[How to Use Loaders in Android](http://www.grokkingandroid.com/using-loaders-in-android/" \o "How to Use Loaders in Android)

September 27th, 2012 By [Wolfram Rittmeyer](http://www.grokkingandroid.com/author/writtmeyer/) [63 Comments](http://www.grokkingandroid.com/using-loaders-in-android/#comments)

With the introduction of Honeycomb Loaders became the preferred way to access data of databases or content providers. They load data asynchronously and notify listeners when the results are ready.

Google did not only introduce Loaders but also deprecated the previous way to handle a Cursor within your activities. You shouldn’t use startManagingCursor() or managedQuery() in your projects anymore.

With managed cursors queries and requeries are executed on the UI thread. This could cause the app to feel unresponsive or to even display an ANR error message. With Loaders your queries will no longer run on the UI thread and your app remains responsive.

In this post I introduce the classes that form the Loader API and show you how to use them.

|  |  |
| --- | --- |
| Class | Usage |
| LoaderManager | Manages your Loaders for you. Responsible for dealing with the Activity or Fragment lifecycle |
| LoaderManager.LoaderCallbacks | A callback interface you must implement |
| Loader | The base class for all Loaders |
| AsyncTaskLoader | An implementation that uses an [AsyncTask](https://developer.android.com/reference/android/os/AsyncTask.html" \o "Documentation of Android's AsyncTask) to do its work |
| CursorLoader | A subclass of AsyncTaskLoader for accessing ContentProvider data |
| **The classes and interfaces of the Loader API** | |

In the following sections I describe most of these classes and what you need to know about them – starting with the [LoaderManager](http://developer.android.com/reference/android/app/LoaderManager.html" \o "Documentation of the LoaderManager class).

LoaderManager

This class keeps your Loaders in line with the lifecycle of your activities or fragments. If Android destroys your fragments or activities, the LoaderManager notifies the managed loaders to free up their resources. The LoaderManager is also responsible for retaining your data on configuration changes like a change of orientation and it calls the relevant callback methods when the data changes. In short: The LoaderManager is way more powerful than the old startManagingCursor() or managedQuery() methods.

You do not instantiate the LoaderManager yourself. Instead you simply call getLoaderManager() from within your activity or your fragment to get hold of it.

Most often you are only interested in two methods of the manager:

* initLoader() and
* restartLoader()

**initLoader()**

The initLoader() method adds a Loader to the LoaderManager:

getLoaderManager().initLoader(LIST\_ID, null, this);

It takes three arguments:

* a unique ID for this loader,
* an optional Bundle with arguments for your Loader and
* a [LoaderCallbacks](https://developer.android.com/reference/android/app/LoaderManager.LoaderCallbacks.html" \o "Documentation of the LoaderCallbacks interface) interface

You might need the ID for further method calls. So using a final static field for the ID makes your code more readable. The Bundle can be used to pass additional arguments to your Loader, but isn’t used by the CursorLoader. The third argument, the callback interface, will be covered in detail later on.

The initLoader() method creates a new Loader only if for this ID none has been created previously. Keep in mind that Android deals with configuration changes for you, thus a simple change in orientation is enough to trigger a new call to initLoader(). In this case the method returns the existing instance and your query is not executed again.

**restartLoader()**

Because Android doesn’t execute the query again, you need a way to re-initialize the Loader when data, that is used to build the query, changes. Typical examples are search queries.

You reset your Loader by using the restartLoader() method. It takes the same parameters as initLoader(). Of course you have to use the same ID you used for initializing.

getLoaderManager().restartLoader(LIST\_ID, null, this);

LoaderManager.LoaderCallbacks

The interface LoaderCallbacks defines methods you must implement to create your Loader, to deal with the results and to clean up resources.

Since the interface is parameterized you must specify the type of data your Loader holds. Most often the type will be Cursor:

public class YourFragment extends Fragment

implements LoaderCallbacks<Cursor> {

//…

}

The methods you have to implement are:

* onCreateLoader(),
* onLoadFinished() and
* onLoadReset()

In the next sections I show what to do in each of these callback methods.

**onCreateLoader()**

The LoaderManager calls this method when you call initLoader() for the first time. As mentioned, the manager only calls this method if no loader for the given ID exists.

The method gets an int value and a Bundle passed in. These are the same values you used for your initLoader() call.

A typical example creating a CursorLoader looks like this:

public Loader<Cursor> onCreateLoader(int id, Bundle args) {

CursorLoader loader = new CursorLoader(

this.getActivity(),

SOME\_CONTENT\_URI,

projection,

selection,

selectionArgs,

sortOrder);

return loader;

}

As you can see the parameters are a Context object plus those of the ContentResolver’s query() method. If you’re not familiar with these arguments, I recommend you read my [post about accessing content providers](http://www.grokkingandroid.com/android-tutorial-using-content-providers/).

If you need to track multiple queries and thus use different IDs for your Loaders, all you need to add is a simple case- or if-else-branch.

**onLoadFinished()**

This method is the most interesting one. Here you update the UI based on the results of your query.

For ListAdapters you simply swap the cursor of the adapter as described in the section “[Changes needed for CursorAdapters](http://www.grokkingandroid.com/using-loaders-in-android/#how_to_deal_with_cursoradapters)“.

For all other cases you have to get references to the view elements and set their value according to the result.

This is how it looks in the sample project:

public void onLoadFinished(

Loader<Cursor> loader,

Cursor cursor) {

if (cursor != null && cursor.getCount() > 0) {

cursor.moveToFirst();

int idIndex =

cursor.getColumnIndex(LentItems.\_ID);

int nameIndex =

cursor.getColumnIndex(LentItems.NAME);

int borrowerIndex =

cursor.getColumnIndex(LentItems.BORROWER);

this.itemId = cursor.getLong(idIndex);

String name = cursor.getString(nameIndex);

String borrower = cursor.getString(borrowerIndex);

((EditText)findViewById(R.id.name)).

setText(name);

((EditText)findViewById(R.id.person)).

setText(borrower);

}

}

**onLoadReset()**

This method allows you to release any resources you hold, so that the Loader can free them. You can set any references to the cursor object you hold to null.  
**But do not close the cursor** – the Loader does this for you.

See also the section about [how to deal with CursorAdapters](http://www.grokkingandroid.com/using-loaders-in-android/#how_to_deal_with_cursoradapters).

Loader, AsyncTaskLoader and CursorLoader

The Loader interface and its implementations are not very interesting – unless you write your own custom Loaders. You have to create a Loader of course. But other than using the constructor of CursorLoader, you normally do not interact with these objects yourself.

If you use multiple Loaders you need to access the ID in the callback methods. You can do so by calling getId() on the loader passed in to the callbacks.

If you want to write a custom Loader yourself, please have a look at Alex Lockwood’s [tutorial on implementing loaders](http://www.androiddesignpatterns.com/2012/08/implementing-loaders.html).

Changes needed for CursorAdapters

An important use of cursors in Android is to use a CursorAdapter as data source for ListViews, AutoCompleteTextViews and so on. When working with Loaders you have to adapt the old way slightly.

First of all: You do not have a Cursor object before the onLoadFinished() method of your callback has been called. In other words: The cursor is not ready when you create the adapter. Thus you create the adapter using null for the cursor argument:

SimpleCursorAdapter adapter =

new SimpleCursorAdapter(

getApplicationContext(),

android.R.layout.simple\_list\_item\_1,

null,

columns,

layoutIds,

0);

When the cursor is finally available you have to add it. You do this by calling swapCursor() on your adapter and passing in the cursor object of your callback method:

public void onLoadFinished(

Loader<Cursor> loader,

Cursor cursor) {

((SimpleCursorAdapter)this.getListAdapter()).

swapCursor(cursor);

}

And to clean up resources in the onLoadReset() method you also use swapCursor(), this time passing in a null value:

public void onLoaderReset(Loader<Cursor> loader) {

((SimpleCursorAdapter)this.getListAdapter()).

swapCursor(null);

}

Use the Support Library for older Android versions

To support newer features on older Android versions, Google provides the [Support Library](http://developer.android.com/tools/extras/support-library.html). The two main components this library contains are Fragments and Loaders.

You can and should use this library for projects that must support older versions. All reasons for using the Loader-API are valid for pre-Honeycomb devices as well.

But you have to keep an eye on the import statements. All import statements for classes of the Support Library begin with android.support.v4:

import android.support.v4.app.LoaderManager;

import android.support.v4.app.LoaderManager.LoaderCallbacks;

import android.support.v4.content.CursorLoader;

import android.support.v4.content.Loader;

Don’t mix classes of the support library with normal classes. The compiler will spot most problems for you and then there is also [Lint](http://tools.android.com/tips/lint) to help you. But if you ever wonder why some code is marked as invalid, this most likely is the reason.

To use Loaders for older Android versions, you need to use activities or fragments of the Support Library. The normal ones do not have the getLoaderManager() method before Honeycomb. You project would still compile (if your build target is at least SDK 11) but at runtime you would get a NoSuchMethodError on older devices.

Using Loaders to access your SQLiteDatabase

Since Android’s CursorLoader is only for Cursors returned by content providers we need another Loader implementation if we want to use SQLite directly.

Thankfully [Mark Murphy](http://commonsware.com/blog/) has written a library that offers [enhancements to the Loader framework](https://github.com/commonsguy/cwac-loaderex). This library also contains a SQLiteCursorLoader.

To use Mark Murphy’s SQLiteCursorLoader you have to create an SQL select statement and pass it to the constructor.The constructor also expects an SQLiteOpenHelper object. An example of the onCreateLoader() method could look like this:

public Loader<Cursor> onCreateLoader(int id, Bundle args) {

String rawQuery = "SELECT …";

String[] queryParams = // to substitute placeholders

SQLiteCursorLoader loader =

new SQLiteCursorLoader(

getActivity().getApplicationContext(),

yourSqliteOpenHelper,

rawQuery,

queryParams);

return loader;

}

But there is more to it. You have to use this loader also for deleting, updating or inserting rows as well. Only by doing so, the loader knows about a data change and can call the correct callback methods.

I won’t go into these details here. See the [project’s github page](https://github.com/commonsguy/cwac-loaderex) or have a look at Mark Murphy’s sample project.

If you intend to use this library, don’t use the jar-file offered on the project page. It is outdated. Instead clone the git repository and add the project to your IDE directly.

When not to use Loaders

On [reddit](http://www.reddit.com/r/androiddev/comments/10jwcr/how_to_use_loaders_in_android/" \o "Reddits thread about this blog post), [cokacokacoh](http://www.reddit.com/user/cokacokacoh" \o "cokacokacoh's profile on reddit) noted that my post missed a section on when not to use Loaders. He was right of course. So I added the following paragraph to this post.

As cokacokacoh points out, you shouldn’t use Loaders if you need the background tasks to complete. Android destroys Loaders together with the Activities/Fragments they belong to. If you want to do some tasks, that have to run until completion, do not use Loaders. 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!

**How to Use Loaders in Android**

In this post, we’re going to learn about loaders as well as content providers. Loaders allow us to *load* vast amounts of data asynchronously so we don’t bog down the main UI thread. We can get this data from querying databases using content providers instead of having it stored in a resources file like we’ve done in other posts. We’ll be building a small application that will just list the names of all of the contacts on our phone.

If you’re not familiar with SQLite, I highly suggest that you look over [this](http://androidkennel.org/?p=1318) post on SQLite since content providers deal with accessing data from the system’s SQLite databases.

The source code for this post can be downloaded [here](https://zenva.com/fileExt/54307a654264576a3136324a394c4f42716747415532366d506c4569516435467570682f41554d437449495257304273483730676c612f42524432326b554b456e6d58466761786b344366785678565058356a4164687350675463627a5377714f444252784e38664d704b3755762f39346c4e2f595058457641524c6e7a426a3334394d2b655870347377667433766136534e455868546849596c7271673675654561464f4479784d7030336b78626977786e616b7768692b7054554f49576a63364241725a4742786d6f646a627a674579745a6d566f37697548477559504259426f332f685a78737354525649647179496a496c6370456f4f4d6a35523650).

We’ve looked before at how to [load data into a ListView using adapters](http://androidkennel.org/?p=1281), but what if we have a very large database or data set? Most of the data sets that we’ve been using have been pretty small so it doesn’t hurt that much to load them immediately in onCreate(...) . However, when we have databases or data sets of thousands or hundreds of thousands of entries, instead of stalling the app, it might be better to load them on a background thread. We’ve already looked at how we can [create and manage background threads](http://androidkennel.org/?p=1402), but since loading data from a database in a background thread is such a common task, Android has a wonderful CursorLoader class that prevents us from having to write the boilerplate code ourselves.

But where do we get the data for the loaders to load? Let me talk about cursors and content providers. Let’s look at an example of one: the Contacts app. On Android phones, there’s a way to manage all of our contacts. But where is that data stored? It’s stored in a database! In fact, the system can even let us access into that database through a content provider. Content providers allow applications to share data between each other. This is how Twitter or LinkedIn know about your contacts: they have access to them if you give the permission for them to read your contacts.

The Android system has hundreds of content providers for many different pieces of information like contacts, media, and photos. To differentiate all of these different content providers, each group of related information has a content URI that we provide to tell the content provider that we’re querying one database and not the other. Think of URIs as being URLs on the internet. Inputting androidkennel.org will get you to this site, but inputting google.com will get you to another site. These URIs are also unique so no two content providers should have the same URI. We simply provide the URI and the query we want to run, although most of the querying part is actually abstracted away for good reasons.

When we query a content provider, we get back a Cursor object. This allows us to step through the rows of the result query using a similar approach as if we had an iterator: while the Cursor has a next row, grab the current one and then move on to the next row. We can access the data stored in the current row of the Cursor through the index of the column, but we won’t have to do this since Android provides us with a SimpleCursorAdapter that will display cursor results for us, allowing us to skip the step of making our own custom adapter.

To summarize our approach to display contacts in a ListView, we’ll first need to ask for permission to read contacts. Then we’ll initialize the loader with the appropriate query (that is, retrieve all of the contacts’ names). Then, when the query is finished, we switch our adapter’s Cursor with the one we get back from the loader.

Let’s get started and create the LoaderDemo project! We’ll need Android 6.0 Marshmallow with an empty activity. First things first, we need a ListView! Replace the layout/activity\_main.xml with the following.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <?xml version="1.0" encoding="utf-8"?>  <ListView xmlns:android="http://schemas.android.com/apk/res/android"      xmlns:tools="http://schemas.android.com/tools"      android:id="@+id/listView"      android:layout\_width="match\_parent"      android:layout\_height="match\_parent"      tools:context="com.deshpande.loaderdemo.MainActivity" /> |

Now that we have a ListView, we need to deal with the runtime permissions for reading contacts. Add the following line in the AndroidManifest.xml file before the application tag.



|  |  |
| --- | --- |
| 1 | <uses-permission-sdk-23 android:name="android.permission.READ\_CONTACTS" /> |

Now that we’ve done this, we can request the READ\_CONTACTS permission right when we start the app. Open up MainActivity and add the following to the onCreate(...)  method. We’ll also need to create a constant at the top of our class for PERMISSION\_CONTACTS.



|  |  |
| --- | --- |
| 1  2  3  4  5 | if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ\_CONTACTS) != PackageManager.PERMISSION\_GRANTED) {      ActivityCompat.requestPermissions(this, new String[] { Manifest.permission.READ\_CONTACTS }, PERMISSION\_CONTACTS);  } else {      getLoaderManager().initLoader(LOADER\_CONTACTS, null, this);  } |

Now after we’ve been granted permission, we can initialize the loader. We’ll need another constant for LOADER\_CONTACTS since we could have initialized multiple loaders if we wanted to.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Override  public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {      switch (requestCode) {          case PERMISSION\_CONTACTS:              if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION\_GRANTED) {                  getLoaderManager().initLoader(LOADER\_CONTACTS, null, this);              }              break;      }  } |

The first parameter is the ID of the loader, the next is an optional Bundle we could pass, and the final argument is the callback listener. This will make us implement the callbacks, but change the declaration so that instead of the parameter type being Object, it’s Cursor. We’ll have to change this in the methods as well.



|  |  |
| --- | --- |
| 1 | public class MainActivity extends AppCompatActivity implements LoaderManager.LoaderCallbacks<Cursor> { ... } |

Before we do that, we’ll need to create a SimpleCursorAdapter and set our ListView’s adapter to be that SimpleCursorAdapter. We’ll need to create a private field in our class for the adapter and then initialize it inonCreate(...)  since we’ll need the adapter later. The first parameter is the context, the second is the layout, and the third is the Cursor. We’re passing in a simple ListView item layout that’s really just a TextView whose ID is text1.

We’re initially passing null since we’ll give it a Cursor after the loader is finished. The next two fields are the mapping between database column and the contents of a view ID. In our case, we want to map the name to the text1 TextView. To reiterate, text1 is the ID of the single TextView in the simple list item layout. Android is smart enough to know that the ID we’re passing is that of a TextView so we set it’s text to be whatever is in the row’s DISPLAY\_NAME column. The final parameter is just some arbitrary flags that we won’t set since we don’t have any flags to set.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | String[] from = { ContactsContract.Contacts.DISPLAY\_NAME };  int[] to = { android.R.id.text1 };  adapter = new SimpleCursorAdapter(this, android.R.layout.simple\_list\_item\_1, null, from, to, 0);    ListView listView = (ListView) findViewById(R.id.listView);  listView.setAdapter(adapter); |

Now that we’ve configured our adapter, we can actually look at how we can implement the loader. First we need to create the loader. We’ll be creating a new CursorLoader that will handling loading up the data we need. The first parameter is the context, the next is the URI that I mentioned before. The third parameter is called the projection and it’s the columns that we want to return. We declared this as a constant at the top of the class. We need to make sure to **always** have the unique \_ID value in the projection. Then we need the DISPLAY\_NAME. The next two parameters are the selection clause and the selection arguments. We’re passing in null so we get all of the contacts (null translates to \* in SQLite) and we don’t have any selection parameters, of course. The final parameter is the sorting order and we don’t care, but we could have our ListView sorted if we needed to.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | private static final String[] PROJECTION = { ContactsContract.Contacts.\_ID, ContactsContract.Contacts.DISPLAY\_NAME };    ...    @Override  public Loader<Cursor> onCreateLoader(int id, Bundle bundle) {      if (id == LOADER\_CONTACTS) {          return new CursorLoader(this, ContactsContract.Contacts.CONTENT\_URI, PROJECTION, null, null, null);      } else {          return null;      }  } |

Now the loader will go and execute the query on the background thread and call onLoadFinished(...)  when it’s done with the query. Note that we get the original loader and an object of the parameterized type, a Cursor in our case. This part of the code is simple, we just swap the null Cursor with the one that the loader returned.



|  |  |
| --- | --- |
| 1  2  3  4  5 | @Override  public void onLoadFinished(Loader<Cursor> loader, Cursor c) {      adapter.swapCursor(c);      adapter.notifyDataSetChanged();  } |

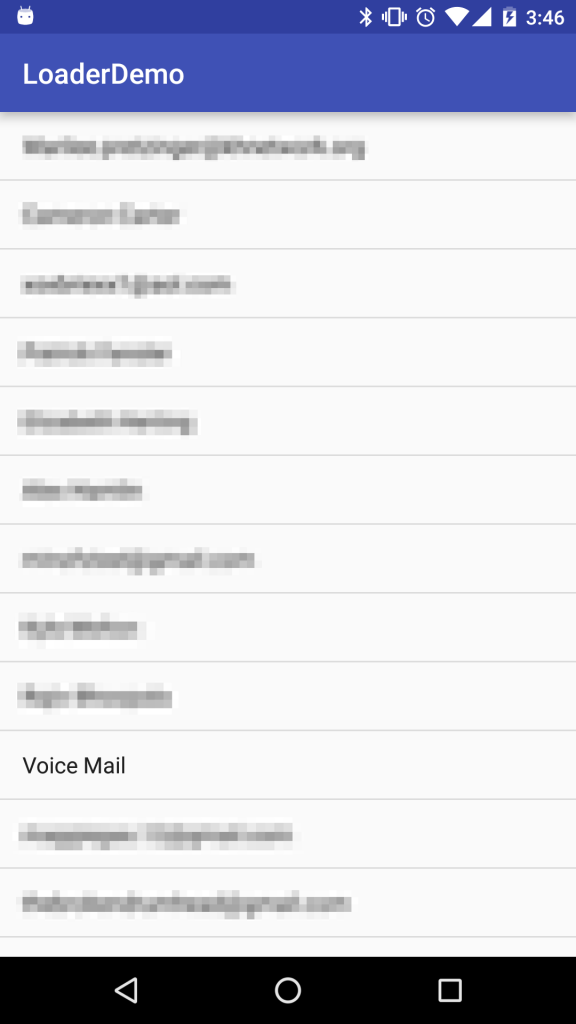
You might have noticed that we have a method called swapCursor(...)  and one called changeCursor(...) . These two are very different by one key fact:changeCursor(...)  will **close** the old Cursor. After we’re done getting data from a Cursor, it’s important that we call close()  on it to free up resources. We haven’t had to do it since SimpleCursorAdapter and the CursorLoader manage it for us. This is also the reason why it’s very important that we use swapCursor(...)  instead of changeCursor(...) . CursorLoader is managing the Cursor so it will close it after the call toonLoadFinished(...) . If we close it and it’s already closed, this will crash our app! This is why we **must** use swapCursor(...)  instead ofchangeCursor(...) .

onLoaderReset(...)  is just a mandatory method we have to implement so we’ll swap the cursor of our adapter to be null since this means that our created loader is being reset, so we don’t want to be accessing data at this time.



|  |  |
| --- | --- |
| 1  2  3  4 | @Override  public void onLoaderReset(Loader<Cursor> loader) {      adapter.swapCursor(null);  } |

That’s all we need to do with the loaders! Now we can run the app in our emulator or on our phone and get a list of all of our contacts! In the emulator, you can sign into your Google account to grab those contacts. Here’s an example of the app running on my phone (obviously my contacts are censored).

[](https://androidkennel.org/wp-content/uploads/2015/11/Loaders-%E2%80%93-1.png?x65546)

**Conclusion**

In this post, we learned how to use content providers to access our phone’s contacts and load them into a ListView asynchronously using loaders. The great benefit of using loaders is that they’re asynchronous so we don’t hang the UI thread while it waits for us to grab potentially thousands of entries. We also learned more about Cursors and how they’re related to content providers as well. Use loaders whenever you predict that you’re going to be fetching a potentially vast amount of data (maybe even showing a spinning progress dialog while the user waits).