CS 478

Software Development for Mobile Platforms

Set 7: Files, Databases and Content Providers

Ugo Buy
Department of Computer Science
University of Illinois at Chicago

April 23, 2019

0

Device hardware: Internal vs. External storage

- Typical device may have two kinds of storage
- *Internal*: Flash memory in mobile device
  - Apps installed here (unless *android:installLocation* used in manifest)
  - Used for data private to applications
  - Data removed when app uninstalled
  - Usually files are not accessible to other apps
  - But hardware shared with other apps, use judiciously
  - 2 kinds of files: *Regular* (persistent) and *Cache* (temporary)
  - Typical size: 32 GB—512 GB (iPhone XS, Samsung S10)

1

Device hardware: Internal vs. External storage

- *External*: Mounted storage device (e.g., SD card)
  - Typically larger than flash on device
  - But data accessible and modifiable by all apps
  - Also, may not be available if mounted storage device is removed from phone/tablet
  - Many phones do not have an SD card slot
  - Phones without card slot typically divide permanent storage into *internal* and *external partitions*
  - Size: up to 1 TB

2

Options for storage of persistent data

- *SharedPreferences files*: Small, private files stored in internal memory
- *Internal storage general files*: Private general storage for an app (persistent or cached)
- *External Storage files*: Large, shared files (e.g., music, media recordings)
- *SQLite database:* Database stored on device, typically in internal storage
- *Network*: Connect to storage server
- *Content Provider*: Abstraction for all kinds of storage—Allow for sharing of large data sets between apps
- https://developer.android.com/guide/topics/data/data-storage.html

3

Shared preferences

- Overview: Store **small** amounts of **primitive** data with an app
- Data persists across app executions
- Shared preferences = Set of (key, value) pairs
  - Keys are strings
  - Values can be *boolean*, *int, long, float, String*, and *Set<String>*
- Physically stored in a data file(s) owned by the application
  - Changes to preferences files are managed by an editor (i.e., instance of *SharedPreferences.Editor*) to guarantee consistency of information as file could be accessed by multiple threads
  - Getter methods defined in *SharedPreferences* class

See: https://developer.android.com/guide/topics/data/data-storage.html#pref

4

Kinds of shared preferences files

1. Activity local file (activity specific)
2. App global files
3. Default app global file

5

Getting local *SharedPreferences* object

- Default *SharedPreferences* file for an activity
- Call API *getPreferences()* in activity to get activity-specific preferences file
  - Defined in *Activity*
  - 1 Arg: *MODE_PRIVATE* (only one allowed now), ~~*MODE_WORLD_READABLE, MODE_WORLD_WRITEABLE*~~
  - Returns *SharedPreferences* object for calling activity
  - Suitable for general purpose use, not just storing user preferences
  - Just one file per activity
- https://developer.android.com/reference/android/app/Activity.html#getPreferences(int)

6

Accessing data in *SharedPreferences* object

- Given *SharedPreferences* object, access information directly with getter methods (e.g., *getInt(), getBoolean(), getLong(), getFloat(), getString(),* etc.)
  - Args: (1) key (a *String*), and (2) default value (*int, String*, etc.)
  - Return *int, boolean, long*, etc.
  - Zero arg method *getAll()* returns *Map< String, ? >* containing all pairs (do not modify)
  - Zero arg method *edit()* returns *SharedPreferences.Editor* object
  - Method *contains(String)* returns *boolean* (obvious)
- https://developer.android.com/reference/android/content/SharedPreferences.html

7

Modifying data in *SharedPreferences* object

- Send message *edit()* to *SharedPreferences* object
- Returned *SharedPreferences.Editor* instance has setter API
- Examples API methods—All return receiver (a *SharedPreferences.Editor*)
  - *putBoolean(String, boolean)*
  - *putInt(String, int)*
  - *putString(String, String)*
  - *putStringSet(String, Set<String>)*
  - *remove(String)*
- Send *commit()* message to editor to save changes to file
  - Return *true* if changes successfully applied to persistent storage
- https://developer.android.com/reference/android/content/SharedPreferences.Editor.html

8

Getting app global *SharedPreferences* object

- Define files shared by all components in an application
- Use *getSharedPreferences()* in activity, service, or broadcast receiver to get application-wide preferences
  - Defined in *Context*
  - Args: (1) Name of preference file (a *String*), and (2) file mode (an *int*)
  - Return *SharedPreferences* object for app
  - Only non-deprecated mode: *MODE_PRIVATE,* ~~*MODE_WORLD_READABLE* and *MODE_WORLD_WRITEABLE*~~
- https://developer.android.com/reference/android/content/Context.html#getSharedPreferences(java.lang.String, int)

9

Default preferences

- Use *getDefaultSharedPreferences(Context)* to get global application preferences used by default in application
  - Static method defined in class *PreferenceManager*
  - 1 arg: Context of application for which default preferences are returned
  - Return *SharedPreferences* object
  - Manipulate as usual (e.g., edit with *SharedPreferences.Editor* object)
- https://developer.android.com/reference/android/preference/PreferenceManager.html

10

Simple *SharedPreferences* example

```
public class CalendarActivity extends Activity {
   …
   static final int DAY_VIEW_MODE = 0;
   static final int WEEK_VIEW_MODE = 1;
   private SharedPreferences mPrefs;
   private int mCurViewMode;

   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      SharedPreferences mPrefs = getSharedPreferences(“elMiFile”, MODE_PRIVATE);
      mCurViewMode = mPrefs.getInt("view_mode", DAY_VIEW_MODE);
   }

   protected void onPause() {
      super.onPause();
      SharedPreferences.Editor ed = mPrefs.edit();
      ed.putInt("view_mode", mCurViewMode);
      ed.commit();
   }
}
```

Imaginary activity in a *Calendar* app.

Declaration of *SharedPreferences* field.

Read existing preferences from file.

Getter methods use default values.

Get a preferences editor to modify preferences file.

Update “view_mode” entry and commit.

11

Using general-purpose files

- Files can be stored in internal memory or external memory (e.g., SD card)
- Internal storage: Use for smaller private data
  - Advantage: Privacy
  - Disadvantage: Data erased when app uninstalled
- External storage: Use for larger, public data (e.g., media files); however, private areas available on external memory as well
  - Keep data that should not be lost when an app is uninstalled (e.g., pictures, music, etc.)

12

Java's *File* class

- "Abstract representation of a file or directory pathname" (https://developer.android.com/reference/java/io/File.html)
- Abstract representation independent of OS notations (e.g., \ vs. /)
- Actual file or directory may or may not exist
- Useful for navigating file systems, while possibly adding and deleting files
- Complex rules for forming path names
- Examples of constructors
  - *File(String)* – Constructs new instance using input path
  - *File(URI)* – Input path specified as a URI

13

Java's *File* class

- Main methods
  - *createNewFile()* : boolean – Creates actual empty file in file system
  - *canRead()* : boolean – Indicates whether context can read actual file
  - *canWrite()* : boolean – Indicates whether context can write actual file
  - *delete()* : boolean – Deletes actual file
  - *exists()* : boolean – Indicates whether actual file exists in file system
  - *isDirectory()* : boolean – Indicates whether actual file is a directory
  - *list()* : String[] – List directory (receiver must represent a directory)
  - *mkdir(), mkdirs()* : boolean – Makes directory (with parents if needed)
  - *toURI()* : URI – Returns URI for this file (receiver)

14

Reading and writing bytes in file

- Java classes *FileInputStream* and *FileOutputStream* support reading and writing of bytes in a given file
- Get *FileInputStream* by calling *Context* method *openFileInput(String)*
  - Gets *String* with name of private file
  - Returns *FileInputStream* instance
  - Can throw *FileNotFoundException*
- https://developer.android.com/reference/android/content/Context.html
  https://developer.android.com/reference/java/io/FileInputStream.html
  https://developer.android.com/reference/java/io/FileOutputStream.html

15

Writing bytes to a file

- Get *FileOutputStream* with *Context* method *openFileOutput(String, int)*
  - Opens private file
  - Params: (1) *String* with name of private file, (2) *int* specifying file mode (can be MODE_PRIVATE or MODE_APPEND)
  - Will create file if it does not exist
  - MODE_APPEND will append content to file rather than erasing it
  - Returns *FileOutputStream* instance
- http://developer.android.com/reference/android/content/Context.html
  http://developer.android.com/reference/java/io/FileOutputStream.html

16

Reading characters from file

- Class *InputStreamReader* converts byte stream (e.g., obtained from *FileInputStream* instance) into character stream
- Get an *InputStreamReader* by *FileInputStream* by calling constructor with *FileInputStream* arg, e.g,

  new *InputStreamReader(aFileInputStream)*

  - Part of Java
  - Main method: *read()* (reads one character, return it as *int*)
- Still, a bit inconvenient to use; would rather get whole lines or words at once from input file
- Doc: http://developer.android.com/reference/java/io/InputStreamReader.html

17

Reading whole lines from file

- Class *BufferedReader* buffers character stream until end-of-line
- Get *BufferedReader* from *InputStreamReader* by calling constructor with *InputStreamReader* arg

  new *BufferedReader(InputStreamReader)*

  - Again, part of Java
  - Main method: *readLine()* (reads one line of text, return it as a *String*)
- Now we are talking…
- Doc: http://developer.android.com/reference/java/io/BufferedReader.html

18

Writing (multi-line) text to file

- Class *OutputStreamWriter* turns character stream into byte stream
- The inverse of *InputStreamReader*
- Get *OutputStreamWriter* from *FileOutputStream* by calling constructor with *FileOutputStream* arg

  new *OutputStreamWriter(aFileOutputStream)*

  - Again, part of Java
  - Main methods: *write(int), flush(), close()* (closes also associated *FileOutputStream*)
  - Method *write(String, int_offset, int_length)* writes a portion of a string
- Doc: http://developer.android.com/reference/java/io/OutputStreamWriter.html

19

Writing (multi-line) text to file (cont'd)

- Class *BufferedWriter* writes multiple lines of text to output stream
- Get *BufferedWriter* from *OutputStreamWriter* by calling constructor with *OutputStreamWriter* arg

  new *BufferedWriter(OutputStreamWriter)*

  - Again, part of Java
  - Main methods: *write(String)* (inherited from *Writer*), *close(), flush(), newLine()…*
- Like *BufferedReader,* a useful class
- Doc: http://developer.android.com/reference/java/io/BufferedWriter.html

20

Working with external storage

- Similar to case of internal storage
- Big caveat: External storage could be added or removed without warning (e.g., an SD card)
- Good idea: Before working with external memory, check state
- Class *Environment* defines **static** method *getExternalStorageState()* (overloaded, no-arg or file path arg)
- Return string indicate state of external storage, e.g.,
  - MEDIA_REMOVED, MEDIA_MOUNTED, MEDIA_UNMOUNTED, MEDIA_EJECTING, MEDIA_MOUNTED_READ_ONLY, etc.
- Doc: https://developer.android.com/reference/android/os/Environment.html

21

Working with external memory (cont'd)

- Another caveat: Needs permissions to read and write to external memory since API 16 (changed for private files since KK, though)
  - android.permission.READ_EXTERNAL_STORAGE
    android.permission.WRITE_EXTERNAL_STORAGE
  - Dangerous level: Must ask user at R-T in post-MM model
  - Not needed for app-specific files obtained with *getExternalFilesDir()* since KK (API 19)
- E.g.,

```
<manifest ...>
  <uses-permission
     android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
  …
```

- Doc: https://developer.android.com/guide/topics/data/data-storage.html

22

Getting shared file in external memory

- Paradigm:
  1. Get path to external directory (an instance of *File*)
  2. Create directory at that path
  3. Create a new *File* instance specifying path (a *File*) and file name (a *String*)
- Step 1 can be accomplished in different ways (see next)
- Class *Environment* manages general device information
- Doc: https://developer.android.com/reference/java/io/File.html
- Doc: https://developer.android.com/reference/android/os/Environment.html

23

1. Getting directory path

- Static *Environment* method *GetExternalStoragePublicDirectory()*
  - Arg string specifies the type of content to be stored in directory, e.g., DIRECTORY_MUSIC, DIRECTORY_PICTURES, DIRECTORY_MOVIES, DIRECTORY_DOWNLOADS, etc.
  - Returns *File* object
- Alternatively use *Context* method *GetExternalFilesDir()*
- Again, arg specifies the type of content to be stored (see above)
  - Directory is considered private, no permissions required of owner app (KitKat and subsequent versions)
  - Other apps require usual permissions to read/write files there
- Doc: https://developer.android.com/reference/android/content/Context.html
  Doc: https://developer.android.com/reference/android/os/Environment.html

24

2. Creating directory + 3. Creating file in directory

- Use path returned by one of two earlier methods, to create directory, e.g., *path.mkdirs()* (defined in *File*)
  - Return *true* if creation was successful, *false* if directory already existed or failed to create directory
- If directory exists, now create the file by passing directory (a *File*) and file name (a *String*) to *File* constructor

25

SQLite databases

- Support for DBs accessible within an app, not shared with other apps
- SQLite is a full-fledged relational DBMS, not just for Android
  - SQL (Sequential Query Language) is the universal language for programming relational databases
  - Relational databases organize data in tables
- Compliant with SQL-92 standard except for few missing features:
  - Few ALTER TABLE options (support RENAME TABLE and ADD COLUMN; no support for DROP COLUMN, etc.)
  - VIEWS are read-only, etc.
- SQLite footprint is very small wrt to regular databases, with support for query optimization, fast access to data, ACID transactions

26

More on SQLite

- ACID transactions, just like any database
  - Atomic (all or nothing transaction effects)
  - Consistent (transactions preserve database constraints)
  - Isolated (Concurrent transactions are serializable)
  - Durable (Transaction effects are permanent even in the event of software crashes)
- No client/server model with multiple server threads
  - Library implementing SQLite is linked with app using it
  - Dynamic linking available
  - Faster access than regular DB because no IPC involved

27

Working with databases in Android

- New classes needed for working with *SQLite* databases
  - *SQLiteOpenHelper* – Abstract class to manage database creation and version management
  - *ContentValues* – Set of <*key, value*> pairs specifying database row
  - *Cursor* – Interface defining iterator over values returned by DB queries
  - *SimpleCursorAdapter* – Class mapping cursors (e.g, returned from database queries) to list items suitable for display according to some XML layout specification
  - *SQLiteDatabase* – DB object
- https://developer.android.com/training/data-storage/sqlite.html

28

Creating SQLite database

- Define class that extends *SQLiteOpenHelper*
  - Helps create/open/update DB
  - Define SQL statements as strings, pass strings to SQLite DB
  - Methods *getReadableDatabase()* and *getWritableDatabase()* will return *SQLiteDatabase* object (no args)
    - DB created if it does not exist, opened if it exists
    - Do not call from UI thread, as these methods can be slow
  - Method *close()* closes DB

https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html

29

Subclassing *SQLiteOpenHelper:* Methods needed

1. Constructor
2. *onCreate(SQLiteDatabase): void –*
3. *onUpgrade(SQLiteDatabase, int, int): void –*
4. (optional) *onOpen(SQLiteDatabaset): void –*

Caveat: Following methods inherited from superclass *SQLiteOpenHelper* need not be redefined

- *GetReadableDatabase(): SQLiteDatabase*
- *GetWritableDatabase(): SQLiteDatabase*

30

Subclassing *SQLiteOpenHelper:* Constructor (cont'd)

- Constructor must invoke superclass's constructor
- Superclass *SQLiteOpenHelper* has multiple constructors, basic constructor takes four parameters
  1. *Context* context – the usual
  2. *String* name – name of database file, **null** if database held in RAM
  3. *SQLiteDatabase.CursorFactory* factory – Java interface used to create cursor objects, **null** for default factory
  4. *int* version – Version number, starting at 1

31

Subclassing *SQLiteOpenHelper:* Constructor

- Constructor returns helper object for creating/opening/managing a database
- Database not created or opened until methods *getWritableDatabase()* or *getReadableDatabase()* are called
- https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.CursorFactory.html

32

Class *SQLiteOpenHelper:* Methods

- *getReadableDatabase()*: *SQLiteDatabase*
  - Create and/or open a database, could be read-only
  - *SQLiteDatabase* instance returned manages actual database, is valid until *close()* or *getWriteableDatabase()* called
  - Do not call from UI thread!
- *getWritableDatabase()*: *SQLiteDatabase*
  - Create and/or open a database used **for reading and writing data**
  - First time calling this method will trigger calls to *onCreate(), onUpgrade(),* and *onOpen()*
  - Do not call from UI thread!
- Either way: Do not forget to call *close()* when done

33

Subclassing *SQLiteOpenHelper:* Callback methods

Define three key abstract methods from *SQLiteOpenHelper*, all return **void**

- *onCreate(SQLiteDatabase)* – Called when database created the first time
  - Create and populate database tables, e.g., using "CREATE TABLE" and "INSERT INTO" commands
- *onUpgrade(SQLiteDatabase, int, int)* – Called when database opened with a different version number than the previous version number
  - Typical actions: add and/or drop tables, add columns to table, etc.
  - **int** args specify old and new version numbers
  - Execute DROP TABLE, ALTER TABLE commands, etc.
- (optional) *onOpen(SQLiteDatabase)* – Called after database is opened (e.g., with *getReadableDatabase()* or *getWritableDatabase()*)

34

Class *ContentValues*

- Class for representing database tuples (rows)
- Set of (*key*, *value*) pairs, where *keys* are column names and *values* are column values in the tuple
- Values are types that can be stored in a database (e.g., primitive types and *Strings*)
- https://developer.android.com/reference/android/content/ContentValues.html

35

Class *ContentValues* (cont'd)

- Main methods:
  - Most getters take key as *String*, return value of appropriate type:
    - *get(String): Object*
    - *getAsBoolean(String): Boolean*
    - *getAsInteger(String): Integer*
    - *getAsString(String): String*
  - Other getters:
    - *keySet(): Set<String>*
    - *valueSet(): Set<Entry<String,Object>>*
    - *size(): int* (obvious)
    - *toString(): String*

36

Class *ContentValues* (cont'd)

- Main methods:
  - Setters (all **void**):
    - *put(String, Boolean)*
    - *put(String, Integer)*
    - *put(String,String)*
    - *putAll(ContentValues)*, etc.
- Other setters: *void remove(String); void clear()*

37

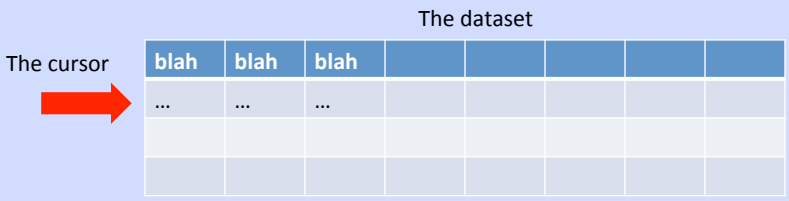
Interface *Cursor*

- Java Interface that "… provides random read-write access to the result set returned by a database query."
- Implement by extending abstract class *AbstractCursor*
- Predefined concrete subclass *SQLiteCursor* often gets job done
- An extension of the *Iterator* design pattern
- Also, allows observers on the dataset or the source backing data set (see, e.g., *Observer* pattern)
- Thread unsafe by default, if multiple threads can access same *Cursor* object, you must write thread-safe code
- Rich APIs allow to retrieve content from database query (see next)
- https://developer.android.com/reference/android/database/Cursor.html

38

Interface *Cursor*

- Logical view of a cursor



39

Java interface *Cursor*

- Main APIs:
  - *getCount()*: *int* – Return the number of rows in the data set
  - Cursor movement (all *boolean*, return *false* if empty cursor, or if element does not exist):

    moveToNext(), moveToFirst(), moveToLast(), moveToPrevious(), moveToPosition(int position), *move(int* offset), etc.
  - Testing (all boolean): *isFirst(), isLast(), isBeforeFirst(), isAfterLast()*
  - DB value getters (all take column index):

    int getInt(int), String getString(int), double getDouble(int), etc.
  - Other getters:

    String[] getColumnNames(), String getColumnName(int), int getColumnIndex(String), etc.

40

Java interface *Cursor*

- Main APIs (continued):
  - Termination API: *close()* : *void*
  - Observer API (all *void)*:

    registerDataSetObserver(DataSetObserver), registerContentObserver(ContentObserver), unregisterDataSetObserver(DataSetObserver), unregisterContentObserver(ContentObserver)
  - Automatically notify registered observers of changes in data set associated with this cursor or data repository backing a data set
- *DataSetObserver* and *ContentObserver* are abstract classes declaring observer behavior (e.g., with method *onChange()* and *onChanged()*)

41

Class *SQLiteCursor*

- Concrete subclass of *AbstractCursor* by way of *AbstractWindowedCursor*

  "A Cursor implementation that exposes results from a query on a SQLiteDatabase. SQLiteCursor is not internally synchronized so code using a SQLiteCursor from multiple threads should perform its own synchronization when using the SQLiteCursor."

  https://developer.android.com/reference/android/database/sqlite/SQLiteCursor.html
- Customer beware: Class not designed to be thread-safe
  - If cursor accessed by multiple threads, implement appropriate synchronization yourself

42

Class *SimpleCursorAdapter*

- Concrete *Cursor* implementation that maps database columns to text views and image views defined in *res/layout* XML files http://developer.android.com/reference/android/widget/SimpleCursorAdapter.html
- Yet another use of *Adapter* pattern from Go4 system
- Must bind columns from cursor to text views or image views in XML file
- *SimpleCursorAdapter(Context context, int layout, Cursor c, String[] from, int[] to, int flags)*
  - Constructor takes (1) context of list view, (2) layout (XML) file id, (3) cursor, (4) column names to bind, (5) text views that will display columns, (6) flags
- Method *bindView(View, Context, Cursor)* : *void* – Perform binding

43

Class *SQLiteDatabase*

- Encapsulate SQLite database
  https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html
- A largish class supporting database creation, deletion and operations, such as queries and transactions
- DB Transaction = Sequence of queries
- Transactions are ACID (Atomic, Consistent, Isolated and Durable)
- Template of a typical transaction:
  - *beginTransaction()* : *void*
  - Sequence of queries
  - *endTransaction()* : *void*

44

Class *SQLiteDatabase* : Query methods

- Methods mirror queries in relation DB systems, e.g.,
  - SELECT, UPDATE, DELETE, etc.
- A largish class supporting database creation, deletion and operations, such as queries and transactions
- Key methods
  - *execSQL(String)* : *void*
  - *beginTransaction(), endTransaction()* : *void*
  - *insert(String table, String nullHack, ContentValues values)* : *long*
  - *delete(String table, String whereClause, String[] whereArgs)* : *int*
  - *query(String table, …)* : *Cursor*

45

The *query()* method

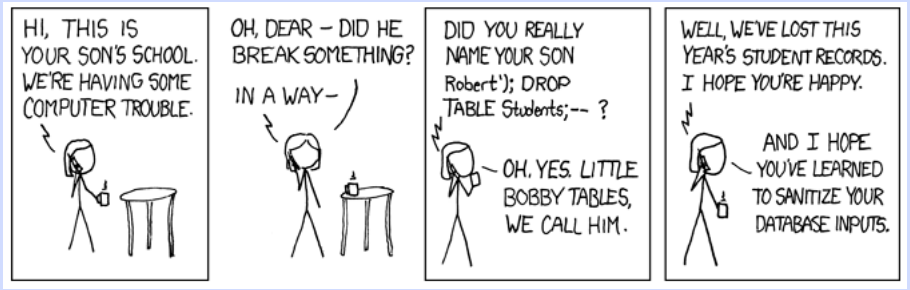
- API for querying a database table
- *query()* message is mapped to corresponding database SELECT cmd
- Arguments:
  - *String* table = name of table to be queried
  - *String[]* columns = columns to be returned
  - *String* selection = condition on rows to be returned (WHERE clause)
  - *String[]* selectionArgs = Arguments to be inserted for ?s in selection
  - *String* groupBy = GROUP BY clause
  - *String* having= HAVING clause
  - *String* orderBy = ORDER BY clause

https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html

46

Why separate selection and selection arguments?

- Prevent database injection attacks
  - User input can become part of SQL statement
  - Check statements entered by user **before** passing along to database
  - From WebComic https://xkcd.com/327/



47

not covered

Content providers

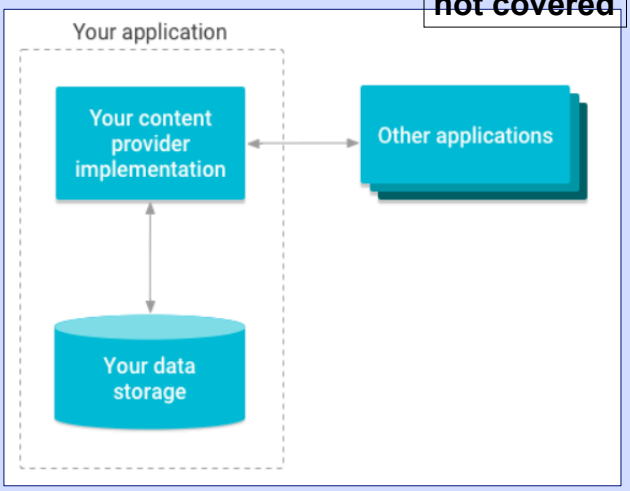
- One of four basic elements of Android apps (along with activities, receivers, and services)
- Used for sharing information among apps
  - No need for provider unless data shared among apps!
- Goal: Provide uniform interface to different kinds of data stores
  - Abstract away how data stored
  - Interface similar to relational database, but actual data could be stored anywhere (RAM, internal + external storage, network, etc.)
  - So, think tables, rows (records), columns (attributes), etc.

https://developer.android.com/guide/topics/providers/content-providers.html
https://developer.android.com/reference/android/content/ContentProvider.html

48

not covered

Abstract view of content provider



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

49

not covered

Content providers (cont'd)

- Most popular predefined providers:
  - Contacts provider
  - Calendar provider
  - Email provider
  - Browser provider
  - and many more
- You may define new providers as well
- Must define content provider if you want to:
  1. Implement custom suggestions in app
  2. Cut/copy/paste data from your app to other apps

50

not covered

Content providers (cont'd)

- Outline of this section:
  1. SQLite databases
  2. Using providers: Authorities, contracts, content resolvers and content providers
  3. Creating new providers

51

SQLite databases and content providers

not covered

- Data accessible through content provider is often stored in an SQLite database
  - No particular requirements on the database in this case
- However, if data in content provider must be displayed in an *AdapterView* (e.g., *ListView* or *GridView*), underlying table(s) must contain an integer primary key column named *_ID*
  - This was the case for *DataManagementSQL* app
  - This key will be used by adapter to enumerate elements to be displayed

52

Working with content providers

not covered

- First, make sure content provider is needed
  - Provider may not be needed if application does not share data with other apps
- However, content provider will be needed, e.g., if your app…
  - Sends data to widget
  - Returns custom suggestions use *SearchRecentSuggestionsProvider*
  - Synchronizes data with remote server using concrete subclass of class *AbstractThreadedSyncAdapter*
  - Loads data to UI using *CursorLoader*

https://developer.android.com/guide/topics/providers/content-provider-basics.html

https://developer.android.com/reference/android/content/SearchRecentSuggestionsProvider.html

53

not covered

Working with content providers

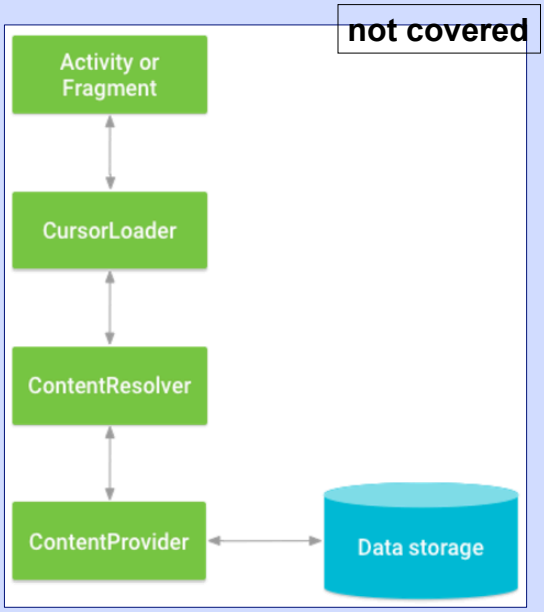
- Clients of content providers must use a *ContentResolver* instance to access content provider defined by another app
  - Get content resolver by calling no-arg method *getContentResolver()* (defined by *Context*)
- Content resolver's responsibilities:
  1. Find appropriate provider, given a *Content URI*
  2. Manage all IPC between client app and other app's *ContentProvider*
- Content resolver uses *content URI* to identify provider and its data
- Client often uses *CursorLoader* instance to access data in content resolver

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

54

not covered

Diagram of typical provider use



Source:
https://developer.android.com/guide/topics/providers/content-provider-basics.html

55

not covered

Content URIs

- Goal: Identify content provider and specific data in the provider
- Format: Special URI with *content* scheme

  content://*authority/path/id*

  - *content:* scheme identifier (vs. *tel*, *http*, etc.)
  - *authority:* id of the content provider, often its package name
  - *path:* (optional) path within provider (e.g., a table name or a directory structure) — zero or many segments separated by "/" chars
  - *id:* (optional) specific location (e.g., a row) within the *path*
- E.g., **content://com.android.chrome/bookmarks/search_suggest_query**
- https://developer.android.com/reference/android/content/ContentUris.html

56

not covered

Finding a provider

- Content resolver takes a content URI object and looks up URI's authority in system-wide table of registered providers
- https://developer.android.com/guide/topics/providers/content-provider-basics.html

57

not covered

Building content URIs

- Helper classes *ContentUris* and *Uri.Builder*
- *Uri.Builder* allows you to build *ContentUri* instance incrementally
  - *appendPath(String* newSegment*)*
  - *authority(String* authority*)*
  - *query(String* query*)*
  - *build()* → *Uri* instance
- https://developer.android.com/reference/android/content/ContentUris.html
  https://developer.android.com/reference/android/net/Uri.Builder.html

58

not covered

Class *ContentResolver*

- Main: Support Inter-Process Communication (IPC) between process running client app and process owning content provider
- Also, support CRUD functionality for data store hosted by provider
  - Create
  - Retrieve
  - Update
  - Delete
- Always use *content URI* to identify data to be accessed and/or modified

59

not covered

Main *ContentResolver* methods

- Methods to support CRUD functionality (without signatures)
  - *query()* – select operation on content URI
  - *insert()* – insert row in content URI
  - *delete()* – delete rows in content URI
  - *update()* – update rows in content URI

60

not covered

Key *ContentResolver* methods: *query()*

- Perform SELECT query on provider – Return *Cursor* instance
- Parameters:
  - *Uri uri* – authority of content to retrieve (either table or row)
  - *String[] projection* – list of column names to return
  - *String selection* – a WHERE clause (except for word "WHERE") restricting range of rows to be returned
  - *String[] selectionArgs* – values matching "?" marks from *selection* (by position)
  - *String sortOrder* – a ORDER BY clause (except for words "ORDER BY") defining order of rows returned by query
- https://developer.android.com/guide/topics/providers/content-provider-basics.html

61

not covered

Key *ContentResolver* methods: *insert()*

- Inserts new row in table
- If content provider supports transactions, insertion is atomic
- Parameters:
  - *Uri uri* – content URI of table
  - *ContentValues values* – Initial values of columns in row
- Passing empty *ContentValues* object inserts empty row
- Return value: *Uri* of inserted row

62

not covered

Key *ContentResolver* methods: *delete()*

- Delete rows in table
- If content provider supports transactions, deletion is atomic
- Parameters:
  - *URI uri* – authority of table or row
  - *String where* – WHERE clause (excluding keyword "WHERE")
  - *String[] selectionArgs* – Args matching "?" marks in *where* clause
- Return value: number of rows deleted (an *int*)

63

not covered

Key *ContentResolver* methods: *update()*

- Updates rows in table
- If content provider supports transactions, updates are atomic
- Parameters:
  - *Uri uri* – authority of table or row to be modified
  - *ContentValues values* – The new field values
  - *String where* – WHERE clause (excluding "WHERE")
  - *String[] selectionArgs* – Args matching "?" marks in *where* clause
- **null** field value in *values* removes existing field value (now empty)
- Throws *NullPointerException* (unchecked) if *uri* or *values* is **null**
- Return value: number of rows updated (an *int*)

64

not covered

Additional *ContentResolver* APIs

- Observer API: register and unregister observers
  - *registerContentObserver(Uri, ContentObserver)*
  - *unregisterContentObserver(ContentObserver)*
  - Observers notified when provider at specified*Uri* changes
- To use, extend abstract class *ContentObserver*, possibly overriding method *onChange()*
- Synchronization API with underlying content provider
- Streaming APIs (e.g., when content provider encapsulates media data)

65

not covered

Displaying query's results

Similar to case of *SQLiteDatabase* query

- Typically results displayed in *ListView* object
- Method *getCount()* of class *Cursor* returns number of items in receiver
- Link cursor to *ListView* by supplying adapter
  - Again, class *SimpleCursorAdapter* often does it
  - *SimpleCursorAdapter(Context c, int layout, Cursor cursor, String[] from, int[] to)*
  - *from* argument specifies column names
  - *to* argument specifies views (by their *id*) that will display matching column (by position)
  - *cursor* could be **null** (not queried yet)

66

not covered

More on class *SimpleCursorAdapter*

Additional key methods

- *bindView(View v, Context c, Cursor cursor)* : *void*
  - Receiver is *SimpleCursorAdapter* instance
  - Receiver binds all fields names in *to* arg with cursor columns specified in *from* arg
- *swapCursor(Cursor c)* : *Cursor*
  - Swaps new cursor, returning old cursor
- https://developer.android.com/reference/android/widget/SimpleCursorAdapter.html

67

not covered

Batch access to content provider

- Useful when multiple insertions (i.e., rows) needed
  - Goal: Coalesce multiple provider access operations into a single one
- Method:
  1. Create *ArrayList* of class *ContentProviderOperation*
  2. Pass *ArrayList* instance to *applyBatch()*
- *applyBatch(String authority, ArrayList<CPO> operations)*
  - Applies all *operations* to provider, returns array of *ContentProviderResult* objects
- https://developer.android.com/reference/android/content/ContentProviderOperation.html

68

not covered

Provider permissions

- Provider app must specify permissions that other apps must have in order to use the content provider
  - Provider is not exported unless it requires a permission of its clients
- Permissions
  - Define bespoke permission with <permission …/> clause in AndroidManifest.xml file of provider app
  - Require permission using android:permission attribute in provider tag
  - Client apps: Declare <uses-permission /> tag in AndroidManifest.xml to be able to use provider

69

not covered

Defining content providers

Recipe for defining new content provider

1. Define storage base, e.g., SQLite database or file
2. Extend framework class *ContentProvider*
3. Define authority string, content URIs, column names, permissions that clients must have to use provider.
4. (Optional) Create final *Contract* class specifying names for things in Item 3
   - Supply *Contract* class to clients as a .jar file that they'll compile in their apps
5. Declare provider component in *AndroidManifest.xml* file

70

not covered

Defining *ContentProvider* subclass

- Required methods – Same signature as identically-named *ContentResolver* methods
  - *query()* -- Could be called from multiple threads
  - *insert()*
  - *update()*
  - *delete()*
  - *getType(Uri)* : *String* – Returns MIME type of arg URI
  - *onCreate()* : *boolean* – Called when app containing provider is started (Beware: do not block UI)
- **All except *onCreate()* must be thread-safe**
- https://developer.android.com/guide/topics/providers/content-provider-creating.html

71

not covered

Defining provider permissions

- Define permissions that other apps must have to read/write data to content exposed by provider
- Content must be stored in private files, consider internal memory
  - If using public files, permission is meaningless as other apps can access content directly
- Caveat: Without permissions, private data backing your provider will be exposed
  - See lectures on permissions on how to define new permissions
  - **Specified permissions must be granted at install time even in the post-MM model**
- https://developer.android.com/guide/topics/providers/content-provider-creating.html

72

not covered

Declaring *ContentProvider* element

- Declare provider in *AndroidManifest.xml* file using <provider> tag

```
<provider android:authorities="semicolon-separated-list”
        android:name="name-of-class”
        android:process=":local-process"
        android:permission="string”
        android:readPermission="string”
        android:writePermission="string”
        android:grantUriPermission=["true" | "false”]
        android:syncable=["true" | "false”] >
</provider>
```

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

73