Programming with Android: Data Management

Federico Montori

Dipartimento di Informatica: Scienza e Ingegneria

Università di Bologna

Data: outline

- Data Management in Android
  - Preferences (key-value pairs)
  - Text Files
  - XML Files
  - SQLite Database
  - Content Provider

Managing Data

Preferences: Key/Value pairs of data

Direct File I/O: Read/write files onboard or on SD cards. Remember to request permission for writing, for instance, on SD card

Database Tables: SQL Lite

Application Direct Access: Read only access from res assets/raw directories

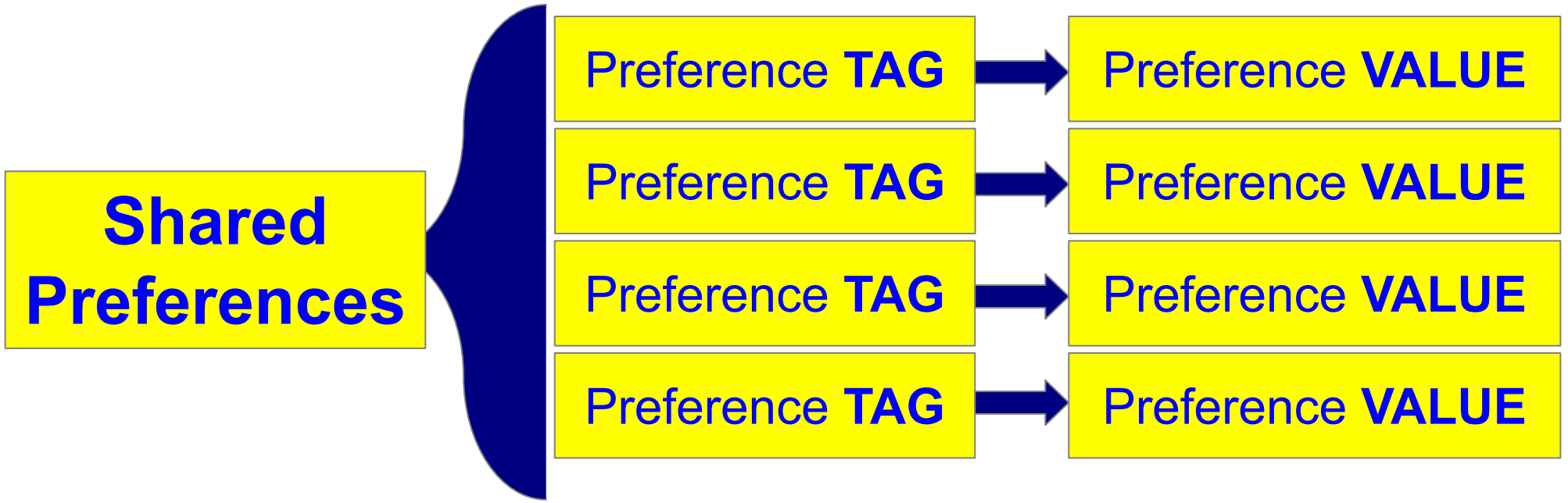
Increase functionality:

Content Providers: expose data to other applications

Services: background processes that run detached from any view

SharedPreferences system

- SharedPreferences are a convenient way to store configuration parameters **on the disk**
- Structured with a key-value mode



SharedPreferences types

- SharedPreferences could be either private or public
  - Public means that other applications could potentially read such preferences
    - (only until Android 7, not that shared anymore...)
  - Private means that they could be restricted at
    - Application level
    - Activity level
- We can also set a Preference screen, by using the Settings API from Jetpack.

SharedPreferences types

- Up to Android 7.0 (excluded)

```
getSharedPreferences(String name, Context.MODE_WORLD_READABLE);
getSharedPreferences(String name, Context.MODE_WORLD_WRITABLE);
```

  - To share data among applications
- Starting from 7.0 it gives a Security Exception
  - To prevent, use FileProvider if you need to share with others.
- Correct method to get shared preferences

```
getSharedPreferences(String preference, Context.MODE_PRIVATE);
```

SharedPreferences types

- Correct method to get shared preferences

```
getSharedPreferences(String preference, Context.MODE_PRIVATE);
```

Name of the preference File

Callable from any **Context** when you need to identify the file by name

```
getDefaultSharedPreferences();
```

Call the default one for your app (for settings, more later...)

- Correct method to get unique shared preferences

```
getPreferences(Context.MODE_PRIVATE);
```

Callable from any Activity, to get the unique preference file assigned to it. No need for name.

SharedPreferences example

```
public void onCreate(Bundle savedInstanceState) {
    Super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    Context context = getActivity();
    SharedPreferences pref = context.getSharedPreferences
            (MY_TAG, Context.MODE_PRIVATE);
    String myData = pref.getString(MY_KEY, “No pref”);
    TextView myView = findViewById(R.id.myTextView);
    myView.setText(myData);
}
```

SharedPreferences editor

- How to edit preferences?
- You need to get a **SharedPreferences.Editor**
- Be sure to commit operations at the end

```
pref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = pref.edit();
editor.putString("myDataLabel", "myDataValue");
editor.commit();
```

Alternatively you could also call **apply**() instead of commit, which writes the data to the disk asynchronously. Calling **commit**() stops your main thread.

Preference screens

You can interact with the default SharedPreferences through preferences screen.

- Starting with Android 10, **android.preference** is deprecated.
- Use Androidx Preference Library (or Settings API) instead.

add: **implementation androidx.preference:preference:1.1.0**

- It comes with a built-in Material Design look and feel
- It uses the **res/xml** resource directory

Preference screens

- It has to be a **PreferenceScreen** and use some facilities
  - Use either basic **Preference** if you want no widget
  - Use a facility if you want widgets

```
<PreferenceScreen
      xmlns:app="http://schemas.android.com/apk/res-auto">
      <SwitchPreferenceCompat
          app:key="notifications"
          app:title="Enable message notifications"/>
      <Preference
          app:key="feedback"
          app:title="Send feedback"
          app:summary="Report technical issues or suggest new features"/>
  </PreferenceScreen>
```

Preference screens

- Some specializations to ease the process
  - CheckBoxPreference
  - EditTextPreference
  - ListPreference
  - RingtonePreference

Read more guidelines:
https://source.android.com/devices/tech/settings/settings-guidelines

Preference screens

To inflate the XML hierarchy on a screen, just extend the **PreferenceFragment** and display it as you would with any other Fragment.

- Remember, this works with the DefaultSharedPreferences

```
public class MySettingsFragment extends PreferenceFragmentCompat {
    @Override
    public void onCreatePreferences(Bundle savedInstanceState, String rootKey) {
        setPreferencesFromResource(R.xml.preferences, rootKey);
    }
}
```

Replaces preferences instead of adding to the current hierarchy

The Android FileSystem

- Linux architecture with quite limited user permission.
  - Onboard data (non-shareable)
    - Application-specific files (stuff in /res)
      - res/raw
      - res/xml
    - Application-specific storage on external support
  - External shareable files (different API management)

https://developer.android.com/training/data-storage/shared

Non-Shareable File I/O

- Onboard
  - Write to a designated place for each application
  - **Where?** /data/data/<package>/files
  - **How?** Use standard java I/O classes
  - Get a reference to the directory through *context.getFilesDir()*
- SD card
  - **Where?** ContextCompat.getExternalFilesDir()
  - **How?** Use standard java I/O classes
  - **Permissions?** android.permission.WRITE_EXTERNAL_STORAGE
    - Only for Android <4.4 (then anyone can access)
    - From Android 10 we have **Scoped Storage**

Raw Text Files: how?

- Raw Text File
  - Place it under res/raw/ directory
  - Fill it with the text you like
  - Cannot edit it
  - Get its content

```
InputStream file = getResources().openRawResource(R.raw.myfile);
```

XML Files: how?

- XML File
  - Place it under res/xml/ directory
  - Start the file with

    <?xml version="1.0" encoding="utf-8"?>
  - Add whatever you want with <mytag>value</mytag>

XML Files: example

- We want to visualize all the grades of this class
- Our XML file is a set of these structures:

```
<student
        name="Student's name"
        class="Laboratorio di Applicazioni Mobili"
        year="2020"
        grade="30L" />
```

XML Files: parsing code example

```
XmlResourceParser grades = getResources().getXml(R.xml.myxmlfile);
LinearLayout ll = findViewById(R.id.myLL); int tag = -1;
while (tag != XmlResourceParser.END_DOCUMENT) {
      if (tag == XmlResourceParser.START_TAG) {
            String name = grades.getName();
            if (name.equals("student")) {
                  TextView tv = new TextView(this);
                  LayoutParams lp = new LayoutParams(LayoutParams.MATCH_PARENT,
                        LayoutParams.WRAP_CONTENT);
                  tv.setLayoutParams(lp);
                  String toWrite = grades.getAttributeValue(null, "name") + …;
                  tv.setText(toWrite); ll.addView(tv);
            }     }
            try {       tag = grades.next();      } catch (Exception e) { }
      }
```

File I/O: Java recap

We're not covering complex File I/O in this course, but:

- Simple create new file (see later for constructor):

```
File file = new File(...);
file.createNewFile();
```

- Simple write to file:

```
FileOutputStream outputStream = openFileOutput("fileName.txt", Context.MODE_PRIVATE);
outputStream.write("Internal Content".getBytes());
outputStream.close();
```

File I/O: Java recap

We're not covering complex File I/O in this course, but:

- Simple read from file:

```
FileInputStream inputStream = context.openFileInput("filename.txt");
InputStreamReader inputStreamReader =
                    new InputStreamReader(inputStream, StandardCharsets.UTF_8);
StringBuilder stringBuilder = new StringBuilder();
BufferedReader reader = new BufferedReader(inputStreamReader)) {
    String line = reader.readLine();
    while (line != null) {
        stringBuilder.append(line).append('\n');
        line = reader.readLine();
    }
}
String contents = stringBuilder.toString();
```

Internal files

- Access files like this:

```
File file = new File(context.getFilesDir(), filename);
```

This file is internal to the app

- generated into
  - **data/data**/your.package.name.appname/**files**
- Inaccessible to others
- Can get the list

```
Array<String> files = context.fileList()
```

External app-specific files

- Access files like this:

```
File(Environment.getExternalFilesDir(
        Environment.DIRECTORY_PICTURES), albumName);
```

This file is in the external storage to the app

- generated into
  - **/storage/emulated/0/Android/data/**your.package.name.appname**/files/Pictures/albumName**
- Inaccessible to others (it's app specific) but it has a defined name
- Can get the status of the media
  - e.g. Environment.**MEDIA_MOUNTED**

```
String state = Environment.getExternalStorageState();
```

External shared files

- Access files like this (before Scoped Storage):

```
File(Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES), albumName);
```

This file is in the external storage to the app

- generated into
  - **/storage/emulated/0/Android/data/**your.package.name.appname**/files/Pictures/albumName**
- Accessible to others but needs **WRITE_EXTERNAL_STORAGE** permission till Android 9 (included)

Now with **Scoped Storage** there is a bunch of APIs to manage this...

https://developer.android.com/training/data-storage/shared

Shareable File I/O

- Media Content
  - Write to a designated place for each type (video, photo, audio)
  - **Where?** you need to use the **Mediastore** collection
  - **How?** obtain reference to elements as URI
  - Until Android 9 you need WRITE_EXTERNAL_STORAGE and READ_EXTERNAL_STORAGE
- Documents and Others
  - **Where?** Fire an intent that can open a System File Picker
  - **How?** obtain reference to elements as URI
  - **Permissions?** none (handled by the picker)

Sharing Data

If your APP wants to share data, it can do so with Intents

Specifically ACTION_SEND advertises that an app is sending data to something else

From Android 4.0 use menus (we will see)

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, "This is my text to send.");
sendIntent.setType("text/plain");
startActivity(Intent.createChooser(sendIntent,
        getResources().getText(R.string.send_to)));
```

SQLite

General purpose solution

- Lightweight database based on SQL

Standard SQL syntax

```
SELECT name FROM table WHERE name = “Federico”
```

Android gives a standard interface to SQL tables of other apps

For application tables no content providers are needed

Why a local database? **Since you can’t assume connectivity**

Single Source of Truth: ***SSOT refers to the concept where certain data has only one official source to be used by data consumers (i.e. humans and software) for the true current version of that data (more on that later).***

SQLite: how?

- A database to store information
- Useful for structured informations
- Create a DBHelper which extends SQLiteOpenHelper
- Fill it with methods for managing the database
  - Better to use constants like
    - TABLE_GRADES
    - COLUMN_NAME
    - ….

SQLite: example

- Our database will look like this:
  - grade table:
    - id: integer, primary key, auto increment
    - firstName: text, not null
    - lastName: text, not null
    - class: text, not null
    - grade: integer, not null

SQLite: better to use constants

- Useful for query definition
- Our constants?

```
private static final String DB_NAME = “grades.db”;
private static final int DB_VERSION = 1;
public static final String TABLE_GRADES = “grades”;
public static final String COL_ID = “id”;
public static final String COL_FIRSTNAME = “firstName”;
public static final String COL_LASTNAME = “lastName”;
public static final String COL_CLASS = “class”;
public static final String COL_GRADE = “grade”;
```

SQLite: even better to use contracts

- Static Classes for grouping constants
- Interface “BaseColumns” provides the field _ID required by CursorAdapter (see later)

```
public final class FeedReaderContract {
      // To prevent someone from accidentally instantiating the contract class,
      // make the constructor private.
      private FeedReaderContract() {}

      /* Inner class that defines the table contents */
      public static class FeedEntry implements BaseColumns {
          public static final String TABLE_NAME = "entry";
          public static final String COLUMN_NAME_TITLE = "title";
          public static final String COLUMN_NAME_SUBTITLE = "subtitle";
      }
  }
```

SQLite: creation code

- Constructor: call the superconstructor

```
Public mySQLiteHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
}
```

DB_NAME could be "grades.db" in our example
DB_VERSION = 1

- onCreate(SQLiteDatabase db): create the tables

```
String sql_grade = "create table " + TABLE_GRADES + "( " +
 COL_ID + " integer primary key autoincrement, " +
 COL_FIRSTNAME + " text not null, " +
 COL_LASTNAME + " text not null, " +
 COL_CLASS + " text not null, " +
 COL_GRADE + " text not null );";
db.execSQL(sql_grade);
```

If using a contract, this is replaced by _ID

SQLite: deletion code

- Close the connection in the onDestroy()

```
protected void onDestroy() {
    sql.close();
    super.onDestroy();
}
```

Since getWritableDatabase() and getReadableDatabase() are expensive to call when the database is closed, you should leave your database connection open for as long as you possibly need to access it. Typically, it is optimal to close the database in the onDestroy() **of the calling Activity**.

SQLite: insert code

- Best practice: Create a public method, like insertDb(…)

```
mySQLiteHelper sql = new mySQLiteHelper(getContext());
SQLiteDatabase db = mySQLiteHelper.getWritableDatabase();


ContentValues cv = new ContentValues();
cv.put(mySQLiteHelper.COL_FIRSTNAME, firstName);
cv.put(mySQLiteHelper.COL_LASTNAME, lastName);
cv.put(mySQLiteHelper.COL_CLASS, className);
cv.put(mySQLiteHelper.COL_GRADE, grade);
long id = db.insert(mySQLiteHelper.TABLE_GRADES, null, cv);
```

The first parameter is the db, the second parameter tells what to do when nothing gets inserted.

SQLite: delete code

- Best practice: Create a public method, like deleteDb(…)
- The delete method returns the number of rows affected
- Example:

```
db.delete(mySQLiteHelper.TABLE_GRADES, “id = ?”, new     String[]
{Integer.toString(id_to_delete)});
```

Second parameter is the “WHERE” clause of the delete operation.

Third parameter is a list of elements that get injected into the second parameter in order (replacing the ?)

SQLite: update code

- Create a public method, like updateDb(…)

```
ContentValues cv = new ContentValues();
values.put(mySQLiteHelper.FIRSTNAME, firstName);
values.put(mySQLiteHelper.LASTNAME, lastName);

db.update(mySQLiteHelper.TABLE_GRADES, values, “id = ?”, new    String[]
{Integer.toString(id_to_update)};
```

SQLite: search code

- Create a public method, like getFromDb(…)

```
Cursor gradeCursor = db.query(mySQLiteHelper.TABLE_GRADES, // the table (FROM)
 new String[]{mySQLiteHelper.COL_GRADE},  // Cols to include in result (the SELECT)
mySQLiteHelper.COL_ID + " = ?",           // WHERE clause
new String[] {Integer.toString(id_to_update)},   // Inject in the where
null,                                     // GROUP BY?
null,                                     // FILTER BY?
mySQLiteHelper.COL_GRADE + " DESC");      // SORT BY?
```

Cursors: data handlers

- A Cursor stores data given by a DB query
- Some methods:
  - getCount()
  - moveTo{First,Next,Last,Position,Previous}()
  - close()
- You need to look inside the Cursor to see query's results

```
while (gradeCursor.moveToNext()) {
  Log.v("GRADES",gradeCursor.getString(0));
}
```

This because the cursor pointer gets initiated to -1

Cursors: methods

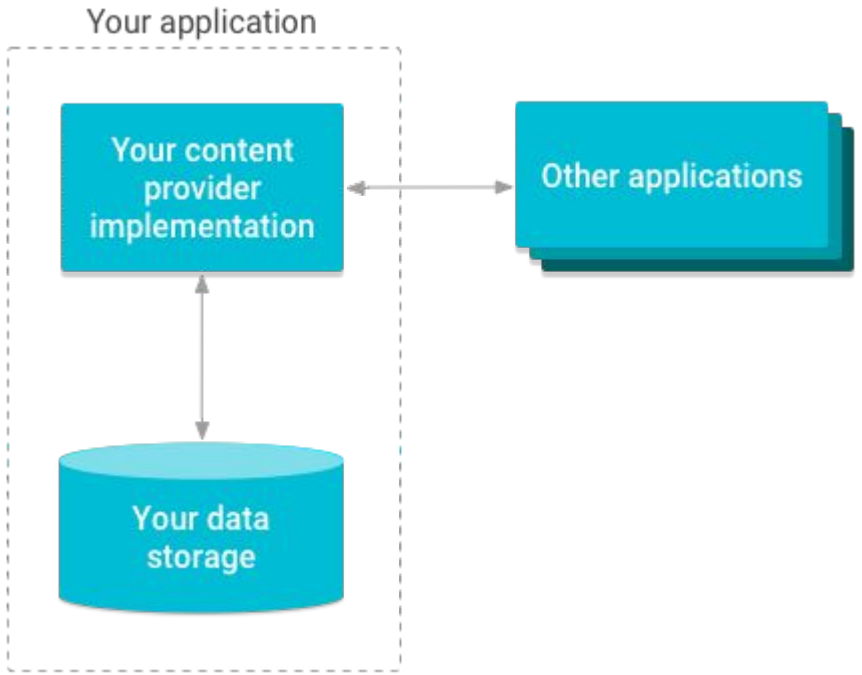
- Manipulating the cursor
  - cursor.moveToFirst()
  - while (cursor.moveToNext())
  - for (cursor.moveToFirst(); !cursor.isAfterLast(); cursor.moveToNext())
- Get column numbers from names
  - int nameColumn = cursor.getColumnIndex(People.NAME);
  - int phoneColumn = cursor.getColumnIndex(People.NUMBER);
- Get Data from column
  - String name = cursor.getString(nameColumn);
  - String number = cursor.getString(phoneColumn);

Content Providers

- A system to access shared data
- Similar to a REST web service
- For each Content Provider, one or more URIs are assigned in the form:

  content://<authority>/path

- Be aware that some ContentProviders may request permissions



How to use a Content Provider

- Need to get the URI
  - Usually this is declared as public inside the content provider class
  - URI = Table in the provider (authority = DB, path = table in the DB)
- Make a query, maybe adding some where clauses
  - You'll get a Cursor after that
- Navigate the Cursor

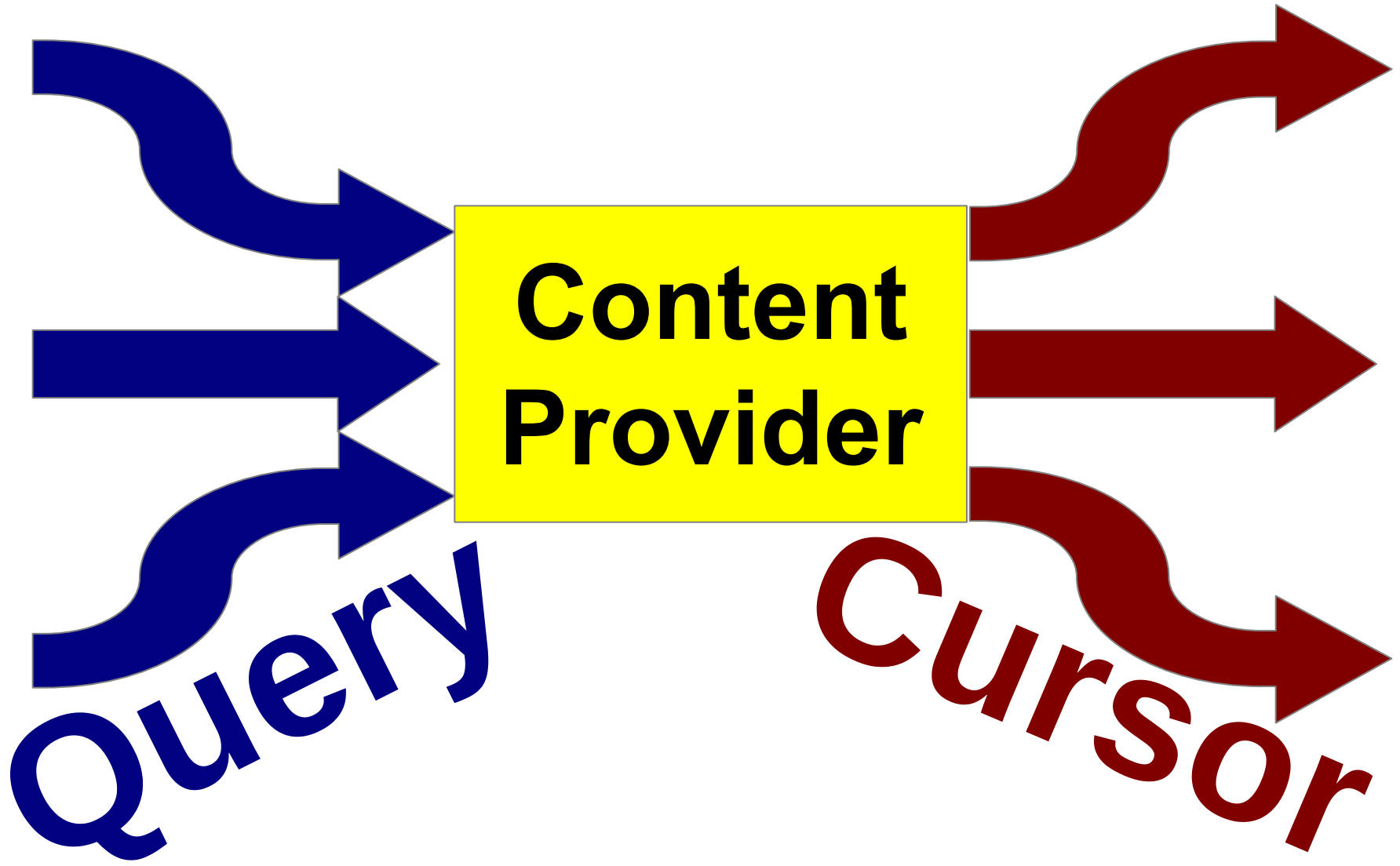
Android Content Providers

Class	Description
AlarmClock	To interact with the alarm
BlockedNumberContract	To get blocked numbers
Browser	To perform commands on the browser
CalendarContract	To handle calendar information
CallLog	Log of past calls
ContactsContract	Get and add contacts
DocumentsContract	Interact with documents
DocumentsProvider	Interact with documents
MediaStore	Access Video, Pictures, Audio and more
Settings	Inquiry system settings
…	

As you can see, Content Providers allow the access to Public Files in the **Scoped Storage**

Find them all at https://developer.android.com/reference/android/provider/package-summary.html

Content Providers



Example: Content Provider "client"

- Query the contacts content provider
- Contacts information are shared among applications
- You need to request a permission

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

- Usual runtime permissions if Android > 6

Contacts: code

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    Cursor mCursor = getContentResolver().query(
                ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
    while (mCursor.moveToNext()) {
        String contactName = mCursor.getString(cursor.getColumnIndex(
         ContactsContract.Contacts.DISPLAY_NAME));
    }
    mCursor.close();
}
```

Contacts: CursorAdapter

```
// Defines a list of columns to retrieve from the Cursor and load into an output row
String[] contactListColumns =
                { ContactsContract.Contacts.DISPLAY_NAME  }; // Contract class constant containing the word column name

// Defines a list of View IDs that will receive the Cursor columns for each row
int[] contactListItems = { R.id.displayName };

// Creates a new SimpleCursorAdapter
cursorAdapter = new SimpleCursorAdapter(
    getApplicationContext(),          // The application's Context object
    R.layout.display_contact,         // A layout in XML for one row in the ListView (containing R.id.displayName)
    mCursor,                          // The result from the query
    contactListColumns,               // A string array of column names in the cursor
    contactListItems,                 // An integer array of view IDs in the row layout
    0);                           // Flags (usually none are needed)

// Sets the adapter for the ListView called contactList (and defined elsewhere)
contactList.setAdapter(cursorAdapter);
```

To build a Content Provider (at a glance)

- Create a class that extends android.content.ContentProvider
- Pick the URIs for your resources… (not necessarily a DB)

```
public class ExampleProvider extends ContentProvider {
    private static final UriMatcher uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);

        static {
            uriMatcher.addURI("com.example.app.provider", "table3", 1);
             uriMatcher.addURI("com.example.app.provider", "table3/#", 2);  } // Wildcard

     public Cursor query ( Uri uri, [[ usual query params … ]] ) {
        switch (uriMatcher.match(uri)){
          case 1: …
          case 2: ...
```

To build a Content Provider

- Register the ContentProvider in the manifest using the <provider> tag and:
  - android:authorities (unique name of the provider)
  - android:name (Class that implements it)
  - various permissions...

Sharing Files

- You can easily share files using FileProvider
- Add an entry in the manifest

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.myapp">
   <application
      ...>
      <provider
         android:name="androidx.core.content.FileProvider"
         android:authorities="com.example.myapp.fileprovider"
         android:grantUriPermissions="true"
         android:exported="false">
         <meta-data
            android:name="android.support.FILE_PROVIDER_PATHS"
            android:resource="@xml/filepaths" />
      </provider>
      ...
   </application>
</manifest>
```

Sharing Files

- Create res/xml/filepaths.xml

```
<paths>
   <files-path path="images/" name="myimages" />
</paths>
```

- Now other apps can access your file using URI like

```
content://com.example.myapp.fileprovider/myimages/default_image.jpg
```