Mobile programming

Omid Jafarinezhad

Android 06

Storage Options

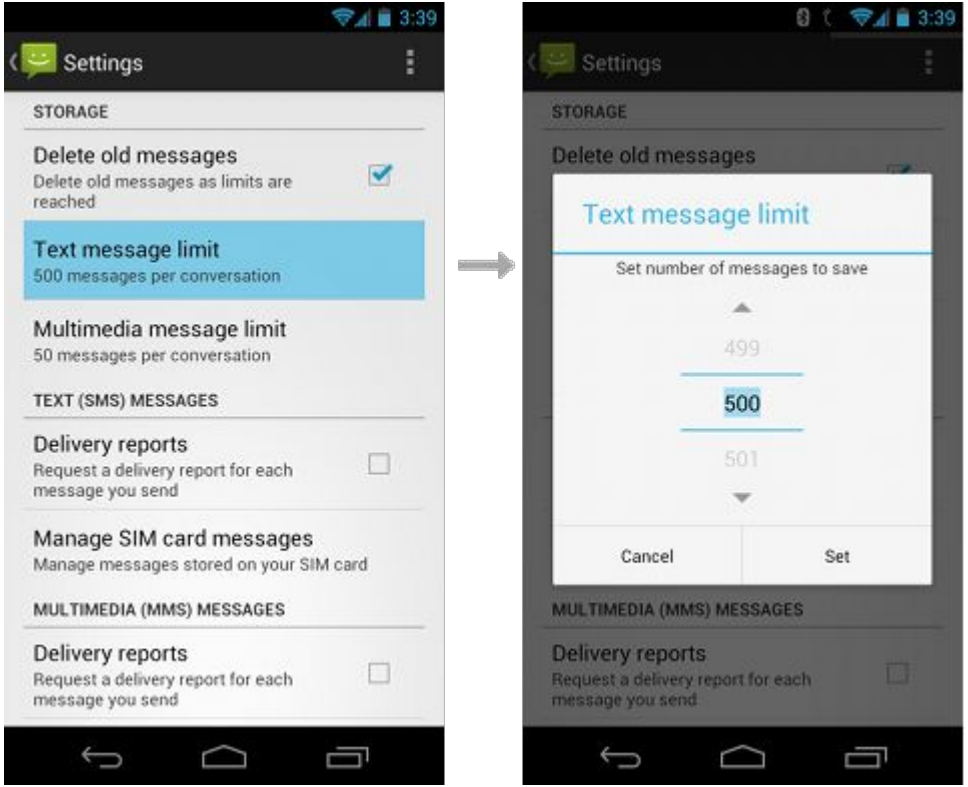
Android provides several options for you **to save persistent application data**. The solution you choose depends on your specific needs, such as whether the data should be *private* to your application or *accessible to other applications* (and the user) and *how much space* your data requires.

Shared Preferences: Store private primitive data in key-value pairs.
Internal Storage: Store private data on the device memory.
External Storage: Store public data on the shared external storage.
SQLite Databases: Store structured data in a private database.
Network Connection: Store data on the web with your own network server.

Settings

Applications often include settings that allow users to modify app features and behaviors

For example, some apps allow users to specify whether notifications are enabled or specify how often the application syncs data with the cloud



Android's Preference APIs

A Preference object is the building block for a single setting

Each Preference appears as an item in a list and provides the appropriate UI for users to modify the setting. For example, a CheckBoxPreference creates a list item that shows a checkbox, and a ListPreference creates an item that opens a dialog with a list of choices

Each Preference you add has a corresponding **key-value pair** that the system uses to save the setting in a **default SharedPreferences file** for your app's settings

SharedPreferences

If you have a relatively **small collection of key-values** that you'd like to save, you should use the SharedPreferences APIs

A **SharedPreferences** object points to **a file containing key-value pairs** and provides simple methods to **read and write** them

Each SharedPreferences file is managed by the framework and **can be private or shared**

Write to Shared Preferences

To write to a shared preferences file, create a SharedPreferences.Editor by calling edit() on your SharedPreferences

Pass the keys and values you want to write with methods such as putInt() and putString(). Then call commit() to save the changes

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putInt(getString(R.string.saved_high_score), newHighScore);
editor.commit();
```

Read from Shared Preferences

You can create a new shared preference file or access an existing one by calling one of two methods:

- **getSharedPreferences()** — Use this if *you need multiple shared preference files identified by name, which you specify with the first parameter*. You can call this from any Context in your app.

- **getPreferences()** — Use this from an Activity if *you need to use only one shared preference file for the activity*. Because this retrieves a default shared preference file that belongs to the activity, you don't need to supply a name.

Read from Shared Preferences

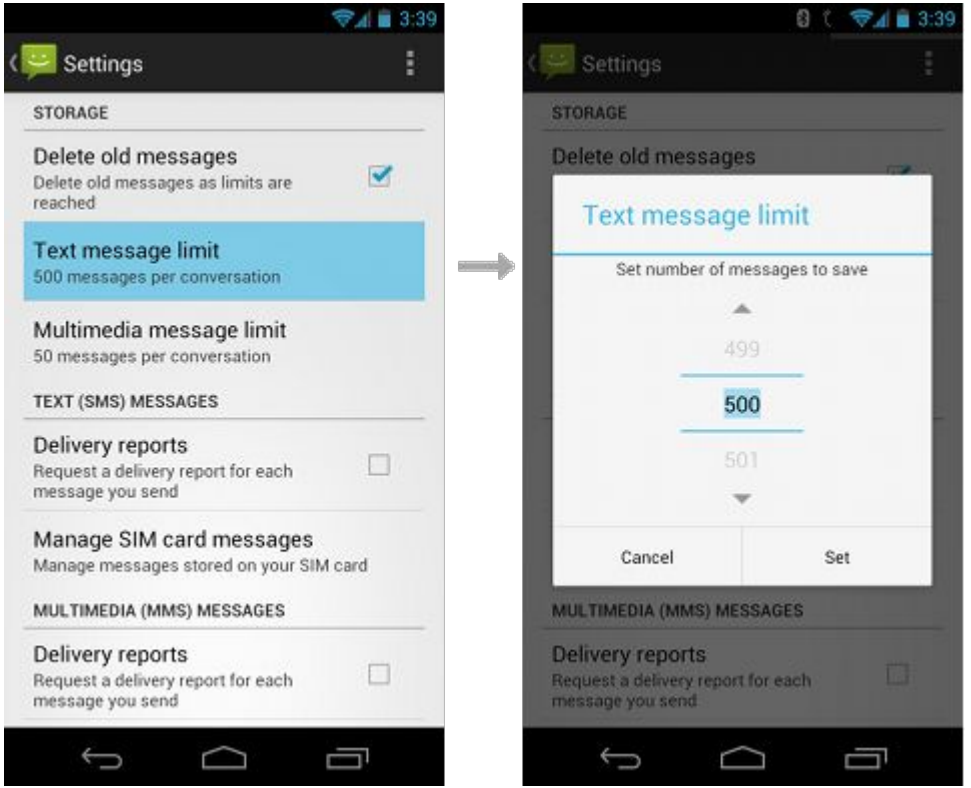
```
Context context = getActivity();
SharedPreferences sharedPref = context.getSharedPreferences(
        getString(R.string.preference_file_key), Context.MODE_PRIVATE);
```

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
```

```
SharedPreferences sharedPref = getActivity().getPreferences(Context.MODE_PRIVATE);
int defaultValue = getResources().getInteger(R.string.saved_high_score_default);
long highScore = sharedPref.getInt(getString(R.string.saved_high_score), defaultValue);
```

Android's Preference APIs

Each Preference you add has a corresponding **key-value pair** that the system uses to save the setting in a **default SharedPreferences file** for your app's settings



SharedPreferences and data types

The value saved in SharedPreferences for each setting can be one of the following data types:

- Boolean
- Float
- Int
- Long
- String
- String Set

Overview

if your app supports versions of **Android older than 3.0** (API level 10 and lower), you must build the activity as an extension of the **PreferenceActivity** class

On Android 3.0 and later, you should instead use a traditional Activity that **hosts a PreferenceFragment** that displays your app settings. However, you can also use PreferenceActivity to create a two-pane layout for large screens when you have multiple groups of settings.

Preferences

Every setting for your app is represented by a specific subclass of the Preference class

- A few of the most common preferences are:
  CheckBoxPreference: Shows an item with a checkbox for a setting that is either enabled or disabled. The saved value is a boolean (true if it's checked).
  ListPreference: Opens a dialog with a list of radio buttons. The saved value can be any one of the supported value types (listed above).
  EditTextPreference: Opens a dialog with an EditText widget. The saved value is a String.

Defining Preferences in XML

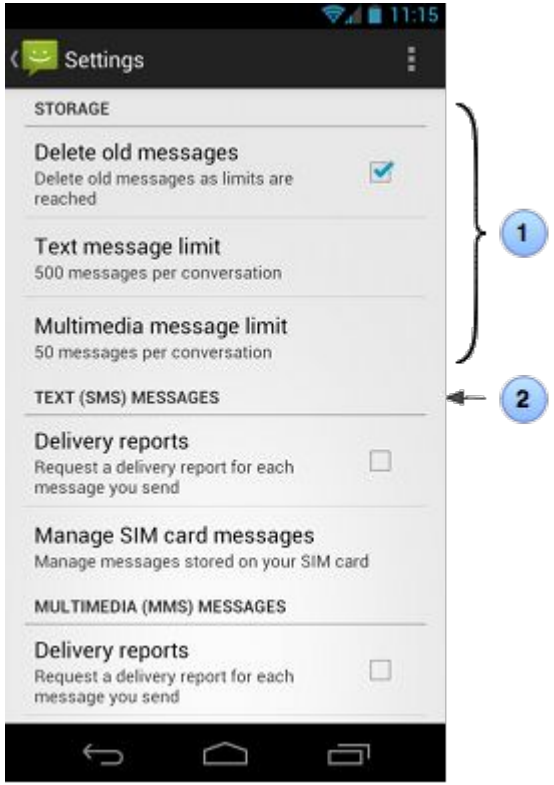
You must save the XML file in the **res/xml/** directory. Although **you can name the file anything you want**, it's traditionally named **preferences.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <CheckBoxPreference
        android:key="pref_sync"
        android:title="@string/pref_sync"
        android:summary="@string/pref_sync_summ"
        android:defaultValue="true" />
    <ListPreference
        android:dependency="pref_sync"
        android:key="pref_syncConnectionType"
        android:title="@string/pref_syncConnectionType"
        android:dialogTitle="@string/pref_syncConnectionType"
        android:entries="@array/pref_syncConnectionTypes_entries"
        android:entryValues="@array/pref_syncConnectionTypes_values"
        android:defaultValue="@string/pref_syncConnectionTypes_default" />
</PreferenceScreen>
```

Creating setting groups

A group of related settings can be presented in one of three ways:

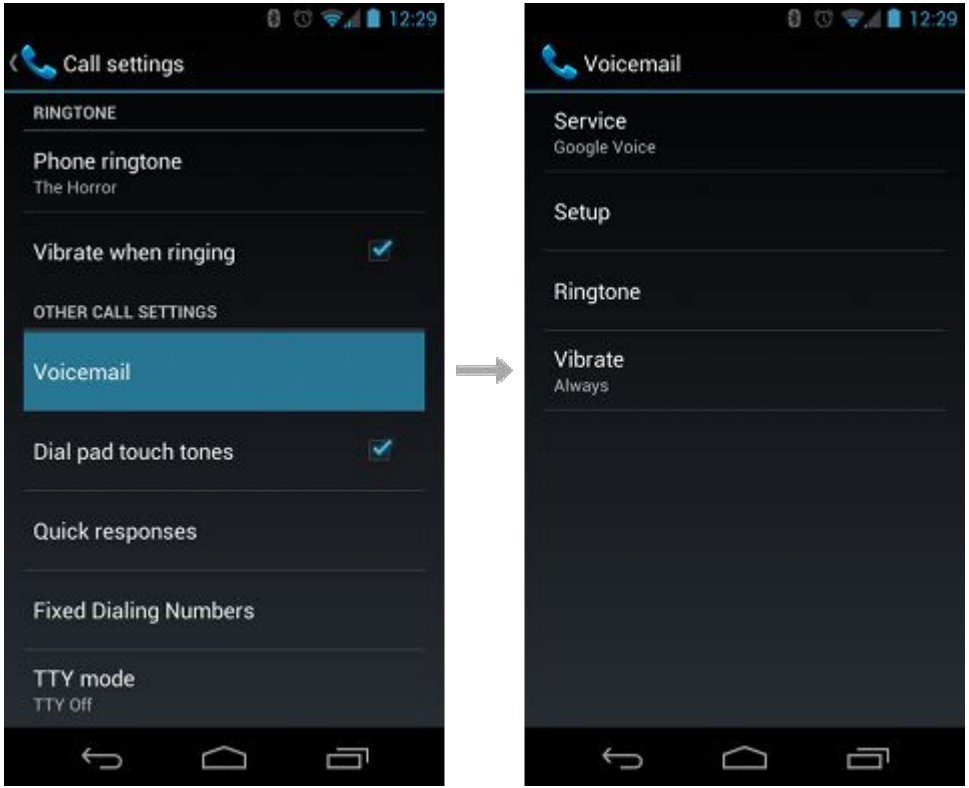
- **Using titles**
- **Using subscreens**
- **Using intents**



Using titles

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory
        android:title="@string/pref_sms_storage_title"
        android:key="pref_key_storage_settings">
        <CheckBoxPreference
            android:key="pref_key_auto_delete"
            android:summary="@string/pref_summary_auto_delete"
            android:title="@string/pref_title_auto_delete"
            android:defaultValue="false"... />
        <Preference
            android:key="pref_key_sms_delete_limit"
            android:dependency="pref_key_auto_delete"
            android:summary="@string/pref_summary_delete_limit"
            android:title="@string/pref_title_sms_delete"... />
        <Preference
            android:key="pref_key_mms_delete_limit"
            android:dependency="pref_key_auto_delete"
            android:summary="@string/pref_summary_delete_limit"
            android:title="@string/pref_title_mms_delete" ... />
    </PreferenceCategory>
    ...
</PreferenceScreen>
```

Using subscreens



```
<PreferenceScreen  xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- opens a subscreen of settings -->
    <PreferenceScreen
        android:key="button_voicemail_category_key"
        android:title="@string/voicemail"
        android:persistent="false">
        <ListPreference
            android:key="button_voicemail_provider_key"
            android:title="@string/voicemail_provider" ... />
        <!-- opens another nested subscreen -->
        <PreferenceScreen
            android:key="button_voicemail_setting_key"
            android:title="@string/voicemail_settings"
            android:persistent="false">
            ...
        </PreferenceScreen>
        <RingtonePreference
            android:key="button_voicemail_ringtone_key"
            android:title="@string/voicemail_ringtone_title"
            android:ringtoneType="notification" ... />
        ...
    </PreferenceScreen>
    ...
</PreferenceScreen>
```

Using intents

In some cases, **you might want a preference item to open a different activity instead of a settings screen**, such as a web browser to view a web page. To invoke an Intent when the user selects a preference item, add an <intent> element as a child of the corresponding <Preference> element

```
<Preference android:title="@string/prefs_web_page" >
    <intent android:action="android.intent.action.VIEW"
            android:data="http://www.example.com" />
</Preference>
```

Creating a Preference Activity

To display your settings in an activity, **extend the PreferenceActivity class.** This is an extension of the traditional **Activity** class that displays a list of settings based on a hierarchy of Preference objects

If you're developing your application for **Android 3.0 and higher**, you should instead use **PreferenceFragment**

Creating a Preference Activity

The most important thing to remember is that you do not load a layout of views during the onCreate() callback. Instead, you call **addPreferencesFromResource()** to add the preferences you've declared in an XML file to the activity

```
public class SettingsActivity extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

Using Preference Fragments

If you're developing for **Android 3.0 (API level 11) and higher**, you should use a **PreferenceFragment to display your list of Preference objects**. You can add a PreferenceFragment to any activity—you don't need to use PreferenceActivity

```java
public static class SettingsFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Load the preferences from an XML resource
        addPreferencesFromResource(R.xml.preferences);
    }
    ...
}
```

```java
public class SettingsActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Display the fragment as the main content.
        getFragmentManager().beginTransaction()
                .replace(android.R.id.content, new SettingsFragment())
                .commit();
    }
}
```

Setting Default Values

The preferences you create probably define some important behaviors for your application, so it's necessary that **you initialize the associated SharedPreferences file with default values for each Preference when the user first opens your application**

Setting Default Values

The first thing you must do is specify a default value for each Preference object in your XML file using the **android:defaultValue attribute**. The value can be any data type that is appropriate for the corresponding Preference object. For example:

```
<!-- default value is a boolean -->
<CheckBoxPreference
    android:defaultValue="true"
    ... />

<!-- default value is a string -->
<ListPreference
    android:defaultValue="@string/pref_syncConnectionTypes_default"
    ... />
```

Setting Default Values

Then, **from the onCreate() method** in your application's main activity—and in any other activity through which the user may enter your application for the first time—**call setDefaultValues()**:

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

A boolean indicating whether the default values should be set more than once. When false, the system sets the default values only if this method has never been called in the past (or the KEY_HAS_SET_DEFAULT_VALUES in the default value shared preferences file is false)

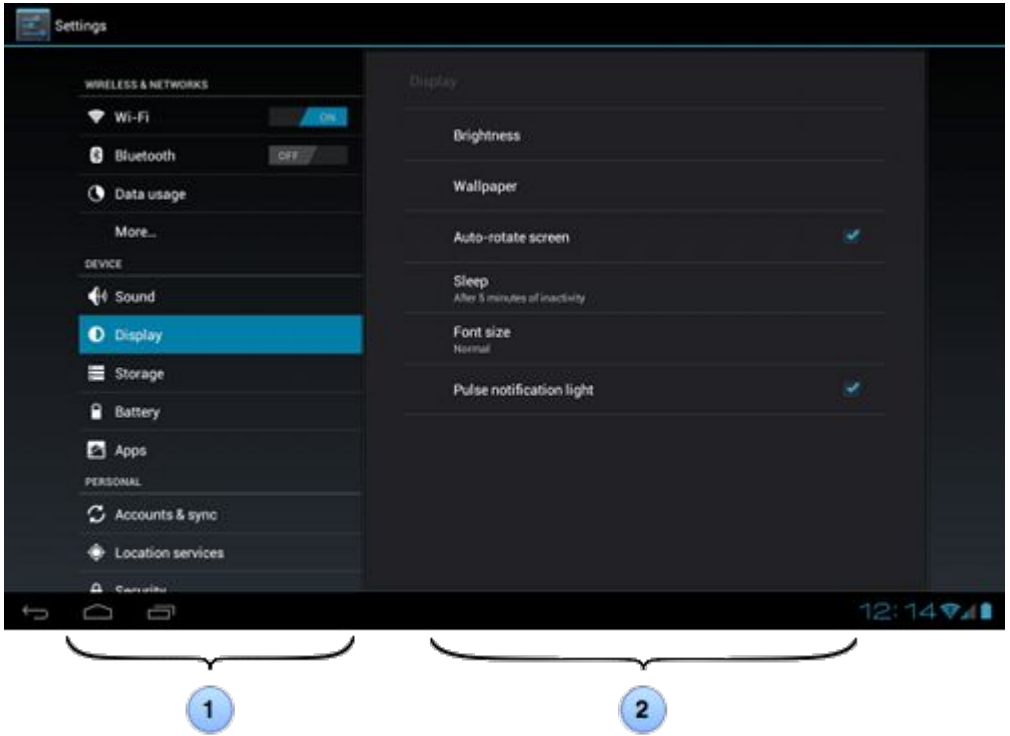
Setting Default Values

As long as you set the third argument to false, you can safely call this method every time your activity starts without overriding the user's saved preferences by resetting them to the defaults. However, if you set it to true, you will override any previous values with the defaults

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

Using Preference Headers

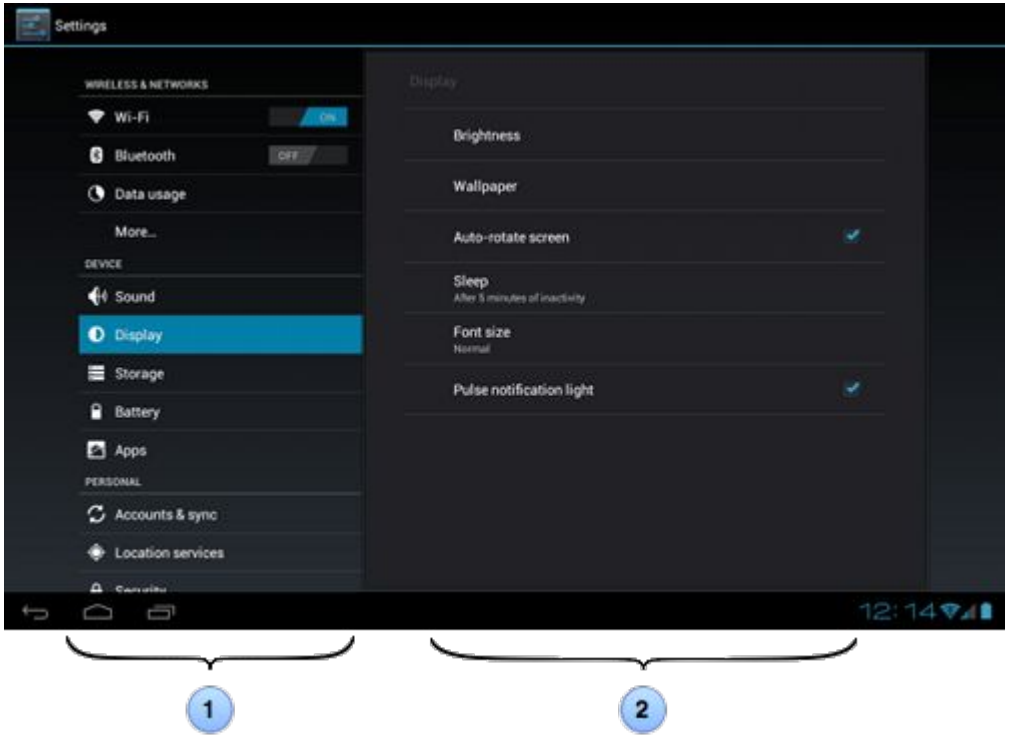
When you're developing such a design for Android 3.0 and higher, you should use a new "headers" feature in Android 3.0, instead of building subscreens with nested PreferenceScreen elements



Using Preference Headers

1. The headers are defined with an XML headers file

2. Each group of settings is defined by a PreferenceFragment that's specified by a <header> element in the headers file



To build your settings with headers

Separate each group of settings into separate instances of PreferenceFragment. That is, each group of settings needs a separate XML file

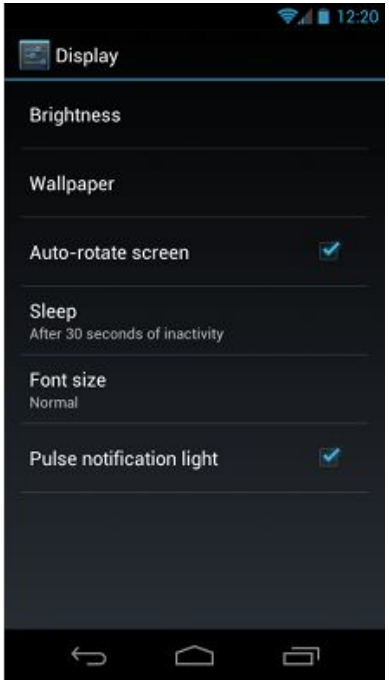
Create an XML headers file that lists each settings group and declares which fragment contains the corresponding list of settings

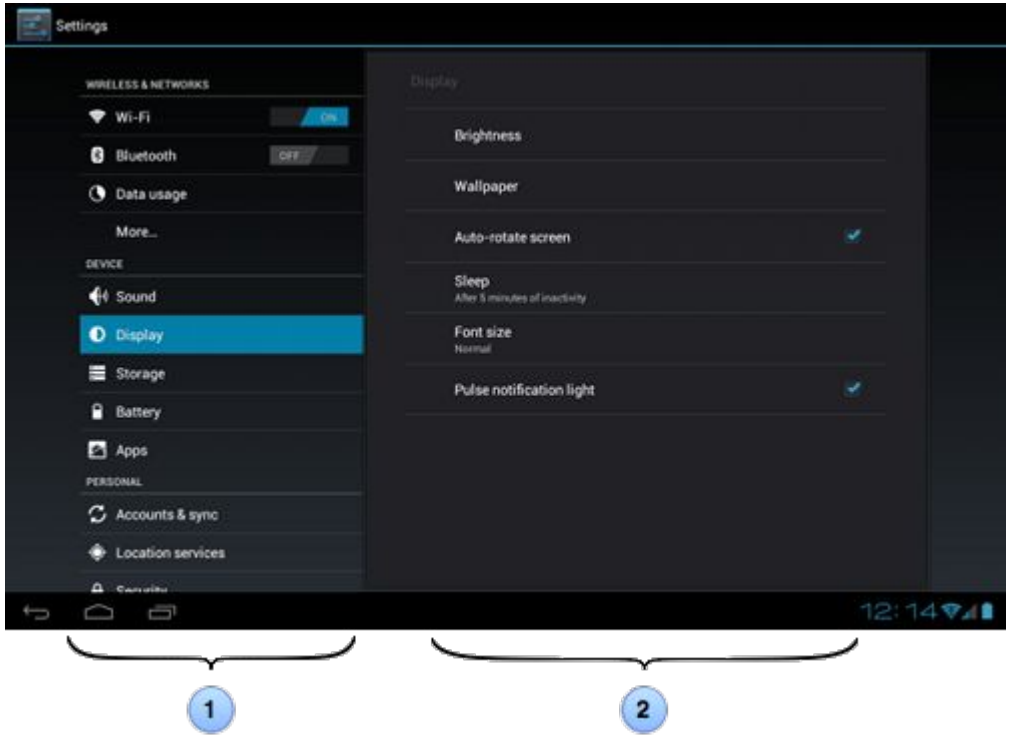
Extend the PreferenceActivity class to host your settings

Implement the onBuildHeaders() callback to specify the headers file

A great benefit to using this design is that PreferenceActivity automatically presents the two-pane layout shown in figure 4 when running on large screens


12:19
Settings
WIRELESS & NETWORKS
Wi-Fi
ON
Bluetooth
OFF
Data usage
More...
DEVICE
Sound
Display
Storage
Battery
Apps


12:20
Display
Brightness
Wallpaper
Auto-rotate screen
Sleep
After 30 seconds of inactivity
Font size
Normal
Pulse notification light


Settings
WIRELESS & NETWORKS
Wi-Fi
Bluetooth
Data usage
More...
DEVICE
Sound
Display
Storage
Battery
Apps
PERSONAL
Accounts & sync
Location services
Display
Brightness
Wallpaper
Auto-rotate screen
Sleep
Font size
Normal
Pulse notification light
12:14
1
2

Add a Fragment to an Activity using XML

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <fragment android:name="com.example.android.fragments.HeadlinesFragment"
              android:id="@+id/headlines_fragment"
              android:layout_weight="1"
              android:layout_width="0dp"
              android:layout_height="match_parent" />

    <fragment android:name="com.example.android.fragments.ArticleFragment"
              android:id="@+id/article_fragment"
              android:layout_weight="2"
              android:layout_width="0dp"
              android:layout_height="match_parent" />

</LinearLayout>
```

Creating the headers file

```xml
<?xml version="1.0" encoding="utf-8"?>
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
    <header
        android:fragment="com.example.prefs.SettingsActivity$SettingsFragmentOne"
        android:title="@string/prefs_category_one"
        android:summary="@string/prefs_summ_category_one" />
    <header
        android:fragment="com.example.prefs.SettingsActivity$SettingsFragmentTwo"
        android:title="@string/prefs_category_two"
        android:summary="@string/prefs_summ_category_two" >
        <!-- key/value pairs can be included as arguments for the fragment. -->
        <extra android:name="someKey" android:value="someHeaderValue" />
    </header>
</preference-headers>
```

<extras> element

Allows you to pass **key-value pairs to the fragment in a Bundle**. The fragment can retrieve the arguments by calling getArguments(). You might pass arguments to the fragment for a variety of reasons, but one good reason is to reuse the same subclass of PreferenceFragment for each group and use the argument to specify which preferences XML file the fragment should load

```
public static class SettingsFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        String settings = getArguments().getString("settings");
        if ("notifications".equals(settings)) {
            addPreferencesFromResource(R.xml.settings_wifi);
        } else if ("sync".equals(settings)) {
            addPreferencesFromResource(R.xml.settings_sync);
        }
    }
}
```

Displaying the headers

To display the preference headers, you must implement the onBuildHeaders() callback method and call loadHeadersFromResource(). For example:

```
public class SettingsActivity extends PreferenceActivity {
    @Override
    public void onBuildHeaders(List<Header> target) {
        loadHeadersFromResource(R.xml.preference_headers, target);
    }
}
```

Note: When using preference headers, your subclass of PreferenceActivity doesn't need to implement the onCreate() method, because the only required task for the activity is to load the headers.

Supporting older versions with preference headers

If your application supports versions of Android older than 3.0, you can still use headers to provide a two-pane layout when running on Android 3.0 and higher. All you need to do is create an additional preferences XML file that uses basic <Preference> elements that behave like the header items (to be used by the older Android versions)

Instead of opening a new PreferenceScreen, however, each of the <Preference> elements sends an Intent to the PreferenceActivity that specifies which preference XML file to load.

example

on Android 3.0 and higher (res/xml/preference_headers.xml)

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
    <header
        android:fragment="com.example.prefs.SettingsFragmentOne"
        android:title="@string/prefs_category_one"
        android:summary="@string/prefs_summ_category_one" />
    <header
        android:fragment="com.example.prefs.SettingsFragmentTwo"
        android:title="@string/prefs_category_two"
        android:summary="@string/prefs_summ_category_two" />
</preference-headers>
```

example

for versions older than Android 3.0 (res/xml/preference_headers_legacy.xml)

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <Preference
        android:title="@string/prefs_category_one"
        android:summary="@string/prefs_summ_category_one"  >
        <intent
            android:targetPackage="com.example.prefs"
            android:targetClass="com.example.prefs.SettingsActivity"
            android:action="com.example.prefs.PREFS_ONE" />
    </Preference>
    <Preference
        android:title="@string/prefs_category_two"
        android:summary="@string/prefs_summ_category_two" >
        <intent
            android:targetPackage="com.example.prefs"
            android:targetClass="com.example.prefs.SettingsActivity"
            android:action="com.example.prefs.PREFS_TWO" />
    </Preference>
</PreferenceScreen>
```

Because support for <preference-headers> was added in Android 3.0, the system calls onBuildHeaders() in your PreferenceActivity only when running on Androd 3.0 or higher. In order to load the "legacy" headers file (preference_headers_legacy.xml), you must check the Android version and, if the version is older than Android 3.0 (HONEYCOMB), call addPreferencesFromResource() to load the legacy header file. For example

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

    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
        // Load the legacy preferences headers
        addPreferencesFromResource(R.xml.preference_headers_legacy);
    }
}

// Called only on Honeycomb and later
@Override
public void onBuildHeaders(List<Header> target) {
   loadHeadersFromResource(R.xml.preference_headers, target);
}
```

The only thing left to do is handle the Intent that's passed into the activity to identify which preference file to load. So retrieve the intent's action and compare it to known action strings that you've used in the preference XML's <intent> tags:

```
final static String ACTION_PREFS_ONE = "com.example.prefs.PREFS_ONE";
...

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

    String action = getIntent().getAction();
    if (action != null && action.equals(ACTION_PREFS_ONE)) {
        addPreferencesFromResource(R.xml.preferences);
    }
    ...

    else if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
        // Load the legacy preferences headers
        addPreferencesFromResource(R.xml.preference_headers_legacy);
    }
}
```

Listening for preference changes

In order to receive a callback when a change happens to any one of the preferences, implement the SharedPreference.
OnSharedPreferenceChangeListener interface and register the listener for the SharedPreferences object by calling
registerOnSharedPreferenceChangeListener()

```
public class SettingsActivity extends PreferenceActivity
                              implements OnSharedPreferenceChangeListener {
    public static final String KEY_PREF_SYNC_CONN = "pref_syncConnectionType";
    ...

    public void onSharedPreferenceChanged(SharedPreferences sharedPreferences,
        String key) {
        if (key.equals(KEY_PREF_SYNC_CONN)) {
            Preference connectionPref = findPreference(key);
            // Set summary to be the user-description for the selected value
            connectionPref.setSummary(sharedPreferences.getString(key, ""));
        }
    }
```

Note

we recommend that you update the summary for **a ListPreference or other multiple choice setting** each time the user changes the preference in order to describe the current setting

For proper lifecycle management in the activity, we recommend that you register and unregister your SharedPreferences.OnSharedPreferenceChangeListener during the onResume() and onPause() callbacks, respectively

```java
@Override
protected void onResume() {
    super.onResume();
    getPreferenceScreen().getSharedPreferences()
            .registerOnSharedPreferenceChangeListener(this);
}

@Override
protected void onPause() {
    super.onPause();
    getPreferenceScreen().getSharedPreferences()
            .unregisterOnSharedPreferenceChangeListener(this);
}
```

Caution: When you call registerOnSharedPreferenceChangeListener(), the preference manager does not currently store a strong reference to the listener. You must store a strong reference to the listener, or it will be susceptible to garbage collection. We recommend you keep a reference to the listener in the instance data of an object that will exist as long as you need the listener.

in the following code, the caller does not keep a reference to the listener. As a result, the listener will be subject to garbage collection, and it will fail at some indeterminate time in the future:

```
prefs.registerOnSharedPreferenceChangeListener(
  // Bad! The listener is subject to garbage collection!
  new SharedPreferences.OnSharedPreferenceChangeListener() {
  public void onSharedPreferenceChanged(SharedPreferences prefs, String key) {
    // listener implementation
  }
});
```

Instead, store a reference to the listener in an instance data field of an object that will exist as long as the listener is needed:

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

Saving Files

Android uses a **file system** that's similar to **disk-based file systems** on other platforms

A **File** object is **suited to reading or writing large amounts of data** in start-to-finish order without skipping around

All Android devices have two file storage areas: "**internal**" and "**external**" storage. These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage)

Internal or External

Internal storage:

It's always available.

Files saved here are accessible by only your app by default.

When the user uninstalls your app, the system removes all your app's files from internal storage.

Internal storage is best when you want to be sure that neither the user nor other apps can access your files.

External storage:

It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.

It's world-readable, so files saved here may be read outside of your control.

When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from getExternalFilesDir().

External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

Obtain Permissions for External Storage

Caution: Currently, all apps have the ability to read the external storage without a special permission. However, this will change in a future release. If your app needs to read the external storage (but not write to it), then you will need to declare the **READ_EXTERNAL_STORAGE** permission. To ensure that your app continues to work as expected, you should declare this permission now, before the change takes effect.

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

Save a File on Internal Storage

When saving a file to internal storage, you can acquire the appropriate directory as a File by calling one of two methods:

getFilesDir(): Returns a File representing an internal directory for your app.

getCacheDir(): Returns a File representing an internal directory for your app's temporary cache files. Be sure to delete each file once it is no longer needed and implement a reasonable size limit for the amount of memory you use at any given time, such as 1MB. If the system begins running low on storage, it may delete your cache files without warning.

To create a new file in one of these directories, you can use the File() constructor, passing the File provided by one of the above methods that specifies your internal storage directory. For example:

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

Alternatively, you can call openFileOutput() to get a FileOutputStream that writes to a file in your internal directory. For example:

```
String filename = "myfile";
String string = "Hello world!";
FileOutputStream outputStream;

try {
  outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
  outputStream.write(string.getBytes());
  outputStream.close();
} catch (Exception e) {
  e.printStackTrace();
}
```

Or, if you need to cache some files, you should instead use createTempFile(). For example:

```
public File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName = Uri.parse(url).getLastPathSegment();
        file = File.createTempFile(fileName, null, context.getCacheDir());
    catch (IOException e) {
        // Error while creating file
    }
    return file;
}
```

Note: *Your app's internal storage directory is specified by your app's package name in a special location of the Android file system*. So as long as you use MODE_PRIVATE for your files on the internal storage, they are never accessible to other apps.

Save a File on External Storage

Because the external storage may be unavailable—such as when the user has mounted the storage to a PC or has removed the SD card that provides the external storage—you should always verify that the volume is available before accessing it

```
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}
```

Although the **external storage is modifiable by the user and other apps**, **there are two categories of files you might save here**:

Public files: Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain available to the user. For example, photos captured by your app or other downloaded files.

Private files: Files that rightfully belong to your app and should be deleted when the user uninstalls your app. Although these files are technically accessible by the user and other apps because they are on the external storage, they are files that realistically don't provide value to the user outside your app. When the user uninstalls your app, the system deletes all files in your app's external private directory. For example, additional resources downloaded by your app or temporary media files.

If you want to save public files on the external storage, use the getExternalStoragePublicDirectory() method to get a File representing the appropriate directory on the external storage. The method takes an argument specifying the type of file you want to save so that they can be logically organized with other public files, such as **DIRECTORY_MUSIC** or **DIRECTORY_PICTURES**. For example

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for the user's public pictures directory.
    File file = new File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

If you want to save files that are private to your app, you can acquire the appropriate directory by calling getExternalFilesDir() and passing it a name indicating the type of directory you'd like. Each directory created this way is added to a parent directory that encapsulates all your app's external storage files, which the system deletes when the user uninstalls your app.

```
public File getAlbumStorageDir(Context context, String albumName) {
    // Get the directory for the app's private pictures directory.
    File file = new File(context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

Query Free Space

If you know ahead of time how much data you're saving, you can find out whether sufficient space is available without causing an IOException by calling **getFreeSpace()** or **getTotalSpace()**. These methods provide the current available space and the total space in the storage volume, respectively. This information is also useful to avoid filling the storage volume above a certain threshold.

Delete a File

You should always delete files that you no longer need. The most straightforward way to delete a file is to have the opened file reference call delete() on itself.

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

If the file is saved on internal storage, you can also ask the Context to locate and delete a file by calling deleteFile():

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

Saving Data in SQL Databases

Saving data to a database is ideal for repeating or **structured data**, such as contact information

NoSql!

Define a Schema and Contract

```
public final class FeedReaderContract {
    // To prevent someone from accidentally instantiating the contract class,
    // give it an empty constructor.
    public FeedReaderContract() {}

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

Create a Database Using a SQL Helper

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the database version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // This database is only a cache for online data, so its upgrade policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

Create a Database Using a SQL Helper

```
private static final String TEXT_TYPE = " TEXT";
private static final String COMMA_SEP = ",";
private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + FeedEntry.TABLE_NAME + " (" +
    FeedEntry._ID + " INTEGER PRIMARY KEY," +
    FeedEntry.COLUMN_NAME_ENTRY_ID + TEXT_TYPE + COMMA_SEP +
    FeedEntry.COLUMN_NAME_TITLE + TEXT_TYPE + COMMA_SEP +
    ... // Any other options for the CREATE command
    " )";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + FeedEntry.TABLE_NAME;
```

Put Information into a Database

```
FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(getContext());
```

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedEntry.COLUMN_NAME_CONTENT, content);

// Insert the new row, returning the primary key value of the new row
long newRowId;
newRowId = db.insert(
         FeedEntry.TABLE_NAME,
         FeedEntry.COLUMN_NAME_NULLABLE,
         values);
```

Read Information from a Database

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    FeedEntry._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_UPDATED,
    ...
    };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_UPDATED + " DESC";

Cursor c = db.query(
    FeedEntry.TABLE_NAME,  // The table to query
    projection,                               // The columns to return
    selection,                                // The columns for the WHERE clause
    selectionArgs,                            // The values for the WHERE clause
    null,                                     // don't group the rows
    null,                                     // don't filter by row groups
    sortOrder                                 // The sort order
    );
```

Read Information from a Database

```
cursor.moveToFirst();
long itemId = cursor.getLong(
    cursor.getColumnIndexOrThrow(FeedEntry._ID)
);
```

Delete Information from a Database

```
// Define 'where' part of query.
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { String.valueOf(rowId) };
// Issue SQL statement.
db.delete(table_name, selection, selectionArgs);
```

Update a Database

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// New value for one column
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the ID
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
String[] selectionArgs = { String.valueOf(rowId) };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```