the integer ID of a `Loader` to destroy.

6.8 The `LoaderManager.LoaderCallbacks` Listener

The `LoaderManager.LoaderCallbacks` listener must implement the following interface:

```
public interface LoaderCallbacks<DataType> {
        public Loader<DataType> onCreateLoader(int id, Bundle args);
        public void onLoadFinished(Loader<DataType> loader, DataType data);
        public void onLoaderReset(Loader<DataType> loader);
}
```

`onCreateLoader()`
The `LoaderManager` invokes this method if it needs to create the `Loader` with the specified `id`. The `LoaderManager` also passes along the `Bundle` argument it received in the `LoaderManager.initLoader()` method.

The listener must instantiate the specified loader and return a reference to it. The `LoaderManager` then manages interaction with the `Loader` as required.

`onLoadFinished()`
The `LoaderManager` invokes this method when the loader has finished loading the requested data. It provides a reference to the `data` and the `loader` that generated it.

The listener should then use the data in whatever way the fragment or activity requires. For example, if a `CursorLoader` provides a `Cursor` as a result of a query, the `onLoadFinished()` method might hook up the `Cursor` to a `CursorAdapter` to display the data.

`onLoaderReset()`
The `LoaderManager` invokes this method when a loader is being reset or destroyed. It indicates that the data provided by the loader is becoming unavailable. At this point, the listener should remove any references it has to the loader's data.

Important
The loader "owns" the data is provides. The listener should not attempt to release or delete the data. For example, a `CursorLoader` provides a `Cursor`, the consumer of the `Cursor` should not attempt to close it; it should simply release its reference to the `Cursor` in response to the `onLoaderReset()` method call.

6.9 The `CursorLoader`

The `CursorLoader` is a concrete loader implementation that queries the `ContentResolver` and returns a `Cursor`.

- Typically, the only interaction your listener implementation needs to have with a `CursorLoader` is to instantiate it in response to an `onCreateLoader()` call.
- The `CursorLoader` constructor take a `Context` followed by the same arguments as `ContentResolver.query()`:

  `uri`
  The URI, using the `content://` scheme, for the content to retrieve.

projection
A list of which columns to return. Passing `null` returns all columns, which is inefficient.

selection
A filter declaring which rows to return, formatted as an SQL `WHERE` clause (excluding the `WHERE` itself). Passing `null` returns all rows for the given URI.

`selectionArgs`: You may include `?`s in `selection`, which will be replaced by the values from `selectionArgs`, in the order that they appear in the selection.

sortOrder
How to order the rows, formatted as an SQL `ORDER BY` clause (excluding the `ORDER BY` itself). Passing `null` use the default sort order.

6.10 A Simple Example

```
public static class CursorLoaderListFragment extends ListFragment
        implements MenuItem.OnMenuItemClickListener,
                LoaderManager.LoaderCallbacks<Cursor> {

    // This is the Adapter being used to display the list's data.
    SimpleCursorAdapter mAdapter;

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

        // Give some text to display if there is no data.  In a real
        // application this would come from a resource.
        setEmptyText("No phone numbers");

        // Create an empty adapter we will use to display the loaded data.
        mAdapter = new SimpleCursorAdapter(getActivity(),
                android.R.layout.simple_list_item_2, null,
                new String[] { Contacts.DISPLAY_NAME, Contacts.CONTACT_STATUS },
                new int[] { android.R.id.text1, android.R.id.text2 }, 0);
        setListAdapter(mAdapter);

        // Prepare the loader.  Either re-connect with an existing one,
        // or start a new one.
        getLoaderManager().initLoader(0, null, this);
    }

    @Override public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
        // Place an action bar item for searching.
        MenuItem item = menu.add("Refresh");
        item.setIcon(android.R.drawable.ic_menu_rotate);
        item.setShowAsAction(MenuItem.SHOW_AS_ACTION_IF_ROOM);
                item.setOnMenuItemClickListener(this);
    }

    public boolean onMenuItemClick(String newText) {
        // Called when the user clicks the refresh button.
        getLoaderManager().restartLoader(0, null, this);
        return true;
    }

    // These are the Contacts rows that we will retrieve.
    static final String[] CONTACTS_SUMMARY_PROJECTION = new String[] {
        Contacts._ID,
        Contacts.DISPLAY_NAME,
        Contacts.CONTACT_STATUS,
    };

    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        // This is called when a new Loader needs to be created.  This
        // sample only has one Loader, so we don’t care about the ID.
        Uri baseUri = Contacts.CONTENT_URI;

        // Now create and return a CursorLoader that will take care of
        // creating a Cursor for the data being displayed.
        String select = "((" + Contacts.DISPLAY_NAME + " NOTNULL) AND ("
                + Contacts.HAS_PHONE_NUMBER + "=1) AND ("
                + Contacts.DISPLAY_NAME + " != '' ))";
        return new CursorLoader(getActivity(), baseUri,
                CONTACTS_SUMMARY_PROJECTION, select, null,
                Contacts.DISPLAY_NAME + " COLLATE LOCALIZED ASC");
    }

    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
        // Swap the new cursor in.  (The framework will take care of closing the
        // old cursor once we return.)
        mAdapter.swapCursor(data);
    }

    public void onLoaderReset(Loader<Cursor> loader) {
        // This is called when the last Cursor provided to onLoadFinished()
        // above is about to be closed.  We need to make sure we are no
        // longer using it.
        mAdapter.swapCursor(null);
    }
}
```

7 Module Summary

You should now be able to:

- Write client code that can read and modify the data managed by a content provider
- Implement a basic content provider to expose structured data to other applications
- Use loaders to retrieve a `Cursor` from a content provider without blocking your application’s main thread