Android Developer Fundamentals

Storing Data

Lesson 9

9.0 Storing Data

Contents

- Files
- Internal Storage
- External Storage
- SQLite Database
- Other Storage Options

Storage Options

Storing data

- Shared Preferences—Private primitive data in key-value pairs
- Internal Storage—Private data on device memory
- External Storage—Public data on device or external storage
- SQLite Databases—Structured data in a private database
- Content Providers—Store privately and make available publicly

Storing data beyond Android

- Network Connection—On the web with your own server
- Cloud Backup—Back up app and user data in the cloud
- Firebase Realtime Database—Store and sync data with NoSQL cloud database across clients in realtime

Files

Android File System

- External storage -- Public directories
- Internal storage -- Private directories for just your app

Apps can browse the directory structure

Structure and operations similar to Linux and java.io

Internal storage

- Always available
- Uses device's filesystem
- Only your app can access files, unless explicitly set to be readable or writable
- On app uninstall, system removes all app's files from internal storage

External storage

- Not always available, can be removed
- Uses device's file system or physically external storage like SD card
- World-readable, so any app can read
- On uninstall, system does not remove files private to app

When to use internal/external storage

Internal is best when

- you want to be sure that neither the user nor other apps can access your files

External is best for files that

- don't require access restrictions and for
- you want to share with other apps
- you allow the user to access with a computer

Save user's file in shared storage

- Save new files that the user acquires through your app to a public directory where other apps can access them and the user can easily copy them from the device
- Save external files in public directories

Internal Storage

Internal Storage

- Uses private directories just for your app
- App always has permission to read/write
- Permanent storage directory—getFilesDir()
- Temporary storage directory—getCacheDir()

Creating a file

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

Use standard java.io file operators or streams to interact with files

External Storage

External Storage

- On device or SD card
- Set permissions in Android Manifest
  - Write permission includes read permission

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

Always check availability of storage

```
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
```

Example external public directories

- DIRECTORY_ALARMS and DIRECTORY_RINGTONES
  For audio files to use as alarms and ringtones
- DIRECTORY_DOCUMENTS
  For documents that have been created by the user
- DIRECTORY_DOWNLOADS
  For files that have been downloaded by the user

developer.android.com/reference/android/os/Environment.html

Accessing public external directories

1. Get a path getExternalStoragePublicDirectory()
2. Create file

```
File path = Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES);
File file = new File(path, "DemoPicture.jpg");
```

How much storage left?

- If there is not enough space, throws IOException
- If you know the size of the file, check against space
  - getFreeSpace()
  - getTotalSpace().
- If you do not know how much space is needed
  - try/catch IOException

Delete files no longer needed

- External storage

```
myFile.delete();
```

- Internal storage

```
myContext.deleteFile(fileName);
```

Do not delete the user's files!

When the user uninstalls your app, your app's private storage directory and all its contents are deleted

Do not use private storage for content that belongs to the user!

For example

- Photos captured or edited with your app
- Music the user has purchased with your app

Shared Preferences & SQLite Database

SQLite Database

- Ideal for repeating or structured data, such as contacts
- Android provides SQL-like database
- Covered in following chapters and practicals
  - SQLite Primer
  - Introduction to SQLite Databases
  - SQLite Data Storage Practical
  - Searching an SQLite Database Practical

Shared Preferences

- Read and write small amounts of primitive data as key/value pairs to a file on the device storage

- Covered in later chapter and practical
  - Shared Preferences

Other Storage Options

Use Firebase to store and share data

Store and sync data with the Firebase cloud database

Data is synced across all clients, and remains available when your app goes offline

firebase.google.com/docs/database/

Firebase Realtime Database

- Connected apps share data
- Hosted in the cloud
- Data is stored as JSON
- Data is synchronized in realtime to every connected client

Network Connection

- You can use the network (when it's available) to store and retrieve data on your own web-based services

- Use classes in the following packages
  - java.net.*
  - android.net.*

Backing up data

- Auto Backup for 6.0 (API level 23) and higher
- Automatically back up app data to the cloud
- No code required and free
- Customize and configure auto backup for your app
- See Configuring Auto Backup for Apps

Backup API for Android 5.1 (API level 22)

1. Register for Android Backup Service to get a Backup Service Key
2. Configure Manifest to use Backup Service
3. Create a backup agent by extending the BackupAgentHelper class
4. Request backups when data has changed

More info and sample code: Using the Backup AP and Data Backup

Learn more about files

- Saving Files
- getExternalFilesDir() documentation and code samples
- getExternalStoragePublicDirectory() documentation and code samples
- java.io.File class
- Oracle's Java I/O Tutorial

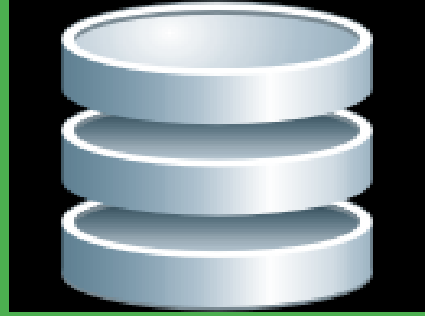
Learn more about backups

- Configuring Auto Backup for Apps
- Using the Backup API
- Data Backup

What's next?

- Concept Chapter: 9.0 C Storing Data
- No practical, this was an overview lecture

END



SQLite

History

- SQlite is an open source embedded database. The original implementation was designed by D. Richard Hipp.
- Hipp was designing software used on board guided missile systems and thus had limited resources to work with.
- The resulting design goals of SQLite were to allow the program to be operated without a database installation or administration.

Owens, Michael (2006). The Definitive Guide to SQLite. Apress. doi:10.1007/978-1-4302-0172-4_1. ISBN 978-1-59059-673-9.
http://en.wikipedia.org/wiki/SQLite#cite_note-1

Continued...

- In 2000 version 1.0 of SQLite was released. This initial release was based off of GDBM (GNU Database Manager). Version 2.0 replaced GBDM with a custom implementation of B-tree data structure.
- Version 3.0 added many useful improvements such as internalization and manifest typing.
- This version was also partially funded by America Online and shows how SQLite has quickly grown from an unheard of pet project to the widely used open source system it is today.

Owens, Michael (2006). The Definitive Guide to SQLite. Apress. doi:10.1007/978-1-4302-0172-4_1. ISBN 978-1-59059-673-9.
http://en.wikipedia.org/wiki/SQLite#cite_note-1

Businesses\ Users

- The SQLite Consortium is a membership association dedicated to the development of SQLite. Their goals are to keep SQLite both high quality and public domain. Key members include Adobe, Bloomberg, Mozilla and Symbian.

http://www.sqlite.org/index.html

Major Users

- **Adobe** - Uses SQLite in Photoshop and Acrobat\ Adobe reader. The Application file Format of SQLite is used in these products.
- **Apple** - Several functions in Mac OS X use SQLite:

-Apple Mail,
-Safari Web Browser,
-Apeture

- The iPhone and iPod Touch platforms may also contain SQLite implementations (unknown due to closed source nature of those systems.)

http://www.sqlite.org/famous.html

Continued...

- **Mozilla** - Uses SQLite in the Mozilla Firefox Web Browser. SQLite is used in Firefox to store metadata.
- **Google** - Google uses SQLite in Google Desktop and in Google Gears. SQLite is also used in the mobile OS platform, Android.

Continued...

- **McAfee**- Uses SQLite in its various Anti-virus programs
- **Phillips** - Phillips mp3 players use SQLite to store and track metadata. (you can even access the database on the usb based mp3 players that phillips produced)
- **PHP** - Php comes with SQLite 2 and 3 built in. **Python**- SQLite is bundled with the Python programming language.

http://www.sqlite.org/famous.html

Specifications for SQLite

- "SQLite is different from most other SQL database engines in that its primary design goal is to be simple"

- SQLite works well with:
- **Application file format** – transactions guarantee ACID, triggers provide undo/redo feature
- **Temporary data analysis** – command line client, import CSV files and use sql to analyze & generate reports
- **Testing** – stand-in for enterprise DB during application testing (limits potential damage!)
- **Embedded devices** – small, reliable and portable

http://www.sqlite.org/whentouse.html

Disadvantages

- It's Slow

  Locks whole file for writing.

  No caching mechanism of it's own.

- Limited

  Database size restricted to 2GB in most cases.

  Not fully SQL92 compliant.

  Not very scalable

Continued...

- **Portable** - uses only ANSI-standard C and VFS, file format is cross platform.
- **Reliable** – has 100% test coverage, open source code and bug database, transactions are ACID even if power fails
- **Small** – 300 kb library, runs in 16kb stack and 100kb heap

http://www.sqlite.org/about.html
http://www.sqlite.org/testing.html
http://www.sqlite.org/selfcontained.html

Disadvantages

- **High concurrency** – reader/writer locks on the entire file
- **Huge datasets** – DB file can't exceed file system limit or 2TB
- **Access control** – there isn't any

Unique Features.

- No configuration. Just drop in the C library and go.
- No server process to administer or user accounts to manage.
- Easy to backup and transmit database (just copy the file)
- Dynamic typing for column values, variable lengths for column records
- Query can reference multiple database files
- A few non-standard SQL extensions (mostly for conflict resolution)
- http://www.sqlite.org/different.html

Android Developer Fundamentals

Storing Data

Lesson 9

9.1 Shared Preferences

Contents

- Shared Preferences
- Listening to changes

What is Shared Preferences?

- Read and write small amounts of primitive data as key/value pairs to a file on the device storage
- SharedPreference class provides APIs for reading, writing, and managing this data
- Save data in onPause()
  restore in onCreate()

Shared Preferences AND Saved Instance State

- Small number of key/value pairs
- Data is private to the application

Shared Preferences vs. Saved Instance State

- Persist data across user sessions, even if app is killed and restarted, or device is rebooted
- Data that should be remembered across sessions, such as a user's preferred settings or their game score
- Common use is to store user preferences

- Preserves state data across activity instances in same user session
- Data that should not be remembered across sessions, such as the currently selected tab or current state of activity.
- Common use is to recreate state after the device has been rotated

Creating Shared Preferences

- Need only one Shared Preferences file per app
- Name it with package name of your app—unique and easy to associate with app
- MODE argument for getSharedPreferences() is for backwards compatibility—use only MODE_PRIVATE to be secure

getSharedPreferences()

```
private String sharedPrefFile =
    "com.example.android.hellosharedprefs";

mPreferences =
    getSharedPreferences(sharedPrefFile,
                         MODE_PRIVATE);
```

Saving Shared Preferences

- SharedPreferences.Editor interface
- Takes care of all file operations
- put methods overwrite if key exists
- apply() saves asynchronously and safely

SharedPreferences.Editor

```
@Override
protected void onPause() {
    super.onPause();
    SharedPreferences.Editor preferencesEditor =
        mPreferences.edit();
    preferencesEditor.putInt("count", mCount);
    preferencesEditor.putInt("color", mCurrentColor);
    preferencesEditor.apply();
}
```

Restoring Shared Preferences

- Restore in onCreate() in Activity
- Get methods take two arguments—the key, and the default value if the key cannot be found
- Use default argument so you do not have to test whether the preference exists in the file

Getting data in onCreate()

```
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
if (savedInstanceState != null) {
    mCount = mPreferences.getInt("count", 1);
    mShowCount.setText(String.format("%s", mCount));

    mCurrentColor = mPreferences.getInt("color", mCurrentColor);
    mShowCount.setBackgroundColor(mCurrentColor);

    mNewText = mPreferences.getString("text", "");
} else { … }
```

Clearing

- Call clear() on the SharedPreferences.Editor and apply changes

- You can combine calls to put and clear. However, when you apply(), clear() is always done first, regardless of order!

clear()

```
SharedPreferences.Editor preferencesEditor =
                    mPreferences.edit();
preferencesEditor.clear();
preferencesEditor.apply();
```

Listening to Changes

Listening to changes

- Implement interface SharedPreference.OnSharedPreferenceChangeListener
- Register listener with registerOnSharedPreferenceChangeListener()
- Register and unregister listener in onResume() and onPause()
- Implement on onSharedPreferenceChanged() callback

Interface and callback

```
public class SettingsActivity extends AppCompatActivity
    implements OnSharedPreferenceChangeListener { ...

    public void onSharedPreferenceChanged(
        SharedPreferences sharedPreferences, String key) {
        if (key.equals(MY_KEY)) {
            // Do something
        }
    }
}
```

Creating and registering listener

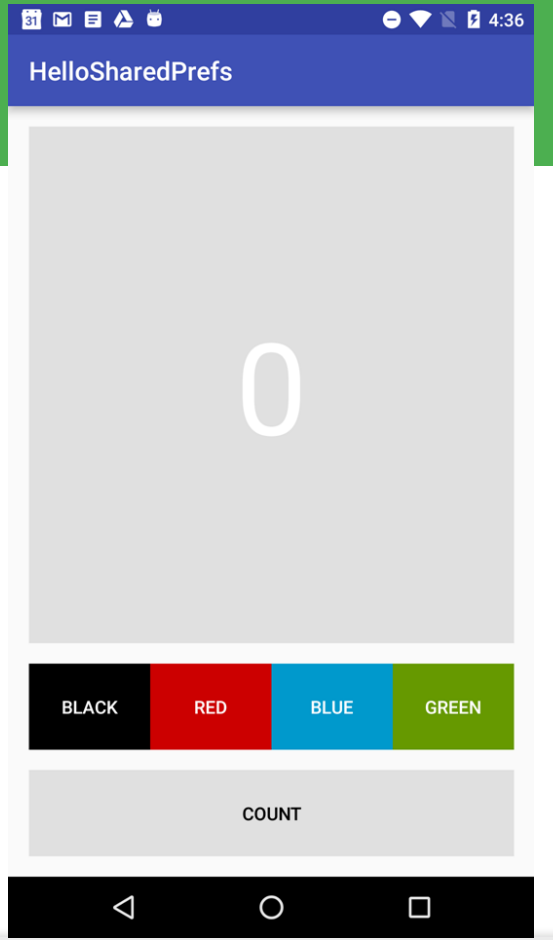
```
SharedPreferences.OnSharedPreferenceChangeListener listener =
    new SharedPreferences.OnSharedPreferenceChangeListener() {
  public void onSharedPreferenceChanged(
     SharedPreferences prefs, String key) {
        // Implement listener here
  }
};
prefs.registerOnSharedPreferenceChangeListener(listener);
```

You need a STRONG reference to the listener

- When registering the listener the preference manager does not store a strong reference to the listener
- You must store a strong reference to the listener, or it will be susceptible to garbage collection
- Keep a reference to the listener in the instance data of an object that will exist as long as you need the listener

Practical: HelloSharedPrefs

- Add Shared Preferences to a starter app
- Add a "Reset" button that clears both the app state and the preferences for the app



Learn more

- Saving Data
- Storage Options
- Saving Key-Value Sets
- SharedPreferences
- SharedPreferences.Editor

Stackoverflow

- How to use SharedPreferences in Android to store, fetch and edit values
- onSavedInstanceState vs. SharedPreferences

What's Next?

- Concept Chapter: 9.1 C Shared Preferences
- Practical: 9.1 P Shared Preferences

END

Android Developer Fundamentals

Settings UI & Preferences Framework

Lesson 9

9.2 Settings UI

Contents

- What are settings?
- Setting screens
- Implement settings
- Default settings
- Save and retrieve settings
- Respond to changes in settings
- Summaries for settings
- SettingsActivity template

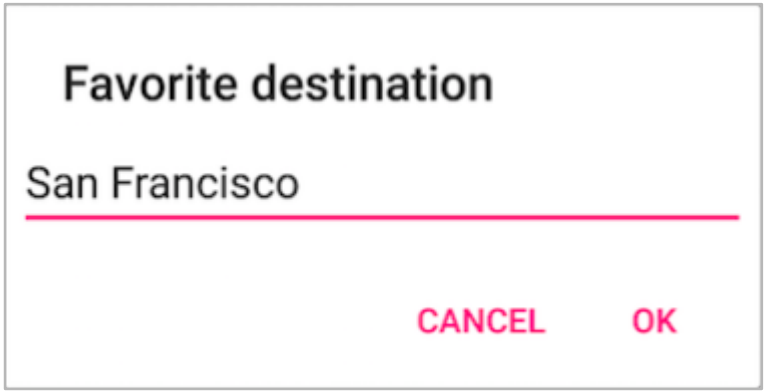
Settings

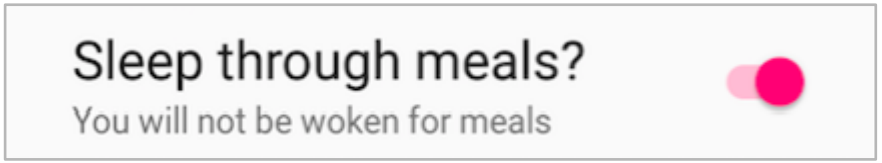
What are app settings?

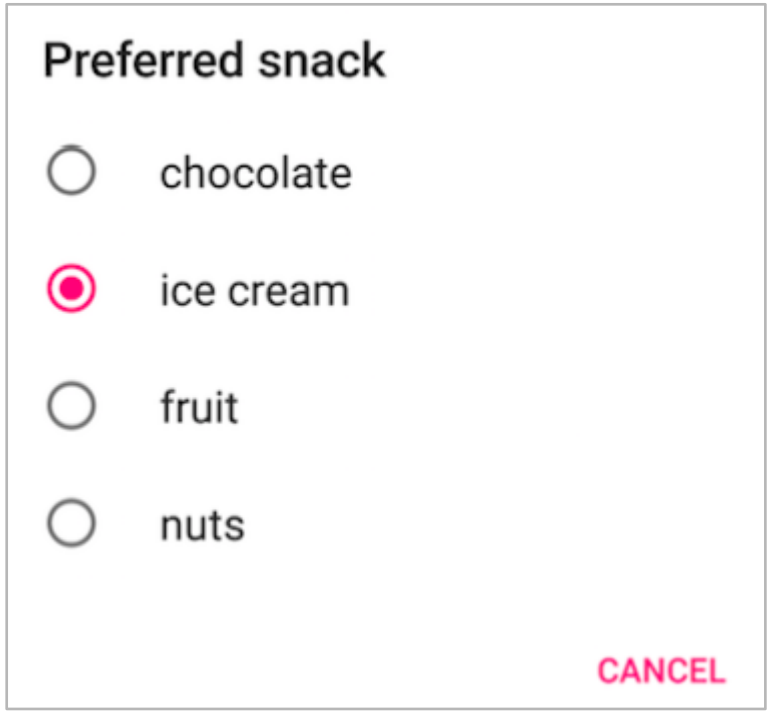
- Users can set features and behaviors of app

  Examples:
  - Home location, defaults units of measurement
  - Notification behavior for specific app
- For values that change infrequently and are relevant to most users
- If values change often, use options menu or nav drawer

Example settings



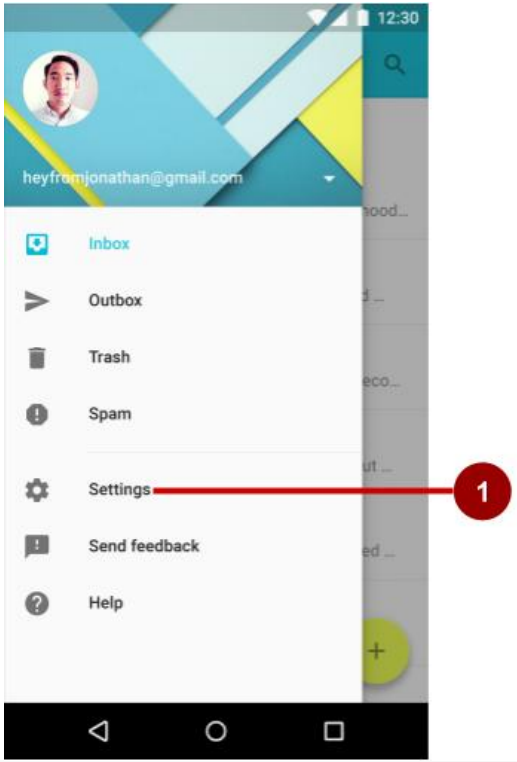


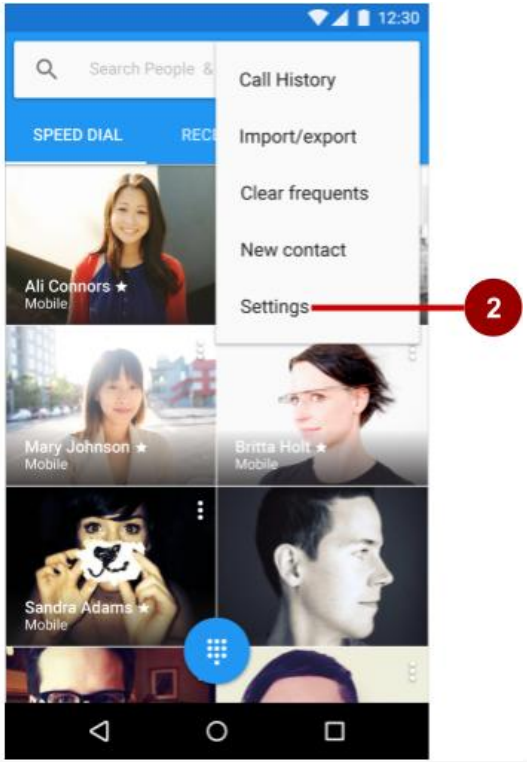


Accessing settings

Users access settings through:

1. Navigation drawer
2. Options menu

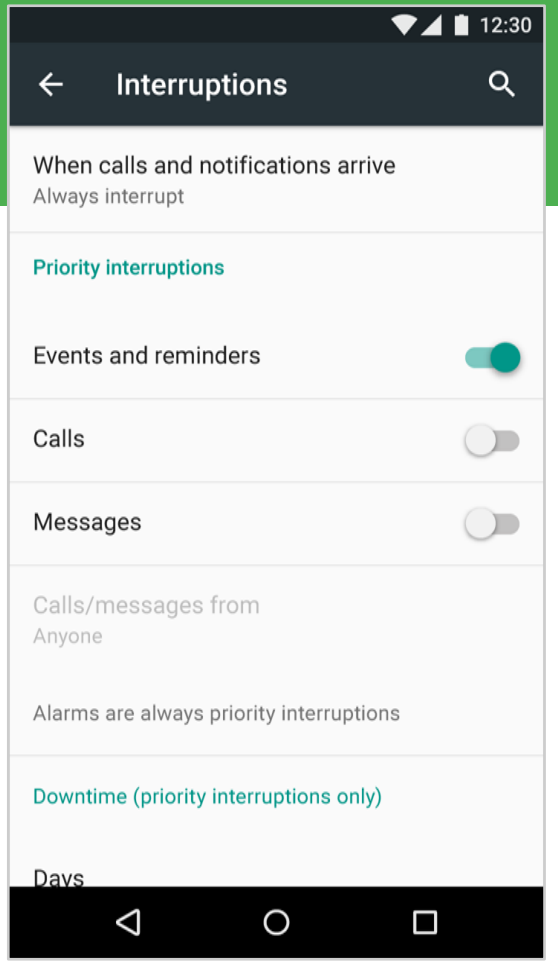




Setting screens

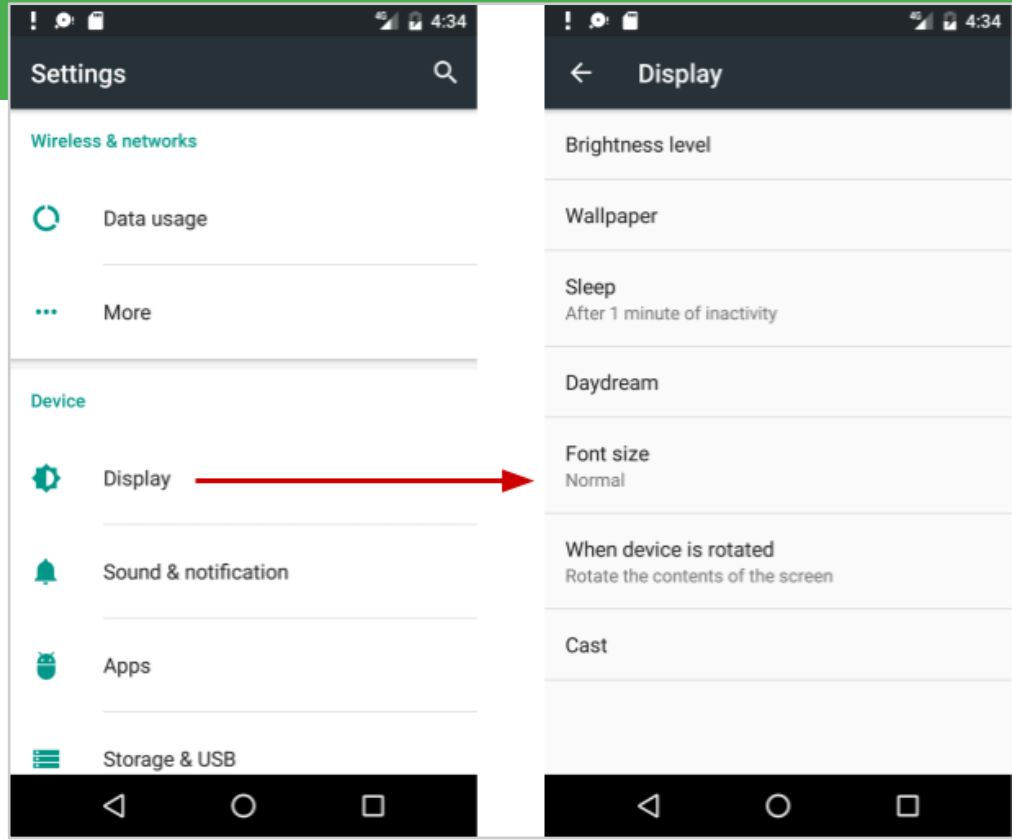
Organize your settings

- Predictable, manageable number of options
- 7 or less: arrange according to priority with most important at top
- 7-15 settings: group related settings under section dividers



16+ Settings

- Group into sub-screens opened from main Settings screen



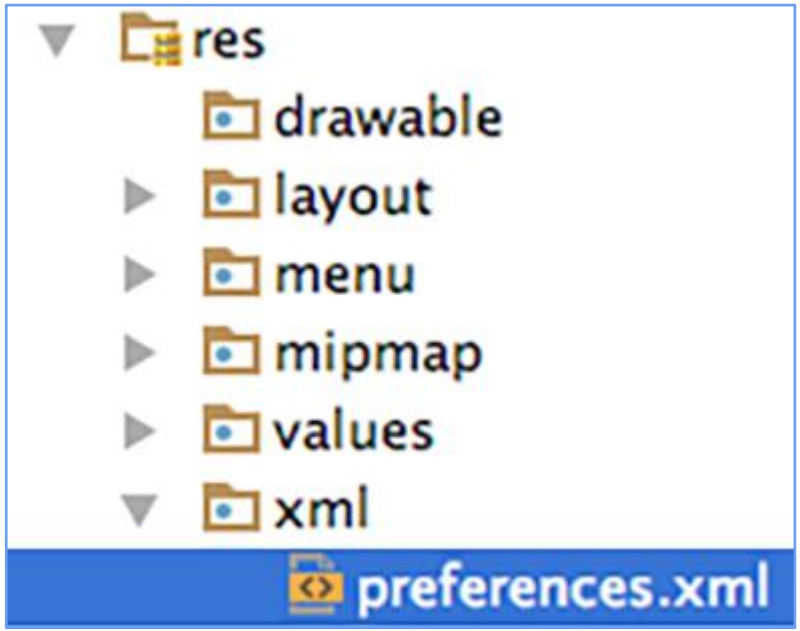
View versus Preference

- Use Preference objects instead of View objects in your Settings screens
- Design and edit Preference objects in the Layout editor just like you do for View objects

Define Settings in a Preference Screen

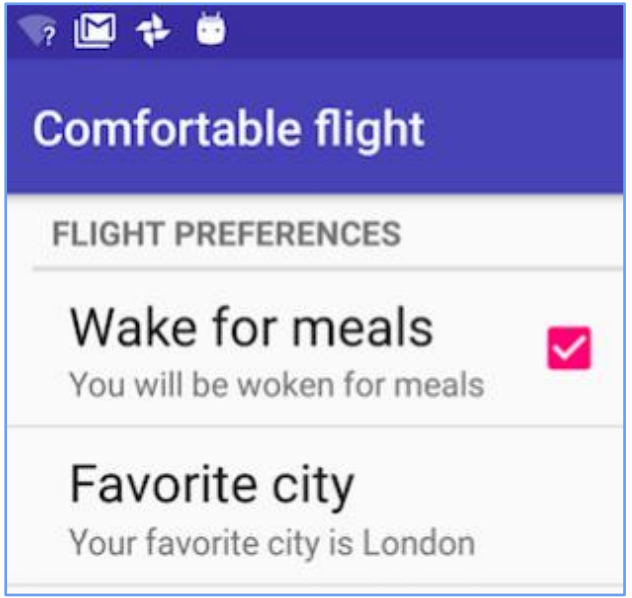
- Define settings in a preferences screen
- It is like a layout
- define in:

  res > xml > preferences.xml



Preference Screen example

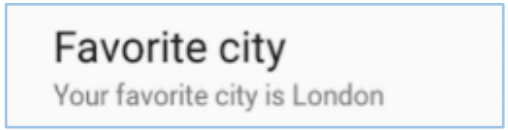
```
<PreferenceScreen>
  <PreferenceCategory
    android:title="Flight Preferences">
    <CheckBoxPreference
       android:title="Wake for meals"
       ... />
    <EditTextPreference
       android:title="Favorite city"
       .../>
  </PreferenceCategory>
</PreferenceScreen>
```



Every Preference must have a key

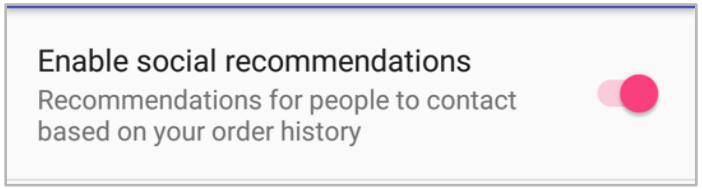
- Every preference must have a key
- Android uses the key to save the setting value

```
<EditTextPreference
    android:title="Favorite city"
    android:key="fav_city"
    … />
```



SwitchPreference

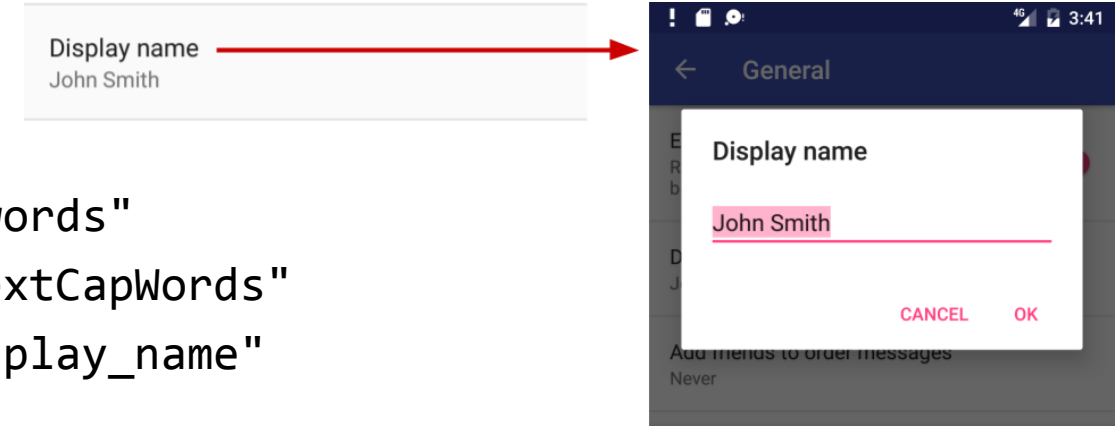
```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android">

    <SwitchPreference
        android:defaultValue="true"
        android:title="@string/pref_title_social"
        android:key="switch"
        android:summary="@string/pref_sum_social" />
</PreferenceScreen>
```



SwitchPreference attributes

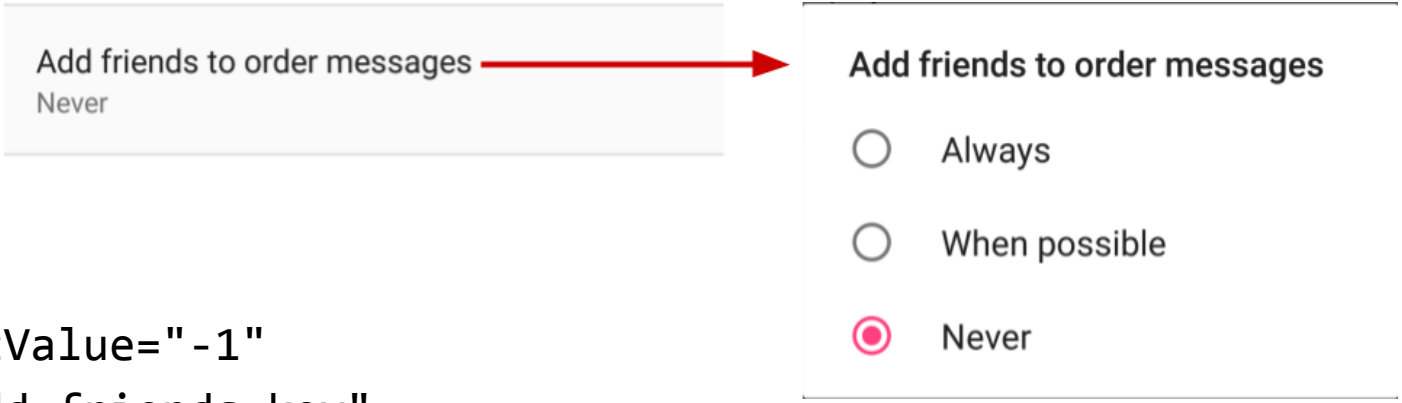
- `android:defaultValue`—true by default
- `android:summary`—text underneath setting, for some settings, should change to reflect value
- `android:title`—title/name
- `android:key`—key for storing value in SharedPreferences

EditTextPreference




```
<EditTextPreference
      android:capitalize="words"
      android:inputType="textCapWords"
      android:key="user_display_name"
      android:maxLines="1"
      android:defaultValue="@string/pref_default_display_name"
      android:title="@string/pref_title_display_name" />
```

ListPreference




```
<ListPreference
  android:defaultValue="-1"
  android:key="add_friends_key"
  android:entries="@array/pref_example_list_titles"
  android:entryValues="@array/pref_example_list_values"
  android:title="@string/pref_title_add_friends_to_messages" />
```

ListPreference

- Default value of -1 for no choice
- `android:entries`—Array of labels for radio buttons
- android:entryValues —Array of values radio button

Preference class

- Preference class provides View for each kind of setting
- associates View with SharedPreferences interface to store/retrieve the preference data
- Uses key in the Preference to store the setting value

Preference subclasses

- CheckBoxPreference—list item that shows a checkbox
- ListPreference—opens a dialog with a list of radio buttons
- SwitchPreference—two-state toggleable option
- EditTextPreference—that opens a dialog with an EditText
- RingtonePreference—lets user to choose a ringtone

Classes for grouping

- PreferenceScreen
  - root of a Preference layout hierarchy
  - at the top of each screen of settings
- PreferenceGroup
  - for a group of settings (Preference objects).
- PreferenceCategory
  - title above a group as a section divider

Implement settings

Settings UI uses fragments

- Use an Activity with a Fragment to display the Settings screen
- Use specialized Activity and Fragment subclasses that handle the work of saving settings

Activities and fragments for settings

- Android 3.0 and newer:
  - AppCompatActivity with PreferenceFragmentCompat
  - OR use Activity with PreferenceFragment
- Android older than 3.0 (API level 10 and lower):
  - build a special settings activity as an extension of the PreferenceActivity class (use the template!)

Lesson focusses on this!

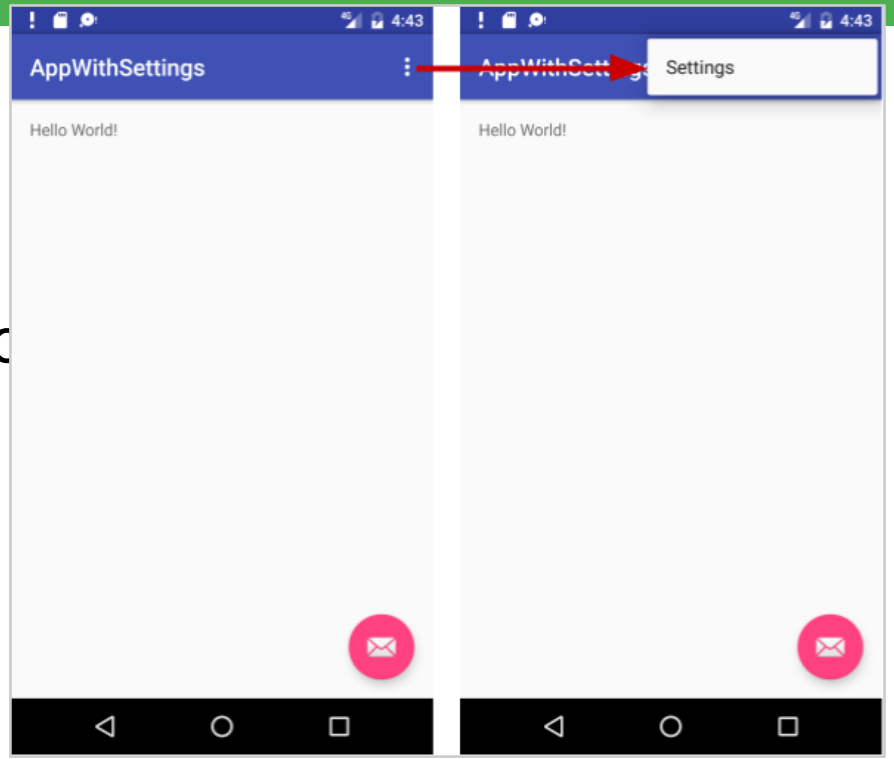
Steps to implement Settings

For AppCompatActivity with PreferenceFragmentCompat:

- Create the preferences screen
- Create an activity for the settings
- Create a fragment for the settings
- Add the preferenceTheme to the AppTheme
- Add code to invoke Settings UI

Basic Activity template

- Basic Activity template
  Includes options menu
- **Settings** menu item provided fo
  options menu



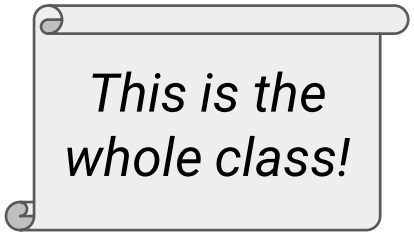
Create a Settings Activity subclass

- Extends AppCompatActivity
- in onCreate() display the settings fragment:

```
getSupportFragmentManager()
    .beginTransaction()
    .replace(android.R.id.content,
             new MySettingsFragment())
    .commit();
```

Settings Activity example

```
public class MySettingsActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getSupportFragmentManager().beginTransaction()
            .replace(android.R.id.content, new MySettingsFragment())
            .commit();
    }
}
```



Create a Settings Fragment subclass

- Extends PreferenceFragmentCompat
- Implement methods:
  - onCreatePreferences() displays the settings
  - setOnPreferenceChangeListener() handles any changes that need to happen when the user changes a preference (optional)

PreferenceFragment

```
public class MySettingsFragment
    extends PreferenceFragmentCompat { …}
```

- Blank fragments include onCreateView() by default
- Replace onCreateView() with onCreatePreferences() because this fragment displays a preferences screen

Settings Fragment example

```
public class MySettingsFragment extends PreferenceFragmentCompat {

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

Add PreferenceTheme to app's theme

If using PreferenceFragmentCompat, set preferenceTheme in styles.xml:

```
<style name="AppTheme" parent="...">
    ...
    <item name="preferenceTheme">
        @style/PreferenceThemeOverlay
    </item>
  …
</style>
```

Invoke Settings UI

Send the Intent to start the Settings Activity:

- From Options menu, update onOptionItemsSelected()
- From Navigation drawer, update onItemClick() on the OnItemClickListener given to setOnItemClickListener

Default Settings

Default settings

- Set default to value most users would choose
  - All contacts
- Use less battery power
  - Bluetooth is off until the user turns it on
- Least risk to security and data loss
  - Archive rather than delete messages
- Interrupt only when important
  - When calls and notifications arrive

Set default values

- Use android:defaultValue in Preference view in xml:

```
<EditTextPreference
    android:defaultValue="London"
   … />
```

- In main activity's onCreate(), save default values.

Save default values in shared preferences

In onCreate() of MainActivity

```
PreferenceManager.setDefaultValues(
        this, R.xml.preferences, false);
```

- App context, such as this
- Resource ID of XML resource file with settings
- `false` only calls method the first time the app starts

Save and retrieve settings

Saving setting values

- No need to write code to save settings!
- If you use specialized Preference Activity and Fragment, Android automatically saves setting values in shared preferences

Get settings from shared preferences

- In your code, get settings from default shared preferences
- Use key as specified in preference view in xml

```
SharedPreferences sharedPref =
    PreferenceManager.getDefaultSharedPreferences(this);

String destinationPref =
     sharedPref.getString("fav_city", "Jamaica");
```

Get settings values from shared preferences

- In preference definition in xml:

```
<EditTextPreference
    android:defaultValue="London"
    android:key="fav_city" />
```

- In code, get fav_city setting:

```
String destinationPref =
    sharedPref.getString("fav_city", "Jamaica");
```

default setting value

is different than

default value returned by pref.getString() if key is not found in shared prefs

Respond to changes in settings

Listening to changes

- Display related follow-up settings
- Disable or enable related settings
- Change the summary to reflect current choice
- Act on the setting

  for example, if the setting changes the screen background, then change the background

Listen for changes to settings

- Define setOnPreferenceChangeListener()
- in onCreate() in the Settings Fragment

onCreatePreferences() example

```
@Override
public void onCreatePreferences(Bundle savedInstanceState,
                                String rootKey) {
   setPreferencesFromResource(R.xml.preferences, rootKey);
   ListPreference colorPref =
                (ListPreference) findPreference("color_pref");
   colorPref.setOnPreferenceChangeListener(
     // see next slide
     // ...);
}
```

onPreferenceChangeListener() example

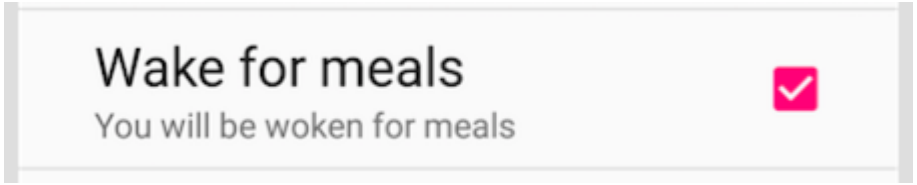
Example: change background color when setting changes

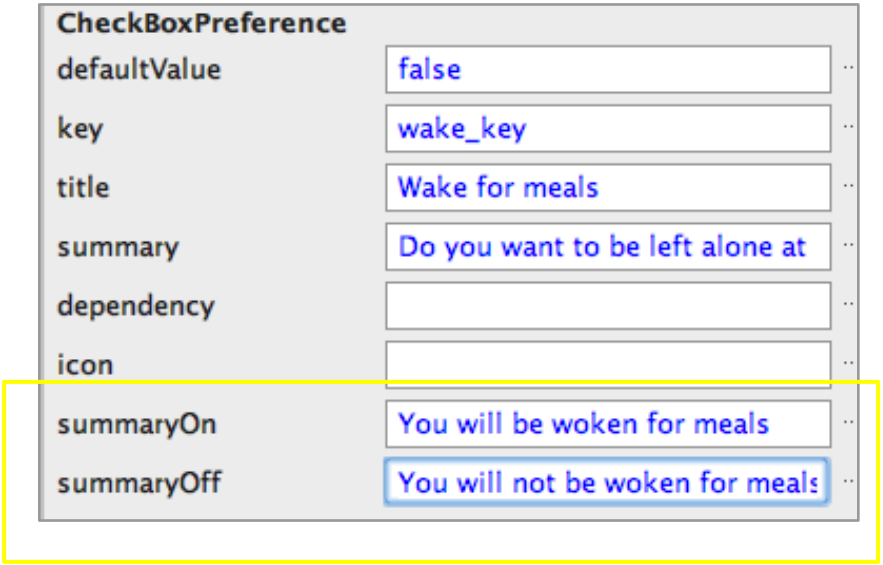
```
colorPref.setOnPreferenceChangeListener(
    new Preference.OnPreferenceChangeListener(){
        @Override
        public boolean onPreferenceChange(
           Preference preference, Object newValue){
               setMyBackgroundColor(newValue);
               return true;
        }
});
```

Summaries for settings

Summaries for true/false values

Set attributes to define conditional summaries for preferences that have true/false values

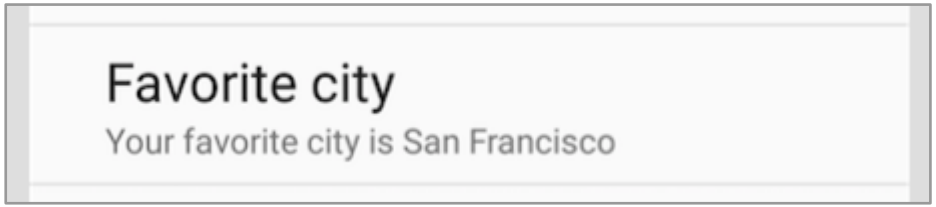


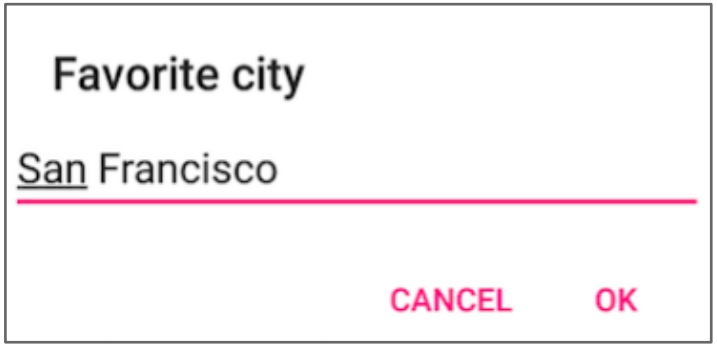


Summaries for other settings

For settings that have values other than true/false, update the summary when the setting value changes

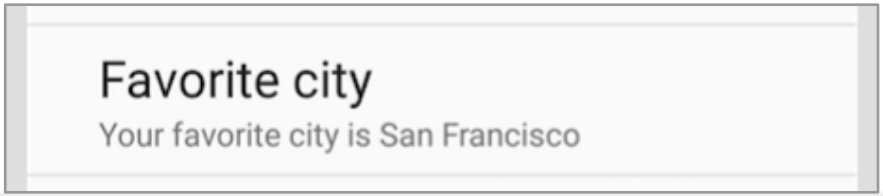
- Set the summary in onPreferenceChangeListener()





Set summary example

```
EditTextPreference cityPref = (EditTextPreference)
                                findPreference("fav_city");
cityPref.setOnPreferenceChangeListener(
  new Preference.OnPreferenceChangeListener(){
    @Override
    public boolean onPreferenceChange(Preference pref, Object value){
      String city = value.toString();
      pref.setSummary("Your favorite city is " + city);
      return true;
    }
  });
```



Activity Template

More complex?

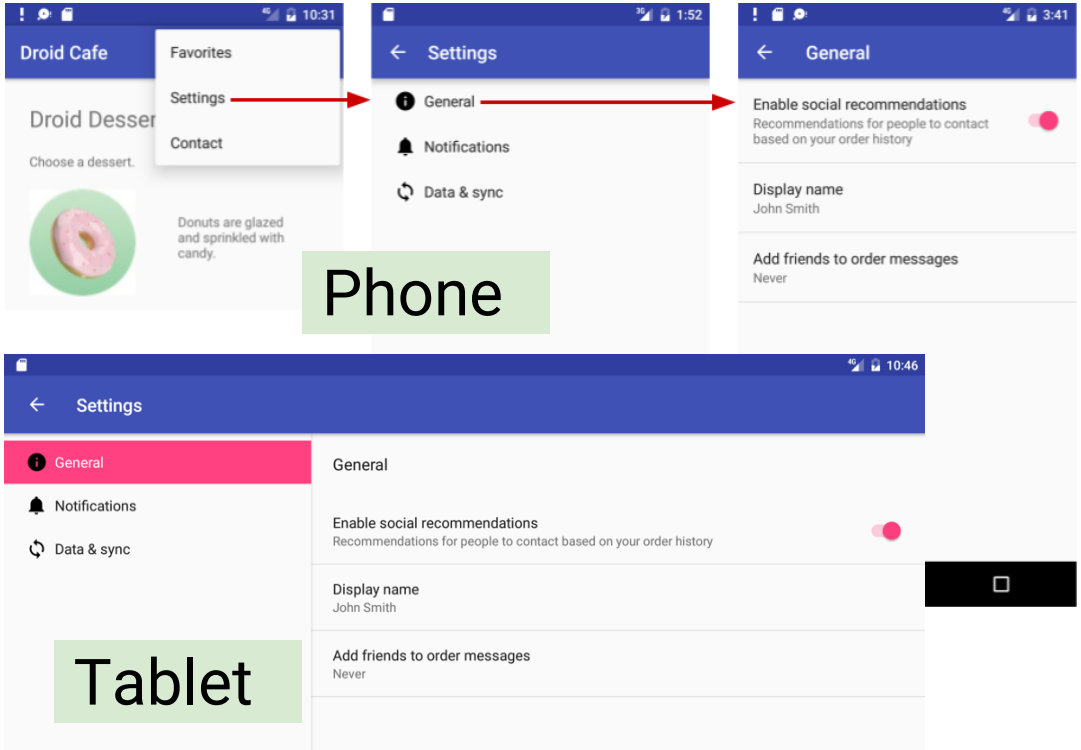
For anything more complex

?

use the SettingsActivity template!

Settings Activity template

- Complex Settings
- Backwards compatibility
- Customize pre-populated settings
- Adaptive layout for phones and tablets



Learn more

Learn more

- Android Studio User Guide
- Settings (coding)
- Preference class
- PreferenceFragment
- Fragment
- SharedPreferences
- Saving Key-Value Sets
- Settings (design)

What's Next?

- Concept Chapter: 9.2 C App Settings
- Practical: 9.2 P Adding Settings to an App

END

Android Developer Fundamentals

Storing Data

Lesson 10

10.1 SQLite Primer

Contents

- SQLite Database
- Queries

This is only a refresher

This course assumes that you are familiar with

- Databases in general
- SQL databases in particular
- SQL query language

This chapter is a refresher and quick reference

SQLite Database

SQL Databases

- Store data in tables of rows and columns (spreadsheet…)
- Field = intersection of a row and column
- Fields contain data, references to other fields, or references to other tables
- Rows are identified by unique IDs
- Column names are unique per table

SQLite software library

Implements SQL database engine that is

- self-contained (requires no other components)
- serverless (requires no server backend)
- zero-configuration (does not need to be configured for your application)
- transactional (changes within a single transaction in SQLite either occur completely or not at all)

What is a transaction?

A transaction is a sequence of operations performed as a single logical unit of work.

A logical unit of work must have four properties

- atomicity
- consistency
- isolation
- durability

All or nothing

All changes within a single transaction in SQLite either occur completely or not at all, even if the act of writing the change out to the disk is interrupted by

- program crash
- operating system crash
- power failure.

ACID

- **Atomicity**—All or no modifications are performed
- **Consistency**—When transaction has completed, all data is in a consistent state
- **Isolation**—Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions
- **Durability**—After a transaction has completed, its effects are permanently in place in the system

Queries

SQL basic operations

- Insert rows
- Delete rows
- Update values in rows
- Retrieve rows that meet given criteria

SQL Query

- SELECT word, description
  FROM WORD_LIST_TABLE
  WHERE word="alpha"

Generic

- SELECT columns
  FROM table
  WHERE column="value"

SELECT columns FROM table

- **SELECT columns**
  - Select the columns to return
  - Use * to return all columns

- **FROM table**—specify the table from which to get results

WHERE column="value"

- **WHERE**—keyword for conditions that have to be met

- **column="value"**—the condition that has to be met
  - common operators: =, LIKE, <, >

AND, ORDER BY, LIMIT

SELECT _id FROM WORD_LIST_TABLE WHERE word="alpha" **AND** definition LIKE "%art%" **ORDER BY word DESC LIMIT 1**

- **AND, OR**—connect multiple conditions with logic operators
- **ORDER BY**—omit for default order, or ASC for ascending, DESC for descending
- **LIMIT**—get a limited number of results

Sample queries

1	SELECT * FROM WORD_LIST_TABLE	Get the whole table
2	SELECT word, definition FROM WORD_LIST_TABLE WHERE _id > 2	Returns [["alpha", "particle"]]

More sample queries

3	SELECT _id FROM WORD_LIST_TABLE WHERE word="alpha" AND definition LIKE "%art%"	Return id of word alpha with substring "art" in definition `[["3"]]`
4	SELECT * FROM WORD_LIST_TABLE ORDER BY word DESC LIMIT 1	Sort in reverse and get first item. Sorting is by the first column (_id) `[["3","alpha","particle"]]`

Last sample query

5	SELECT * FROM WORD_LIST_TABLE LIMIT 2,1	Returns 1 item starting at position 2. Position counting starts at 1 (not zero!). Returns `[["2","beta","second letter"]]`

rawQuery()

```
String query = "SELECT * FROM WORD_LIST_TABLE";
rawQuery(query, null);

query = "SELECT word, definition FROM
WORD_LIST_TABLE WHERE _id> ? ";

String[] selectionArgs = new String[]{"2"}
rawQuery(query, selectionArgs);
```

query()

```
SELECT * FROM
WORD_LIST_TABLE
WHERE word="alpha"
ORDER BY word ASC
LIMIT 2,1;
```

Returns:

```
[["alpha",
"particle"]]
```

```
String table = "WORD_LIST_TABLE"
String[] columns = new String[]{"*"};
String selection = "word = ?"
String[] selectionArgs = new String[]{"alpha"};
String groupBy = null;
String having = null;
String orderBy = "word ASC"
String limit = "2,1"

query(table, columns, selection, selectionArgs,
groupBy, having, orderBy, limit);
```

Cursors

Queries always return a Cursor object

Cursor is an object interface that provides random read-write access to the result set returned by a database query

⇒ Think of it as a pointer to table rows

You will learn more about cursors in the following chapters

Learn more

- SQLite website
- Full description of the Query Language
- SQLite class
- Cursor class
- Practice: HeadFirst Labs

What's Next?

- Concept Chapter: 10.1 C SQLite Primer
- Practical: --

END

Android Developer Fundamentals

Content Providers

Lesson 11

11.1 Content Providers

Share data with other apps

Contents

- What is a ContentProvider
- App with Content Provider Architecture
- Implementation
  - Contract
  - ContentProvider
  - Manifest Permissions
  - Content Resolver

What is a Content Provider

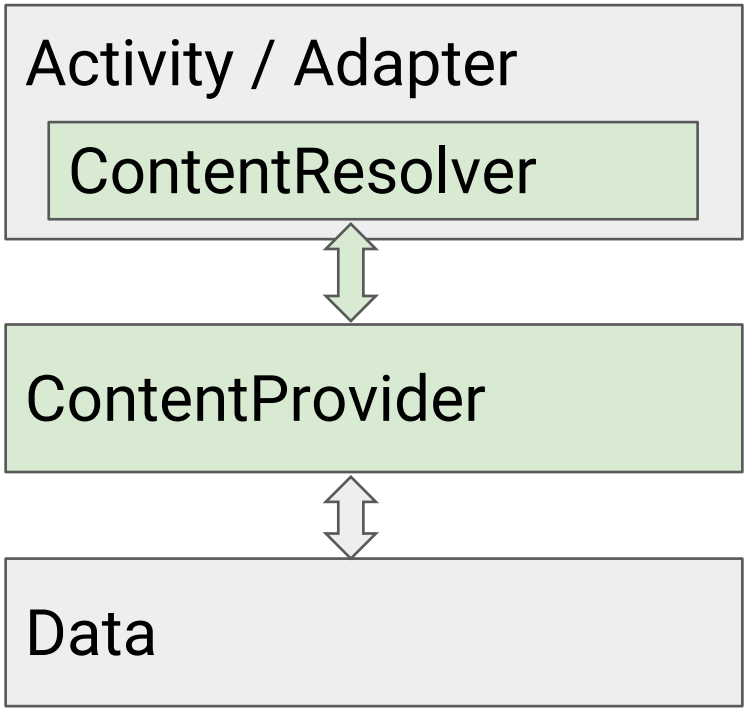
What is a Content Provider?

- A content provider is a component fetches data that the app requests from a repository
- The app doesn't need to know where or how the data is stored, formatted, or accessed

What is a Content Resolver?

- A content resolver is a component that your app uses to send requests to a content provider
- Requests consist of a content URI and an SQL-like query
- The ContentResolver object provides query(), insert(), update(), and delete() methods

How do they work together?



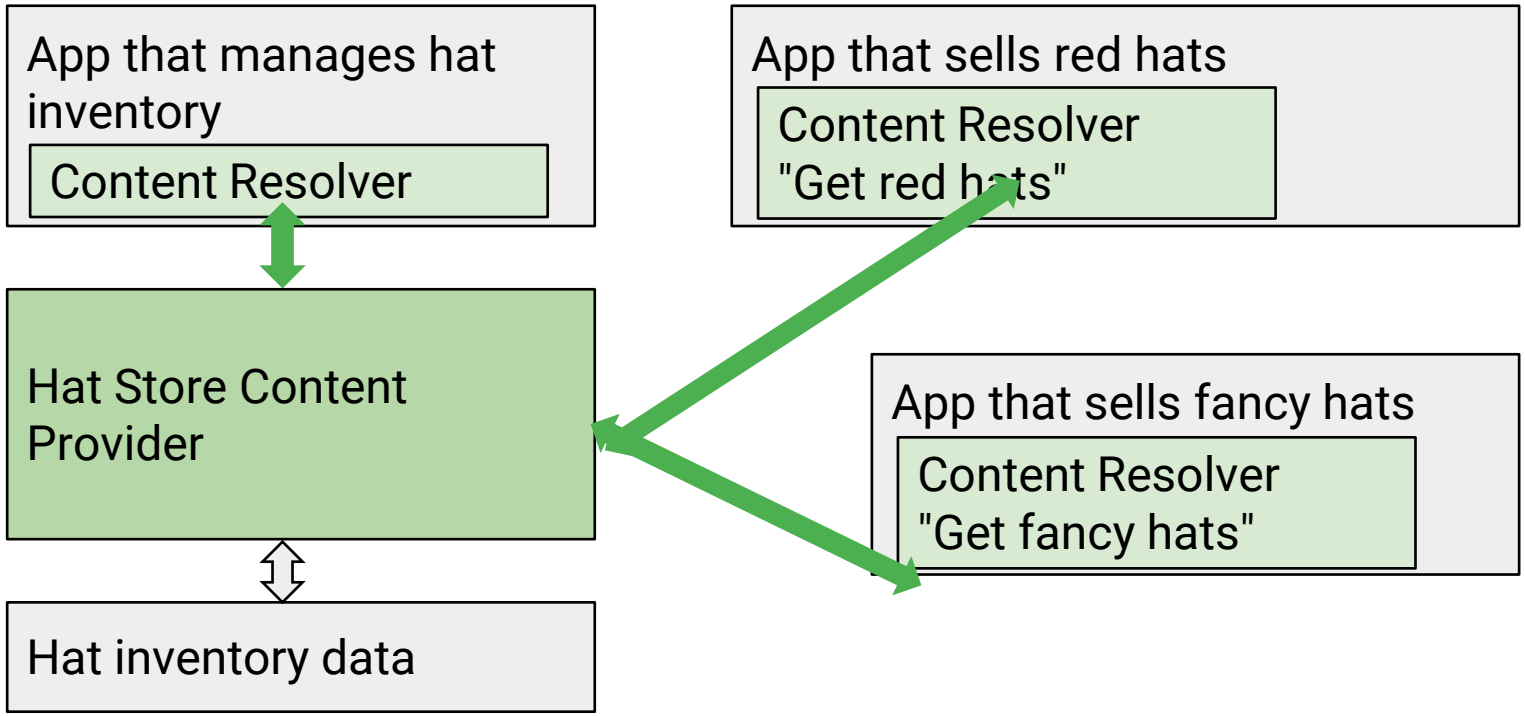
1. Activity/Adapter uses ContentResolver to query ContentProvider
2. ContentProvider gets data
3. ContentResolver returns data as Cursor
4. Activity/Adapter uses data

What is it good for?

- Securely make data available to other apps
- Manage access permissions to app data

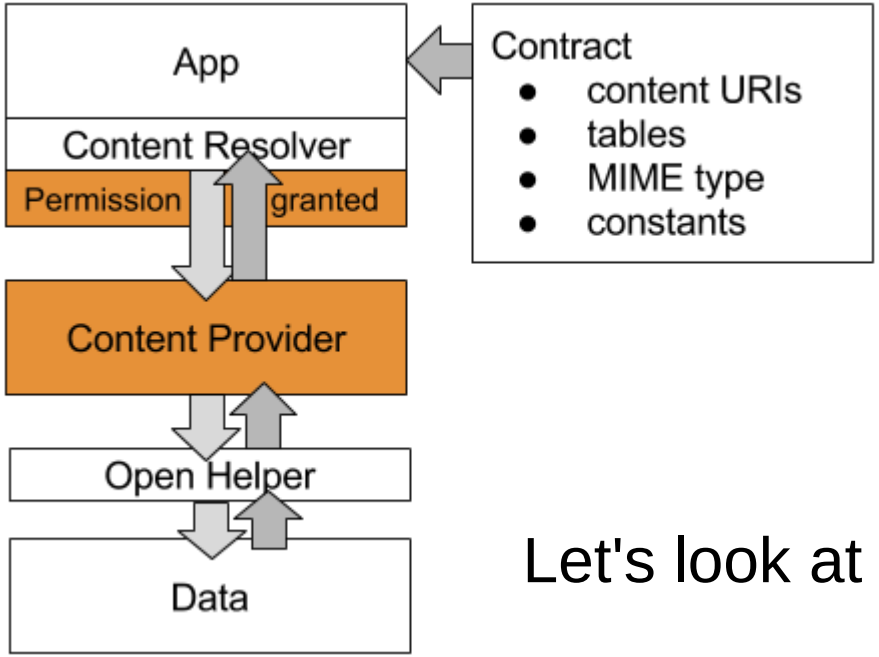
- Store data or develop backend independently from UI
- Standardized way of accessing data
- Required to work with CursorLoaders

Many apps can use one content provider



App with Content Provider

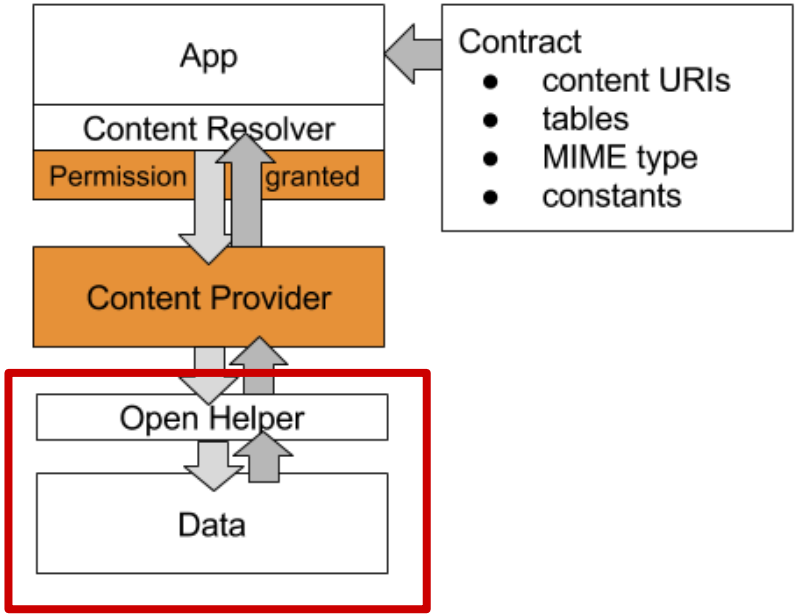
Components of a Content Provider App



Let's look at each of these...

Data Repository

- Data, commonly in SQLite Database but can be any backend
- OpenHelper if you use SQLite database

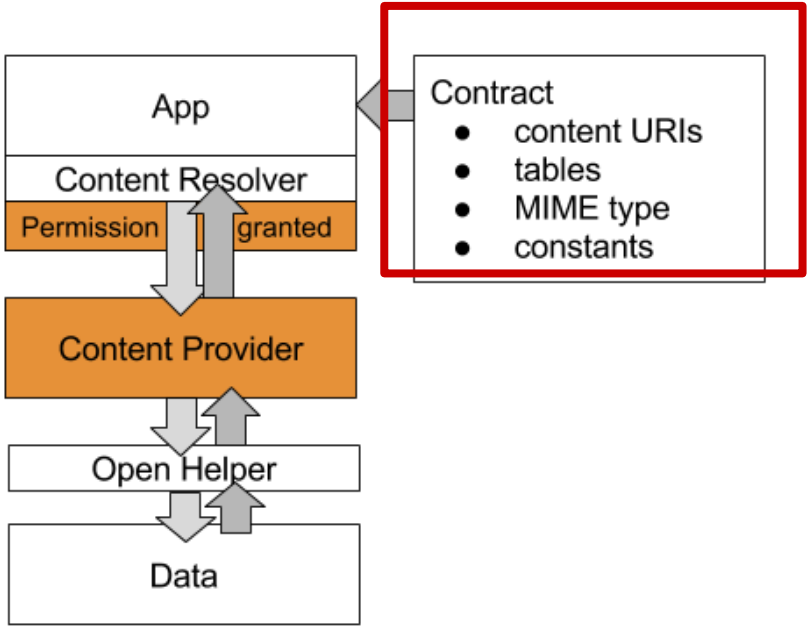


Contract

Public class that exposes information about the content provider to other apps

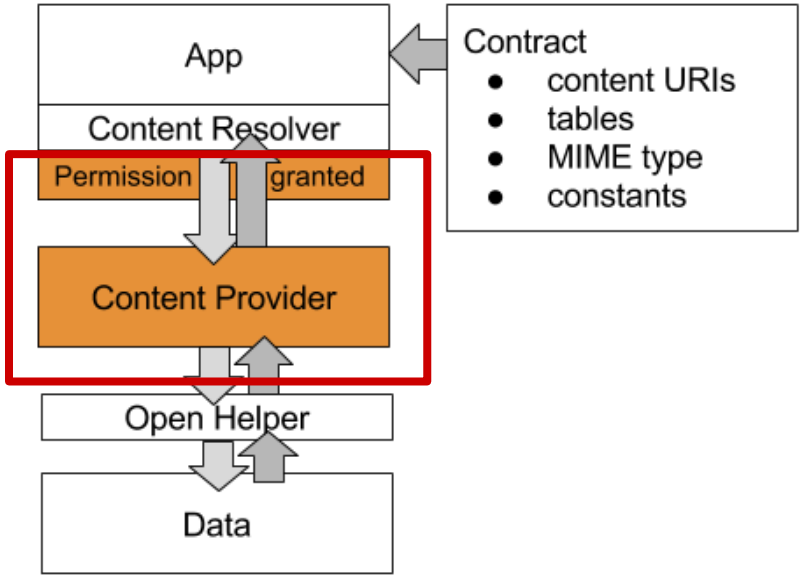
Contract specifies

- URIs to query data
- Table structure of data
- MIME type of returned data
- Constants



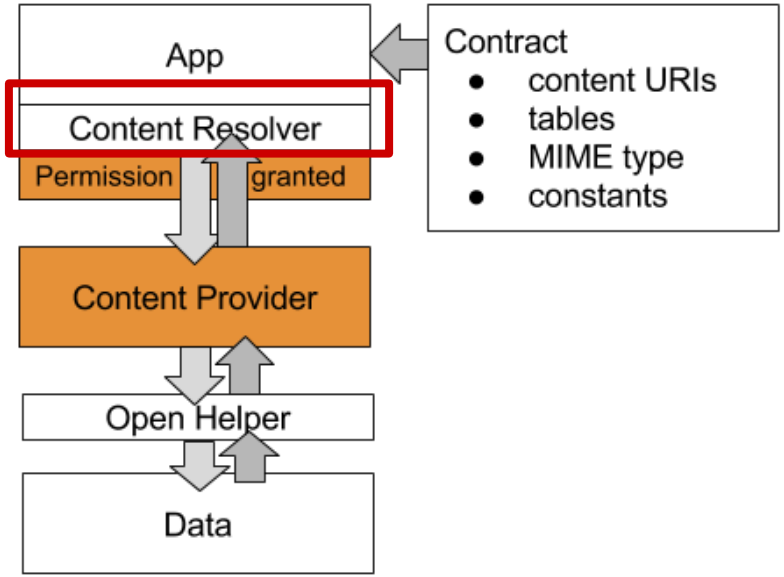
Content Provider

- Public secure interface
- Permissions control access
- Subclass of ContentProvider
- query(), insert(), delete(), update() API



Content Resolver

- Send requests to content provider and return response



Implementation

Implementing a Content Provider

- Data—Commonly an SQLite database
- Write contract—Information about the content provider
- Subclass ContentProvider and implement methods
- Get data using ContentResolver
- Set permissions in Android Manifest

Why SQLite Database?

- Represents data in tables
- Supports same operations as content provider
- Returns requested data as cursor
- OpenHelper class to simplify management

Contract

What's in a Contract?

- Public class that documents content provider API
- Called Contract by convention
- Contains
  - Content URIs and URI scheme to query data
  - Table and column names for returned data
  - MIME types to help process returned data
  - Shared constants to make life easier

Content URI

- `http://` and `file://` are URIs for web pages and files
- Content URI is path to data and uses `content://`

E.g. request all the entries in the "words" table

```
content://com.android.example.wordcontentprovider.provider/words
```

General form of a URI

```
scheme://authority/path/id
```

- **scheme** is always `content://` for content URIs
- **authority** represents the domain, and for content providers customarily ends in `.provider`
- **path** is the path to the data
- **id** uniquely identifies the data set to search

URI Scheme

By convention, provide constants for

- `AUTHORITY`–Domain
- `CONTENT_PATH`–Path to the data
- `CONTENT_URI`–URI to one set of data

URI Scheme in Code

```
public static final String AUTHORITY =
"com.android.example.minimalistcontentprovider.provider";

public static final String CONTENT_PATH =  "words";

public static final Uri CONTENT_URI = Uri.parse("content://" +
AUTHORITY + "/" + CONTENT_PATH);
```

Table definitions

```
public static final String DATABASE_NAME = "wordlist";

public static abstract class WordList implements BaseColumns {
    public static final String WORD_LIST_TABLE = "word_entries";
    // Column names...
    public static final String KEY_ID = "_id";
    public static final String KEY_WORD = "word";
}
```

MIME Type

- Format of returned data
- text/html for web pages, application/json for JSON data
- App calls `getType()` to get MIME type from provider
- Use Android's vendor-specific format for your content providers MIME type

Android's vendor-specific MIME Type

`type.subtype/provider-specific-part`

- Type: **`vnd`**
- Subtype
  - If URI pattern is for a single row: **`android.cursor.item/`**
  - If URI pattern is for more than one row: **`android.cursor.dir/`**
- Provider-specific part: **`vnd.<name>.<type>`**
  - <name>: globally unique, such as company or package name
  - <type> unique to corresponding URI pattern, such as table name

MIME Type Example

Multiple words

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

One word

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

MIME Type code in Contract

```
static final String SINGLE_RECORD_MIME_TYPE =
    "vnd.android.cursor.item/vnd.com.example.provider.words";

static final String MULTIPLE_RECORDS_MIME_TYPE =
        "vnd.android.cursor.item/vnd.com.example.provider.words";
```

getType()

```
@Override
public String getType(Uri uri) {
    switch (sUriMatcher.match(uri)) {
        case URI_ALL_ITEMS_CODE:
            return MULTIPLE_RECORDS_MIME_TYPE;
        case URI_ONE_ITEM_CODE:
            return SINGLE_RECORD_MIME_TYPE;
        default:
            return null;
    }
}
```

Constants

- Constants that are used by multiple classes in an app
- Convenience constants for client use
- Encapsulate parameters whose values might change as constants, so that if the content provider changes, the clients don't break

ContentProvider

Extend ContentProvider

```
public class WordListContentProvider
                        extends ContentProvider {}
```

Implement methods

- query(), insert(), delete(), and update() methods
- interact with the data backend …
- … such as an Open Helper

Example insert()

```
/**
     * Inserts one record.
     *
     * @return URI for the newly created entry.
     */
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        long id = mDB.insert(values);
        return Uri.parse(CONTENT_URI + "/" + id);
    }
```

Manifest Permissions

Set Permissions!

- By default, with no permissions set explicitly, any other app can access a content provider for reading and writing
- Set read or write permissions in AndroidManifest
- Use unique tags that include package name

Provider in Android Manifest

```
<provider
android:name=".WordListContentProvider"
android:authorities=
    "com.android.example.wordlistsqlwithcontentprovider.provider"
android:exported="true" />
```

Permissions inside <provider>

```
android:readPermission=
    "com.android.example.wordlistsqlwithcontentprovider.PERMISSION"

android:writePermission=
    "com.android.example.wordlistsqlwithcontentprovider.PERMISSION"
```

Client permissions

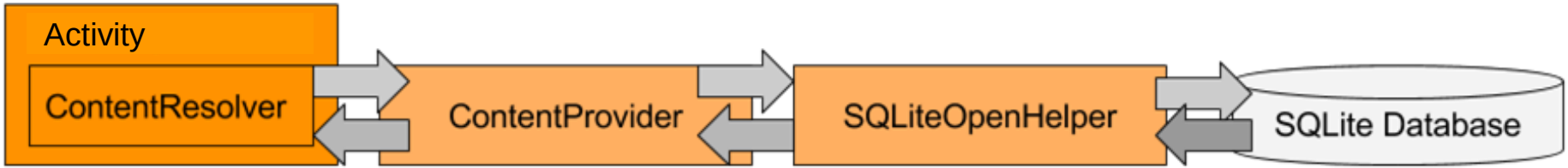
- Granted by the user

```
<uses-permission android:name =
 "com.android.example.wordlistsqlwithcontentprovider.PERMISSION"/>
```

Content Resolver

Content Resolver

- Must use a ContentResolver to send requests as queries to the content provider
- Data is returned in a Cursor object, as rows and columns



Content Resolver methods

- ContentResolver.query()
- ContentResolver.insert()
- ContentResolver.delete()
- ContentResolver.update()

getContentResolver.query()

```
public Cursor query(
    Uri uri,
    String[] projection,
    String selection,
    String[] selectionArgs,
    String sortOrder){ … // implementation }
```

Calling getContentResolver.query()

```
Cursor cursor = getContentResolver().query(
    Uri.parse(queryUri), projection, selectionClause,
    selectionArgs, sortOrder);
```

Query parameters (recap)

```
uri: String queryUri = Contract.CONTENT_URI.toString();

projection: String[] projection =
                    new String[] {Contract.CONTENT_PATH};

selection: String where = KEY_WORD + " LIKE ?";

selectionArgs: String[]whereArgs = new String[]{searchString};

sortOrder: > null for default, ASC / DESC
```

Recap

Summary of implementing content provider

- Data, for example, in a database
- A way for accessing the backend, for example, through an open helper
- Declare content provider in Android Manifest and set permissions
- Extend ContentProvider and implement query(), insert(), delete(), update(), count(), and getType() methods
- Create public Contract class to expose URI scheme, table names, MIME type, and important constants to other classes and apps
- Use a ContentResolver to send requests to content provider
    - Process data returned as cursor

Practicals Info

- Minimalist Content Provider to show you mechanics
- Add a content provider to the WordList app for a more realistic example
- Create a separate client app that accesses the content provider of the WordList app

Learn more

- Uniform Resource Identifiers or URIs
- MIME type
- MatrixCursor and Cursors
- Content Providers

Videos

- Android Application Architecture
- Android Application Architecture: The Next Billion Users

What's Next?

- Concept Chapter
- 11.1 C Content Providers
- Practicals
  - 11.1A P Minimalist Content Provider
  - 11.1B P Add a Content Provider to WorldListSQL
  - 11.1C P Sharing Content with Other Apps

END

Android Developer Fundamentals

Loaders

Lesson 12

12.1 Loaders

Load data in the background

Contents

- Loaders Recap
- Implementing a CursorLoader

Loaders Recap

Give Users Data FAST

- Research shows that if an app or page takes more than 3 seconds to load, 40% of users will abandon it

- The solution is to load most or all of your data in the background

What Are Loaders?

- Special purpose classes that manage loading data asynchronously in the background
- Introduced in Android 3.0
- Uses AsyncTask

Loader features

- Provides asynchronous loading of data
- Monitors the source of data and deliver new results when the content changes
- Automatically reconnects to the loader's last cursor when recreated after a configuration change
- Available to every Activity and Fragment

Types of Loaders

- AsyncTaskLoader keeps data available through configuration changes
- CursorLoader works with content providers and databases
- Rarely necessary to build your own loader

CursorLoader with content provider

Instead of using a content resolver's query() call to fetch data from the content provider ...

1. Create a loader and supply it with a content URI
2. Update the data displayed when the loader returns

Implementing CursorLoader

Steps to using a CursorLoader

1. Activity or Fragment that implements LoaderManager.LoaderCallbacks
2. Create CursorLoader with content URI in **onCreateLoader()**
3. Implement **onLoadFinished()** to display the data
4. Implement **onLoaderReset()** to clear date
5. Create instance of LoaderManager in onCreate()

Implement LoaderManager.LoaderCallbacks<>

```
public class MainActivity extends AppCompatActivity
    implements LoaderManager.LoaderCallbacks<Cursor>
```

onCreateLoader()

```
@Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        String queryUri = CONTENT_URI.toString();
        String[] projection = new String[] {CONTENT_PATH};
        return new CursorLoader(this, Uri.parse(queryUri),
                projection, null, null, null);
    }
```

CursorLoader constructor

```
return new CursorLoader(this, Uri.parse(queryUri),
                        projection, null, null, null);
```

- uri — URI for the content to retrieve
- projection — List of columns to return
- selection — Which rows to return (WHERE clause)
- selectionArgs — Values for selection
- sortOrder — Order of rows

onLoadFinished()

- Data is delivered as a Cursor object
- Do something with the loaded data

```
@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    // Update the adapter to display data
    mAdapter.setData(cursor);
}
```

setData() in the adapter

```
public void setData(Cursor cursor) {
    mCursor = cursor;
    notifyDataSetChanged();
}
```

onLoaderReset()

- Called when a previously created loader is being reset
- Clean all references to the data

```
@Override
public void onLoaderReset(Loader<Cursor> loader) {
    // Clear data in UI
    mAdapter.setData(null);
}
```

Get a LoaderManager

- LoaderManager manages all your loaders
- Registers an observer that receives callbacks when data changes
- There is only one

LoaderManager methods

```
// Get a loader manager and initialize a loader
getLoaderManager().initLoader(0, null, this);

// Restart the loader, discarding all data
getLoaderManager().restartLoader(0, null, this);
```

- First argument is loader id
- Second argument is an optional Bundle of data
- Third argument is the context for the callbacks

More callbacks in Loader

When caching data or using a data observer, you may also need to override these methods—basic Example

- `onStopLoading()`
- `onReset()`
- `onCanceled()`
- `deliverResult(D results)`

Learn more

- Loaders
- Running a query with a CursorLoader
- CursorLoader class

What's Next?

- Concept Chapter: 12.1 C Loaders
- Practical: 12.1 P Load and Display Data Fetched from a Content Provider

END