# 7.0: Storing Data

**Contents:**

- Shared preferences
- Files
- SQLite database
- Other storage options
- Network connection
- Backing up app data

Android offers multiple choices for preserving persistent application data. Your decision should align with your specific requirements, including whether the data should remain exclusive to your application, be open to other applications (and the user), and consider the storage space necessary.

Your data storage options are the following:

- Shared preferences—Store private primitive data in key-value pairs. This will be covered in the next chapter.
- 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.
- Cloud Backup—Backing up app and user data in the cloud.
- Content providers—Store data privately and make them available publicly. This will be covered in the after-next chapter.
- Firebase realtime database—Store and sync data with a NoSQL cloud database. Data is synced across all clients in real time, and remains available when your app goes offline.

## Shared preferences

Using shared preferences is a way to read and write key-value pairs of information persistently to and from a file.

**Note:** By default these key-value pairs are neither shared nor preferences, so do not confuse them with the Preference APIs. Shared Preferences is covered in its ">own chapter.

## Files

Android employs a file system structure akin to disk-based file systems found on other platforms, like Linux. If you have experience with Linux file I/O or the java.io package, file-based operations on Android should feel familiar.

On Android devices, you'll find two primary file storage areas: "internal" and "external" storage. These designations date back to the early days of Android when most devices featured integrated non-volatile memory (referred to as internal storage) alongside a removable storage medium like a micro SD card (referred to as external storage).

In the present landscape, some devices partition their permanent storage into "internal" and "external" segments. This setup ensures the existence of two distinct storage spaces, even in the absence of a removable storage medium. Importantly, the behavior of the API remains consistent, regardless of whether the external storage is removable or integrated. The following lists provide a summary of key details pertaining to each storage space.

| Internal storage | External storage |
|---|---|
| Always available. | Not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device. |
| Only your app can access files. Specifically, your app's internal storage directory is specified by your app's package name in a special location of the Android file system. Other apps cannot browse your internal directories and do not have read or write access unless you explicitly set the files to be readable or writable. | World-readable. Any app can read. |
| When the user uninstalls your app, the system removes all your app's files from internal storage. | 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(). |
| Internal storage is best when you want to be sure that neither the user nor other apps can access your files. | 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. |

## Internal storage

You don't need any permissions to save files on the internal storage. Your application always has permission to read and write files in its internal storage directory.

You can create files in two different directories:

- Permanent storage: `getFilesDir()`
- Temporary storage: `getCacheDir()`. Recommended for small, temporary files totalling less than 1MB. Note that the system may delete temporary files if it runs low on memory.

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, here's how to write some text to a file:

```
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, instead use createTempFile(). For example, the following method extracts the filename from a URL and creates a file with that name in your app's internal cache directory:

```
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;
}
```

## External storage

Utilize external storage for files that require permanent storage, ensuring their availability even if your app is uninstalled. These files should be easily accessible to other users and apps, and may include items like pictures, drawings, or documents generated by your app.
Certain private files that hold no relevance for other apps can also find a home on external storage. Examples of such files include additional app resources downloaded during app use or temporary media files. However, it's crucial to remember to remove these files when your app is uninstalled.

## Obtain permissions for external storage

To write to the external storage, you must request the WRITE_EXTERNAL_STORAGE permission in your manifest file. This implicitly includes permission to read.

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

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.

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

## Always check whether external storage is mounted

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. You can query the external storage state by calling getExternalStorageState(). If the returned state is equal to MEDIA_MOUNTED, then you can read and write your files. For example, the following methods are useful to determine the storage availability:

```
/* 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;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

## Public and private external storage

External storage is very specifically structured and used by the Android system. There are public directories, and private directories specific to your app. Each of these file trees has directories identified by system constants.

For example, any files that you store into the public ringtone directory DIRECTORY_RINGTONES are available to all other ringtone apps.

On the other hand, any files you store in a private ringtone directory DIRECTORY_RINGTONES can, by default, only be seen by your app and are deleted along with your app.

See list of public directories for the full listing.

## Getting file descriptors

To access a public external storage directory, get a path and create a file calling getExternalStoragePublicDirectory().

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

To access a private external storage directory, get a path and create a file calling getExternalFilesDir().

```
File file = new File(getExternalFilesDir(null), "DemoFile.jpg");
```

## Querying storage 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.

You aren't required to check the amount of available space before you save your file. You can instead try writing the file right away, then catch an IOException if one occurs. You may need to do this if you don't know exactly how much space you need.

## Deleting files

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);
```

As a good citizen, you should also regularly delete cached files that you created with getCacheDir().

### Interacting with files summary

Once you have the file descriptors, use standard java.io file operators or streams to interact with the files. This is not Android-specific and not covered here.

## SQLite database

Saving data to a database is ideal for repeating or structured data, such as contact information. Android provides an SQL-like database for this purpose.

The following chapters and practical will teach you in depth how to use SQLite databases with your Android apps:

- SQLite Primer
- Introduction to SQLite Databases SQLite
- Data Storage Practical Searching an
- SQLite Database Practical

## Other storage options

Android provides additional storage options that are beyond the scope of this introductory course. If you'd like to explore them, see the resources below.

## Network connection

You can use the network (when it's available) to store and retrieve data on your own web-based services. To do network operations, use classes in the following packages:

- java.net.*
- android.net.*

## Backing up app data

Users frequently invest substantial time and effort in generating data and configuring preferences within apps. Safeguarding this data for users, whether they replace a malfunctioning device or upgrade to a new one, is a pivotal aspect of delivering an exceptional user experience.

### Auto backup for Android 6.0 (API level 23) and higher

When a user installs your app on a new device or reinstalls it (e.g., after a factory reset), the system automatically restores the app data from the cloud. This automatic backup feature ensures the preservation of the data your app generates on a user's device. The data is securely uploaded to the user's Google Drive account and encrypted for protection. Importantly, neither you nor the user are charged for data storage, and the saved data doesn't count towards the user's personal Google Drive quota.

Each app can store up to 25MB of data. Once the backed-up data reaches this limit, the app ceases to send additional data to the cloud. In the event of a data restore, the system employs the latest data snapshot that the app had previously sent to the cloud.

Automatic backups occur when the following conditions are met:

- The device is idle.
- The device is charging.
- The device is connected to a Wi-Fi network.
- At least 24 hours have elapsed since the last backup.

You can customize and configure auto backup for your app. See Configuring Auto Backup for Apps.

### Backup for Android 5.1 (API level 22) and lower

For users with previous versions of Android, you need to use the Backup API to implement data backup. In summary, this requires you to:

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

More information and sample code:

- Using the Backup API
- Data Backup

## Firebase

Firebase is a mobile platform that helps you develop apps, grow your user base, and earn more money. Firebase is made up of complementary features that you can mix-and-match to fit your needs.

Some features are Analytics, Cloud Messaging, Notifications, and the Test Lab.

For data management, Firebase offers a Realtime Database.

- Store and sync data with a NoSQL cloud database.
- Connected apps share data
- Hosted in the cloud
- Data is stored as JSON
- Data is synchronized in real time to every connected client
- Data remains available when your app goes offline

See the Firebase home for more information.

# 7.1: Shared Preferences

**Contents**:

Shared preferences provide a convenient means of reading and writing small amounts of primitive data stored as key/value pairs within a file on the device's storage. The SharedPreferences class offers various APIs for accessing a preference file, reading, writing, and overseeing this data. It's essential to note that the shared preferences file is managed by the Android framework and is accessible to all components of your app. However, this data remains isolated from and is not shared with or accessible to any other apps.
For the management of substantial data volumes, it's advisable to explore options like a SQLite database or other suitable storage solutions, which will be covered in a subsequent chapter.

## Shared preferences vs. saved instance state

In a previous chapter you learned about preserving state using saved instance states. Here is a comparison between the two.

| **Shared Preferences** | **Saved instance state** |
|---|---|
| Persists across user sessions, even if your app is killed and restarted, or the device is rebooted. | Preserves state data across activity instances in the same user session. |
| Data that should be remembered across sessions, such as a user's preferred settings or their game score. | Data that should not be remembered across sessions, such as the currently selected tab, or any current state of an activity. |
| Small number of key/value pairs. | Small number of key/value pairs. |
| Data is private to the application. | Data is private to the application. |
| Common use is to store user preferences. | Common use is to recreate state after the device has been rotated. |

**Note:** The SharedPreference APIs are also different from the Preference APIs. The Preference APIs can be used to build user interface for a settings page, and they do use shared preferences for their underlying implementation. See Settings for more information on settings and the Preference APIs.

## Creating a shared preferences file

You need only one shared preferences file for your app, and it is customarily named with the package name of your app. This makes its name unique and easily associated with your app.

You create the shared preferences file in the onCreate() method of your main activity and store it in a member variable.

```
private String sharedPrefFile = "com.example.android.hellosharedprefs";
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
```

The mode argument is required, because older versions of Android had other modes that allowed you to create a world- readable or world-writable shared preferences file. These modes were deprecated in API 17, and are now **strongly discouraged** for security reasons. If you need to share data with other apps, use a service or a content provider.

## Saving shared preferences

You save preferences in the onPause() state of the activity lifecycle using tge SharedPreferences.Editor interface.

5. Get a SharedPreferences.Editor. The editor takes care of all the file operations for you. When two editors are modifying preferences at the same time, the last one to call apply wins.
6. Add key/value pairs to the editor using the put method appropriate for the data type. The put methods will overwrite previously existing values of an existing key.
7. Call apply() to write out your changes. The apply() method saves the preferences asynchronously, off of the UI thread. The shared preferences editor also has a commit() method to synchronously save the preferences. The commit() method is discouraged as it can block other operations. As SharedPreferences instances are singletons within a process, it's safe to replace any instance of commit() with apply() if you were already ignoring the return value.

   You don't need to worry about Android component lifecycles and their interaction with apply() writing to disk. The framework makes sure in-flight disk writes from apply() complete before switching states.

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

## Restoring shared preferences

You restore shared preferences in the onCreate() method of your activity. The get() methods take two arguments—one for the key and one for the default value if the key cannot be found. Using the default argument, you don't have to test whether the preference exists in the file.

```
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);
} else { … }
```

## Clearing shared preferences

To clear all the values in the shared preferences file, call the clear() method on the shared preferences editor and apply the changes.

2vds

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();
preferencesEditor.putInt("number", 42);
preferencesEditor.clear();
preferencesEditor.apply();
```

You can combine calls to put and clear. However, when applying the preferences, the clear is always done first, regardless of whether you called clear before or after the put methods on this editor.

## Listening for preference changes

There are several reasons you might want to be notified as soon as the user changes one of the preferences. 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().

The interface has only one callback method, onSharedPreferenceChanged(), and you can implement the interface as a part of your activity.

```
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, ""));
        }
    }
}
```

In this example, the method checks whether the changed setting is for a known preference key. It calls findPreference() to get the Preference object that was changed so it can modify the item's summary to be a description of the user's selection.

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

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

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

### Hold a reference to the listener

When you call registerOnSharedPreferenceChangeListener(), the preference manager does not currently store a reference to the listener. You must hold onto a 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.

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

# 7.2: App Settings

**Contents:**

This chapter describes app settings that let users indicate their preferences for how an app or service should behave.

## Determining appropriate setting controls

Many apps include settings that empower users to customize app features and behaviors. For instance, users may configure notification preferences or determine the frequency of data synchronization with the cloud in some apps.

The settings within an app should encompass controls that capture user preferences affecting the majority of users or offer essential support to a specific user segment. For instance, notification settings influence all users, while a currency setting targeting a foreign market caters to users in that specific market.

Since settings are typically accessed infrequently—once users adjust a setting, they seldom revisit it—consider relocating controls or preferences necessitating frequent access to locations like the app bar's options menu or a side navigation menu such as a navigation drawer.

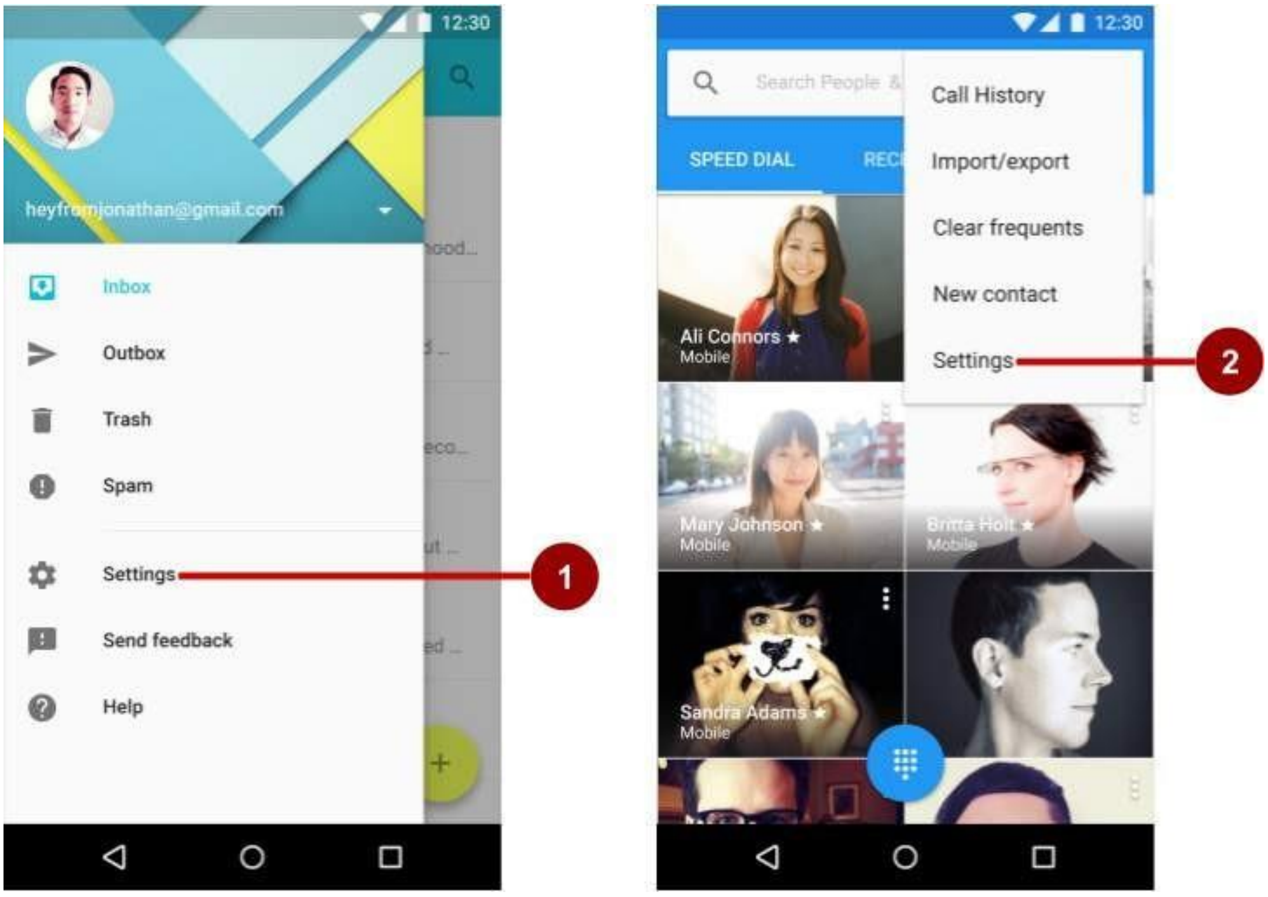
Set defaults for your setting controls that are familiar to users and make the app experience better. The initial default value for a setting should:

- Represent the value most users would choose, such as **All contacts** for "Contacts to display" in the Contacts app. Use
- less battery power. For example, in the Android Settings app, Bluetooth is set to off until the user turns it on. Pose the
- least risk to security and data loss. For example, the default setting for the Gmail app's default action is to archive rather than delete messages.
- Interrupt only when important. For example, the default setting for when calls and notifications arrive is to interrupt only when important.

**Tip**: If the setting contains information about the app, such as a version number or licensing information, move these to a separately-accessed Help screen.

## Providing navigation to Settings

To access app settings, users can conveniently navigate by tapping "Settings," which can be situated in a side navigation menu, such as a navigation drawer (illustrated on the left side of the figure below), or within the options menu located in the app bar (demonstrated on the right side of the figure below). This flexibility allows users to access settings based on the app's design and layout..



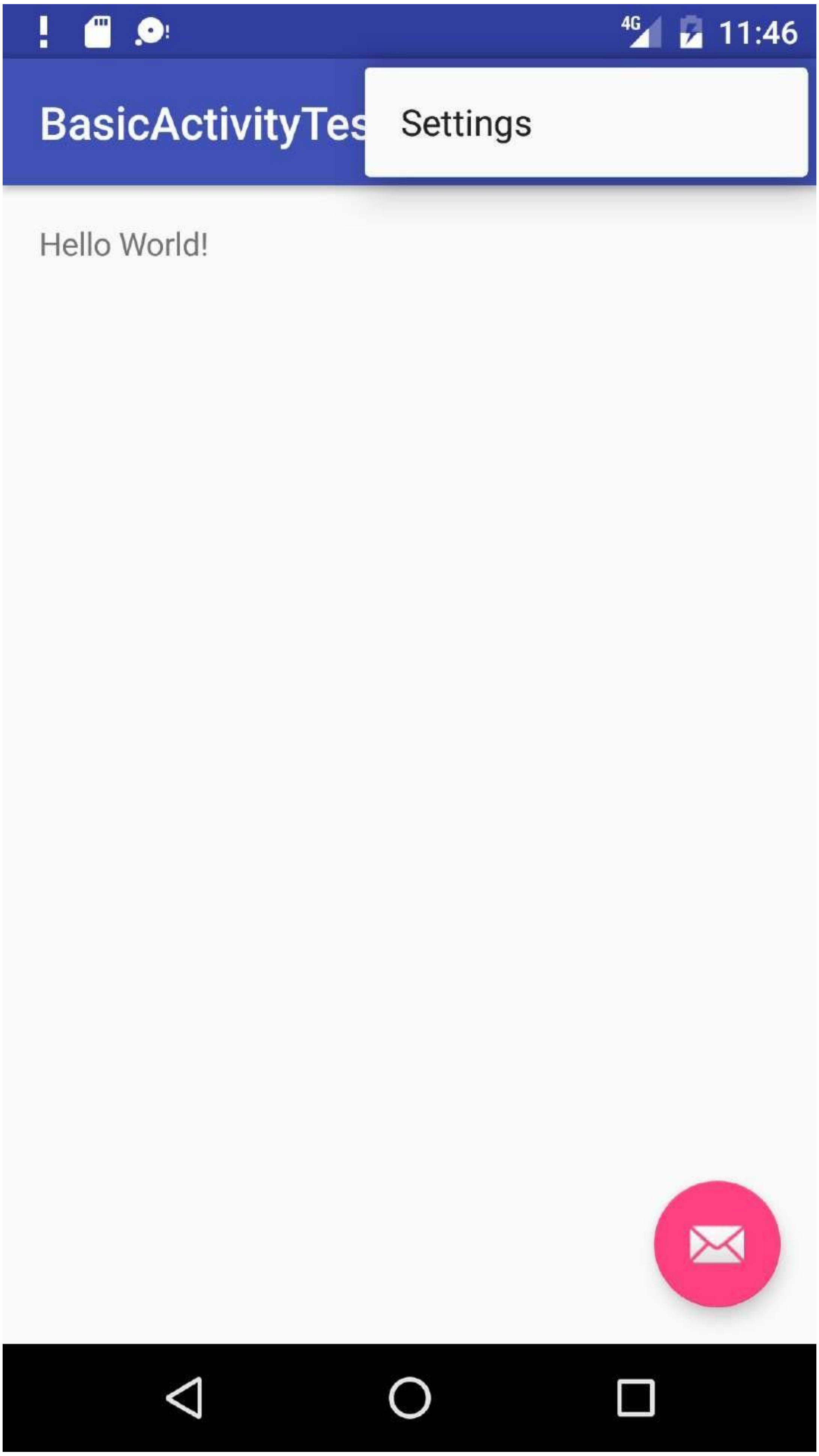
In the figure above:

8. Settings in side navigation (a navigation drawer)
9. Settings in the options menu of the app bar

Follow these design guidelines for navigating to settings:

- If your app offers side navigation such as a navigation drawer, include **Settings** below all other items (except **Help** and **Send Feedback**).
- If your app doesn't offer side navigation, place **Settings** in the app bar menu's options menu below all other items (except **Help** and **Send Feedback**).
  **Note:** Use the word **Settings** in the app's navigation to access the settings. Do not use synonyms such as "Options" or "Preferences."

**Tip**: Android Studio provides a shortcut for setting up an options menu with **Settings**. If you start an Android Studio project for a smartphone or tablet using the Basic Activity template, the new app includes **Settings** as shown below:

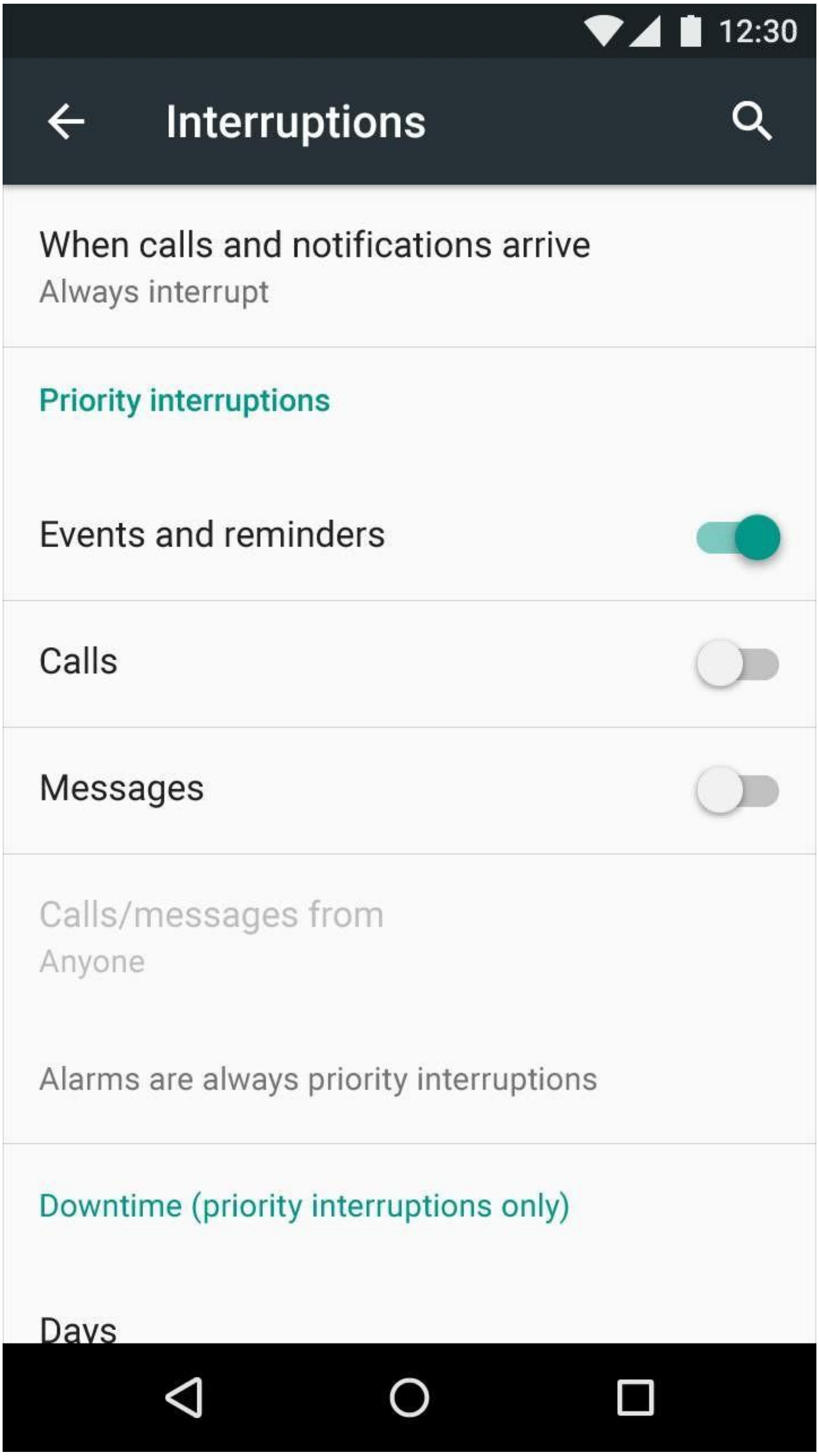


Hello World!

## The Settings UI

Settings should be well-organized, predictable, and contain a manageable number of options. A user should be able to quickly understand all available settings and their current values. Follow these design guidelines:

- **7 or fewer settings**: Arrange them according to priority with the most important ones at the top.
- **7-15 settings**: Group related settings under section dividers. For example, in the figure below, "Priority interruptions" and "Downtime (priority interruptions only)" are section dividers.


12:30

Interruptions

When calls and notifications arrive
Always interrupt

Priority interruptions

Events and reminders ●

Calls ○

Messages ○

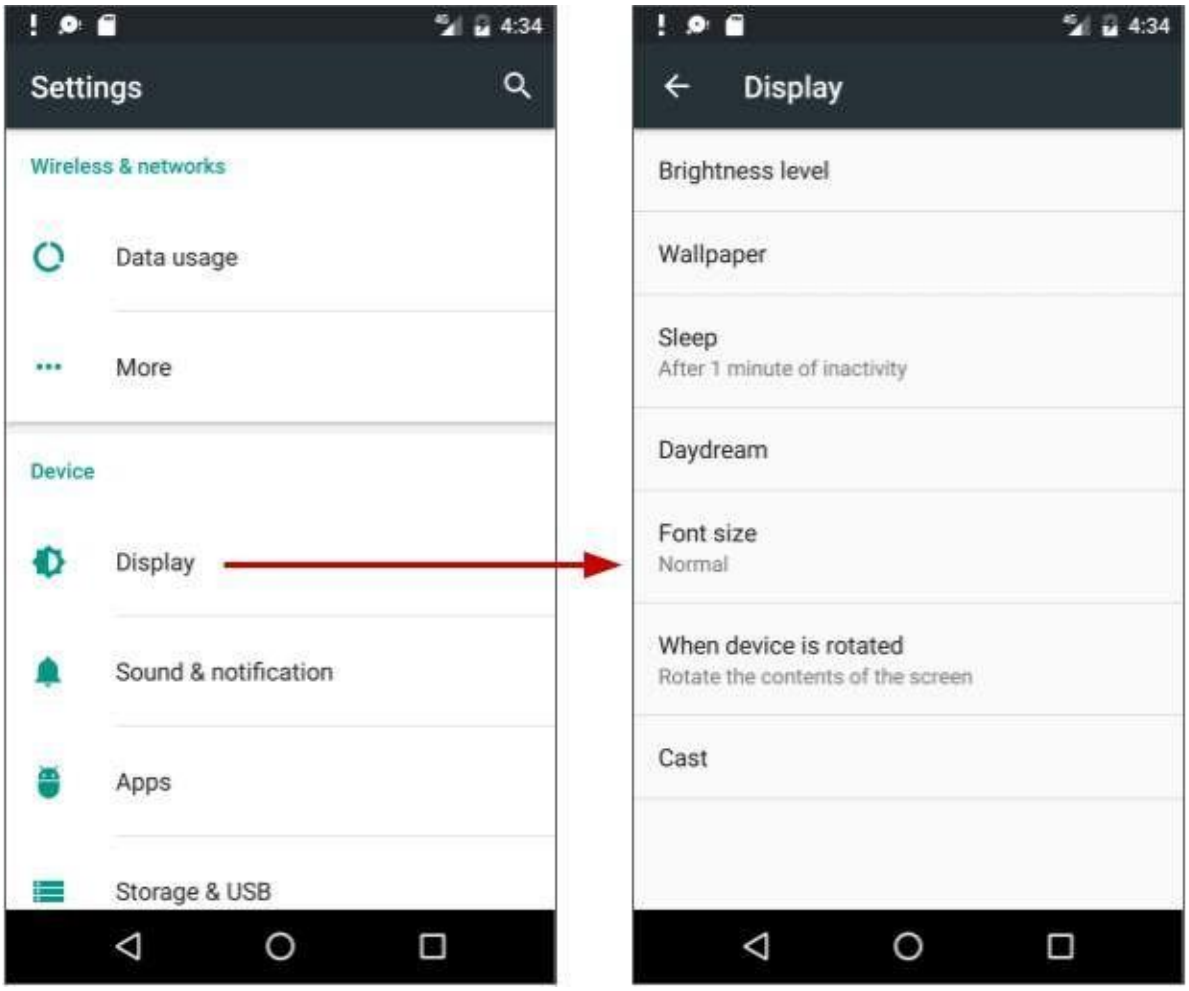
Calls/messages from
Anyone

Alarms are always priority interruptions

Downtime (priority interruptions only)

Days

- **16 or more settings**: Group related settings into separate sub-screens. Use headings, such as **Display** on the main Settings screen (as shown on the left side of the figure below) to enable users to navigate to the display settings (shown on the right side of the figure below):



## Building the settings

Build an app's settings using various subclasses of the Preference class rather than using View objects. This class provides the View to be displayed for each setting, and associates with it a SharedPreferences interface to store/retrieve the preference data.

Each Preference appears as an item in a list. Direct subclasses provide containers for layouts involving multiple settings. For example:

- PreferenceGroup: Represents a group of settings (Preference objects).
- PreferenceCategory: Provides a disabled title above a group as a section divider.
- PreferenceScreen: Represents a top-level Preference that is the root of a Preference hierarchy. Use a PreferenceScreen in a layout at the top of each screen of settings.

For example, to provide dividers with headings between groups of settings (as shown in the previous figure for 7-15 settings), place each group of Preference objects inside a PreferenceCategory. To use separate screens for groups, place each group of Preference objects inside a PreferenceScreen.

Other Preference subclasses for settings provide the appropriate UI for users to change the setting. For example:

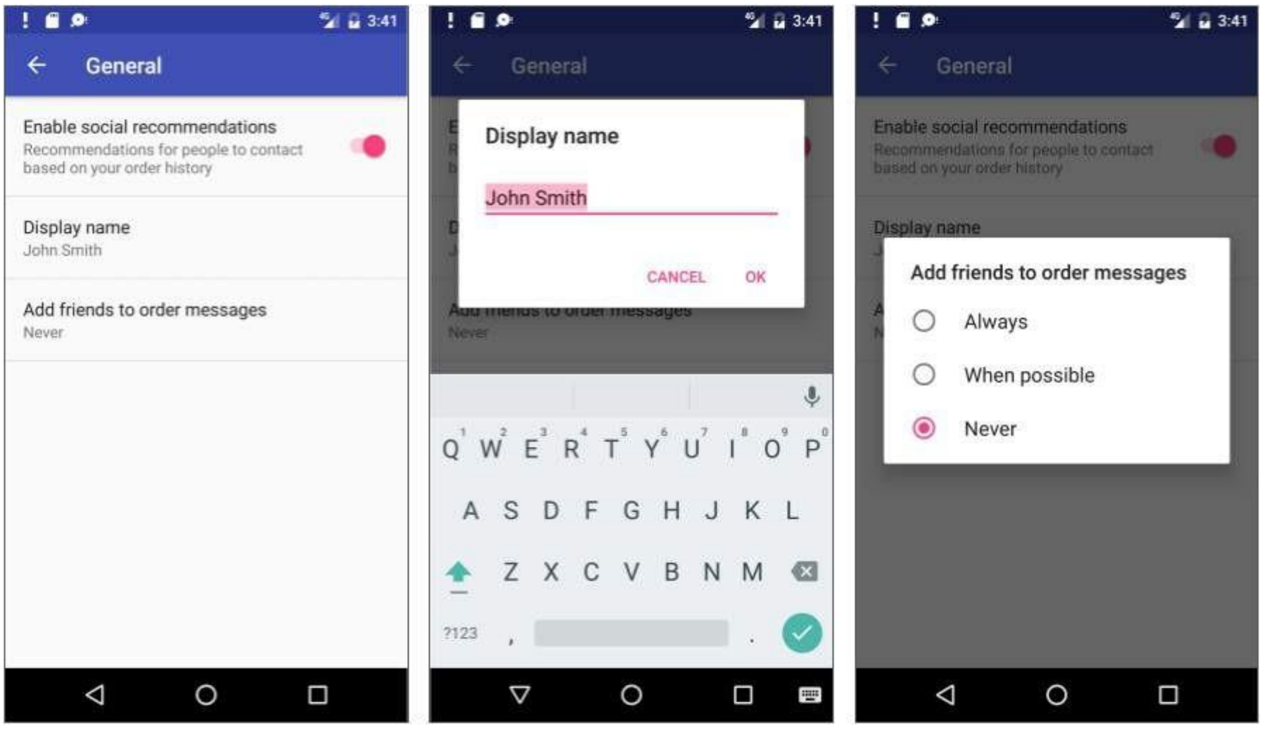
- CheckBoxPreference: Creates a list item that shows a checkbox for a setting that is either enabled or disabled. The saved value is a boolean ( `true` if it's checked).
- ListPreference: Creates an item that opens a dialog with a list of radio buttons.
- SwitchPreference: Creates a two-state toggleable option (such as on/off or true/false).

- EditTextPreference: Creates an item that opens a dialog with an EditText widget. The saved value is a String.
- RingtonePreference: Lets the user to choose a ringtone from those available on the device.

Define your list of settings in XML, which provides an easy-to-read structure that's simple to update. Each Preference subclass can be declared with an XML element that matches the class name, such as `<CheckBoxPreference>` .

## XML attributes for settings

The following example from the Settings Activity template defines a screen with three settings as shown in the figure below: a toggle switch (at the top of the screen on the left side), a text entry field (center), and a list of radio buttons (right):

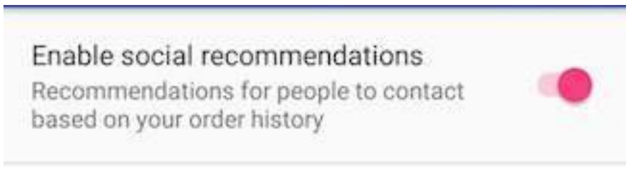

General
Enable social recommendations
Recommendations for people to contact based on your order history
Display name
John Smith
Add friends to order messages
Never
Display name
John Smith
CANCEL
OK
Add friends to order messages
Always
When possible
Never

The root of the settings hierarchy is a `PreferenceScreen` layout:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
. . .
</PreferenceScreen>
```

Inside this layout are three settings:

- `SwitchPreference` : Show a toggle switch to disable or enable an option.

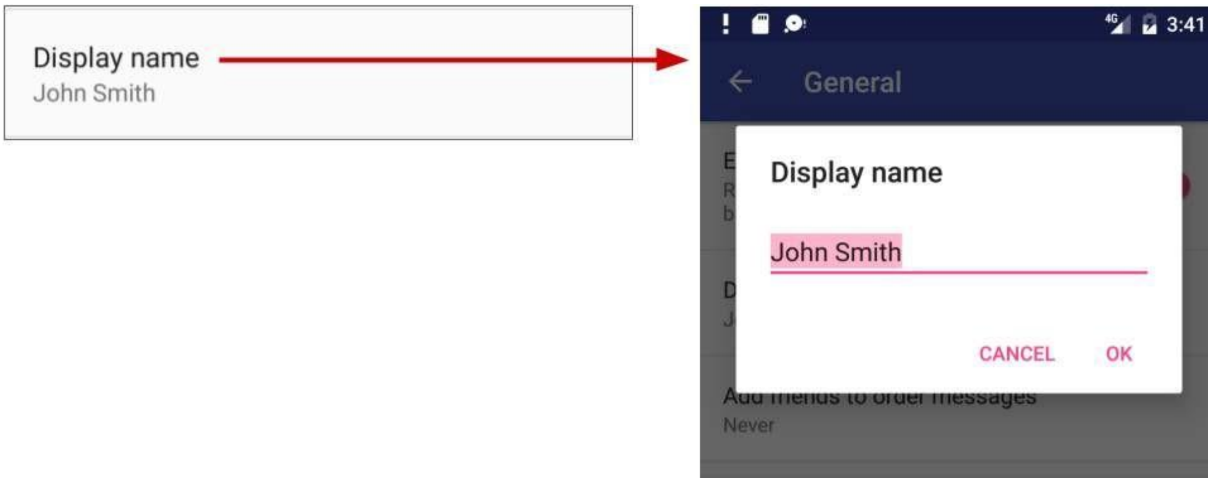

Enable social recommendations
Recommendations for people to contact based on your order history

The setting has the following attributes:

  - `android:defaultValue` : The option is enabled (set to `true` ) by default.
  - `android:summary` : The text summary appears underneath the setting. For some settings, the summary should change to show whether the option is enabled or disabled.
  - `android:title` : The title of the setting. For a `SwitchPreference` , the title appears to the left of the toggle switch.
  - android:key: The key to use for storing the setting value. Each setting (Preference object) has a corresponding key-value pair that the system uses to save the setting in a default SharedPreferences file for your app's settings.

```
<SwitchPreference
     android:defaultValue="true"
     android:key="example_switch"
     android:summary="@string/pref_description_social_recommendations"
     android:title="@string/pref_title_social_recommendations" />
```

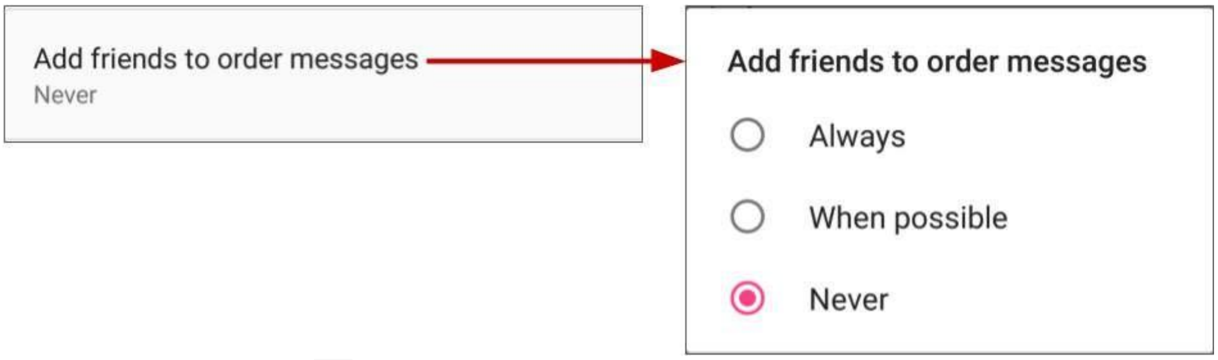
- `EditTextPreference` : Show a text field for the user to enter text.



  - Use EditText attributes such as `android:capitalize` and `android:maxLines` to define the text field's appearance and input control.
  - The default setting is the `pref_default_display_name` string resource.

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

- `ListPreference` : Show a dialog with radio buttons for the user to make one choice.



  - The default value is set to `-1` for no choice.
  - The text for the radio buttons (Always, When possible, and Never) are defined in the `pref_example_list_titles` array and specified by the `android:entries` attribute.
  - The values for the radio button choices are defined in the `pref_example_list_values` array and specified by the `android:entryValues` attribute.
  - The radio buttons are displayed in a dialog, which usually have positive (**OK** or **Accept**) and negative (**Cancel**) buttons. However, a settings dialog doesn't need these buttons, because the user can touch outside the dialog to dismiss it. To hide these buttons, set the `android:positiveButtonText` and `android:negativeButtonText` attributes to `"@null"` .

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

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

If you are using the support v7 appcompat library and extending the Settings Activity with AppCompatActivity and the fragment with PreferenceFragmentCompat, as shown in the next section, change the setting's XML attribute to use the support v7 appcompat library version. For example, for a SwitchPreference setting, change `<SwitchPreference` in the code to:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
   <android.support.v7.preference.SwitchPreferenceCompat
   ... />
</PreferenceScreen>
```

## Displaying the settings

Use a specialized Activity or Fragment subclass to display a list of settings.

- For an app that supports Android 3.0 and newer versions, the best practice for settings is to use a Settings Activity and a fragment for each preference XML file:
  - Add a Settings Activity class that extends Activity and hosts a fragment that extends PreferenceFragment.
  - To remain compatible with the v7 appcompat library, extend the Settings Activity with AppCompatActivity, extend the fragment with PreferenceFragmentCompat.
- If your app must support versions of Android older than 3.0 (API level 10 and lower), build a special settings activity as an extension of the PreferenceActivity class.

Fragments like PreferenceFragment provide a more flexible architecture for your app, compared to using activities alone. A fragment is like a modular section of an activity—it has its own lifecycle and receives its own input events, and you can add or remove a fragment while the activity is running. Use PreferenceFragment to control the display of your settings instead of PreferenceActivity whenever possible.

However, to create a two-pane layout for large screens when you have multiple groups of settings, you can use an activity that extends PreferenceActivity and also use PreferenceFragment to display each list of settings. You will see this pattern with the Settings Activity template as described later in this chapter in "Using the Settings Activity template".

The following examples show you how to remain compatible with the v7 appcompat library by extending the Settings Activity with AppCompatActivity, and extending the fragment with PreferenceFragmentCompat. To use this library and the `PreferenceFragmentCompat` version of PreferenceFragment, you must also add the android.support:preference-v7 library to the **build.gradle (Module: app)** file's `dependencies` section:

```
dependencies {
   ...
   compile 'com.android.support:preference-v7:25.0.1'
}
```

You also need to add the following `preferenceTheme` declaration to the `AppTheme` in the **styles.xml** file:

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
   ...
   <item name="preferenceTheme">@style/PreferenceThemeOverlay</item>
</style>
```

## Using a PreferenceFragment

The following shows how to use a PreferenceFragment to display a list of settings, and how to add a PreferenceFragment to an activity for settings. To remain compatible with the v7 appcompat library, extend the Settings Activity with AppCompatActivity, and extend the fragment with PreferenceFragmentCompat for each preferences XML file.

Replace the automatically generated `onCreate()` method with the onCreatePreferences() method to load a preferences file with `setPreferencesFromResource()`:

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

As shown in the code above, you associate an XML layout of settings with the fragment during the `onCreatePreferences()` callback by calling setPreferencesFromResource() with two arguments:

- `R.xml.` and the name of the XML file ( `preferences` ).
- the `rootKey` to identify the preference root in `PreferenceScreen` .

```
setPreferencesFromResource(R.xml.preferences, rootKey);
```

You can then create an Activity for settings (named `SettingsActivity` ) that extends AppCompatActivity, and add the settings fragment to it:

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

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

The above code is the typical pattern used to add a fragment to an activity so that the fragment appears as the main content of the activity. You use:

- `getFragmentManager()` if the class extends `Activity` and the fragment extends `PreferenceFragment` .
- `getSupportFragmentManager()` if the class extends `AppCompatActivity` and the fragment extends `PreferenceFragmentCompat` .

For more information about fragments, see Fragment.

To set Up navigation for the settings activity, be sure to declare the Settings Activity's parent to be MainActivity in the AndroidManifest.xml file.

## Calling the settings activity

If you implement the options menu with the **Settings** item, use the following intent to call the Settings activity from with the `onOptionsItemSelected()` method when the user taps **Settings** (using `action_settings` for the **Settings** menu resource id):

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
   int id = item.getItemId();
   // ... Handle other options menu items
   if (id == R.id.action_settings) {
      Intent intent = new Intent(this, SettingsActivity.class);
      startActivity(intent);
      return true;
      }
   return super.onOptionsItemSelected(item);
}
```

If you implement a navigation drawer with the **Settings** item, use the following intent to call the Settings activity from with the `onNavigationItemSelected()` method when the user taps **Settings** (using `action_settings` for the **Settings** menu resource id):

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {
   int id = item.getItemId();
   if (id == R.id.action_settings) {
      Intent intent = new Intent(this, SettingsActivity.class);
      startActivity(intent);
   } else if ...
   // ... Handle other navigation drawer items
   return true;
}
```

## Setting the default values for settings

When the user changes a setting, the system saves the changes to a SharedPreferences file. As you learned in another lesson, shared preferences allow you to read and write small amounts of primitive data as key/value pairs to a file on the device storage.

The app must initialize the SharedPreferences file with default values for each setting when the user first opens the app. Follow these steps:

1. Be sure to specify a default value for each setting in your XML file using the `android:defaultValue` attribute:

```
...
<SwitchPreference
         android:defaultValue="true"
         ... />
...
```

2. From the onCreate() method in the app's main activity—and in any other activity through which the user may enter your app for the first time—call setDefaultValues():

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

Step 2 ensures that the app is properly initialized with default settings. The `setDefaultValues()` method takes three arguments:

- The app context, such as `this`.
- The resource ID (`preferences`) for the settings layout XML file which includes the default values set by Step 1 above.

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

## Reading the settings values

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. When the user changes a setting, the system updates the corresponding value in the SharedPreferences file for you. The only time you should directly interact with the associated SharedPreferences file is when you need to read the value in order to determine your app's behavior based on the user's setting.

All of an app's preferences are saved by default to a file that is accessible from anywhere within the app by calling the static method PreferenceManager.getDefaultSharedPreferences(). This method takes the context and returns the SharedPreferences object containing all the key-value pairs that are associated with the Preference objects.

For example, the following code snippet shows how you can read one of the preference values from the main activity's `onCreate()` method:

```
...
SharedPreferences sharedPref =
    PreferenceManager.getDefaultSharedPreferences(this);
Boolean switchPref = sharedPref
    .getBoolean("example_switch", false);
...
```

The above code snippet uses `PreferenceManager.getDefaultSharedPreferences(this)` to get the settings as a SharedPreferences object (`sharedPref`).

It then uses `getBoolean()` to get the boolean value of the preference that uses the key `"example_switch"`. If there is no value for the key, the `getBoolean()` method sets the value to `false`.

## Listening for a setting change

There are several reasons why you might want to set up a listener for a specific setting:

- If a change to the value of a setting also requires changing the summary of the setting, you can listen for the change, and then change the summary with the new setting value.
- If the setting requires several more options, you may want to listen for the change and immediately respond by displaying the options.
  If the setting makes another setting obsolete or inappropriate, you may want to listen for the change and immediately
- respond by disabling the other setting.

To listen to a setting, use the Preference.OnPreferenceChangeListener interface, which includes the onPreferenceChange() method that returns the new value of the setting.

In the following example, the listener retrieves the new value after the setting is changed, and changes the summary of the setting (which appears below the setting in the UI) to show the new value. Follow these steps:

1. Use a shared preferences file, as described in a previous chapter, to store the value of the preference (setting).
   Declare the following variables in the SettingsFragment class definition:

```
public class SettingsFragment extends PreferenceFragment {
   private SharedPreferences mPreferences;
   private String sharedPrefFile = "com.example.android.settingstest";
   ...
}
```

2. Add the following to the `onCreate()` method of SettingsFragment to get the preference defined by the key `example_switch`, and to set the initial text (the string resource `option_on`) for the summary:

```
@Override
public void onCreate(Bundle savedInstanceState) {
   ...
   mPreferences =
            this.getActivity()
            .getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
   Preference preference = this.findPreference("example_switch");
   preference.setSummary(mPreferences.getString("summary",
                              getString(R.string.option_on)));
   ...
}
```

3. Add the following code to `onCreate()` after the code in the previous step:

```
...
preference.setOnPreferenceChangeListener(new
                           Preference.OnPreferenceChangeListener() {
   @Override
   public boolean onPreferenceChange(Preference preference,
                           Object newValue) {
      if ((Boolean) newValue == true) {
         preference.setSummary(R.string.option_on);
         SharedPreferences.Editor preferencesEditor =
                                          mPreferences.edit();
         preferencesEditor.putString("summary",
                           getString(R.string.option_on)).apply();
      } else {
         preference.setSummary(R.string.option_off);
         SharedPreferences.Editor preferencesEditor =
                                          mPreferences.edit();
         preferencesEditor.putString("summary",
                           getString(R.string.option_off)).apply();
      }
      return true;
   }
});
...
```

The code does the following:

i. Listens to a change in the switch setting using onPreferenceChange(), and returns `true`:

```
@Override
public boolean onPreferenceChange(Preference preference,
                  Object newValue) {
 ...
 return true;
}
```

ii. Determines the new boolean value ( `newValue` ) for the setting after the change ( `true` or `false` ):

```
if ((Boolean) newValue == true) {
 ...
} else {
 ...
}
```

iii. Edits the Shared Preferences file (as described in a previous practical) using SharedPreferences.Editor:

```
...
preference.setSummary(R.string.option_on);
SharedPreferences.Editor preferencesEditor =
                                   mPreferences.edit();
...
```

iv. Puts the new value as a string in the summary using putString() and applies the change using apply():

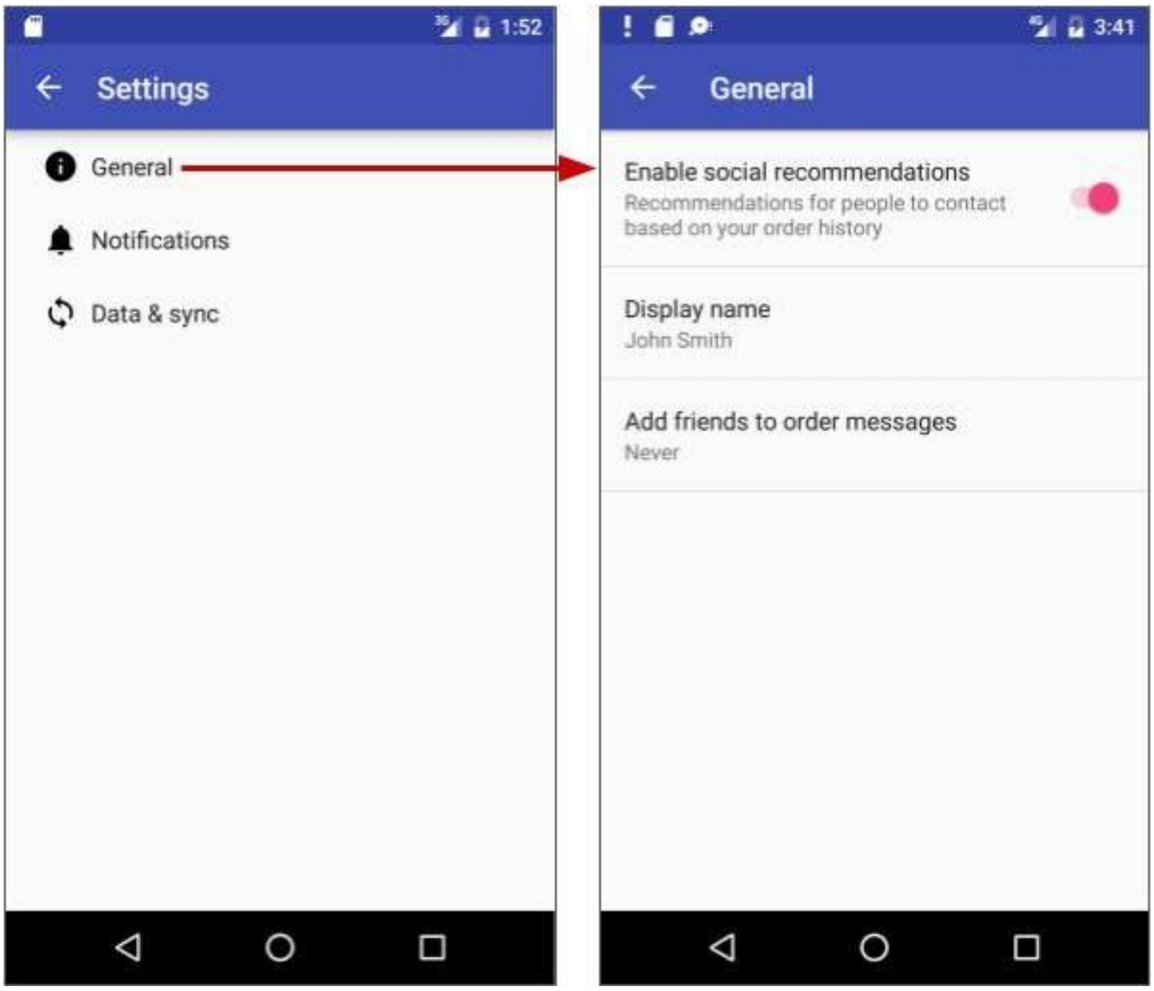
```
...
preferencesEditor.putString("summary",
                   getString(R.string.option_on)).apply();
...
```

## Using the Settings Activity template

If you need to build several sub-screens of settings and you want to take advantage of tablet-sized screens as well as maintain compatibility with older versions of Android for tablets, Android Studio provides a shortcut: the Settings Activity template.

The Settings Activity template is pre-populated with settings you can customize for an app, and provides a different layout for smartphones and tablets:

- *Smartphones*: A main Settings screen with a header link for each group of settings, such as General for general settings, as shown below.

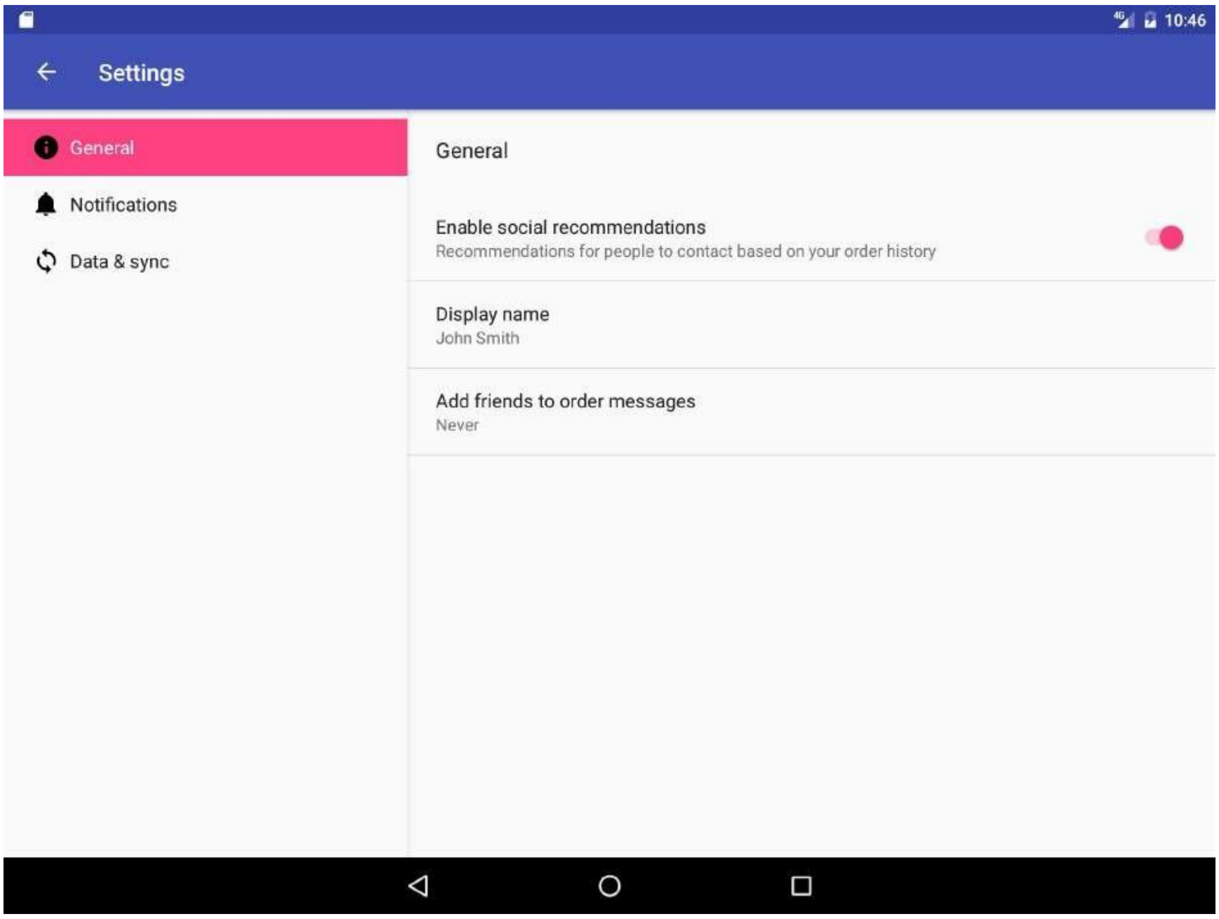


Enable social recommendations
Recommendations for people to contact based on your order history

Display name
John Smith

Add friends to order messages
Never

- *Tablets*: A master/detail screen layout with a header link for each group on the left (master) side, and the group of settings on the right (detail) side, as shown in the figure below.



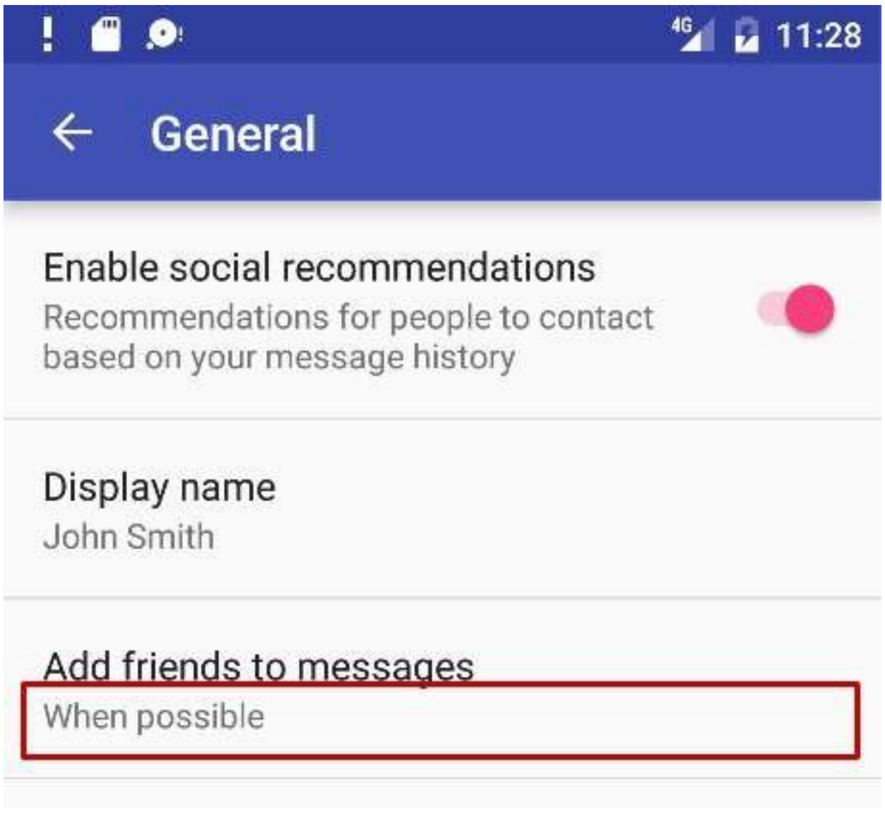
General

Enable social recommendations
Recommendations for people to contact based on your order history

Display name
John Smith

Add friends to order messages
Never

The Settings Activity template also provides the function of listening to a settings change, and changing the summary to reflect the settings change. For example, if you change the "Add friends to messages" setting (the choices are **Always**, **When possible**, or **Never**), the choice you make appears in the summary underneath the setting:



Enable social recommendations
Recommendations for people to contact based on your message history

Display name
John Smith

Add friends to messages
When possible

In general, you need not change the Settings Activity template code in order to customize the activity for the settings you want in your app. You can customize the settings titles, summaries, possible values, and default values without changing the template code, and even add more settings to the groups that are provided. To customize the settings, edit the string and string array resources in the strings.xml file and the layout attributes for each setting in the files in the **xml** directory.

You use the Settings Activity template code as-is. To make it work for your app, add code to the Main Activity to set the default settings values, and to *read* and *use* the settings values, as shown later in this chapter.

## Including the Settings Activity template in your project

To include the Settings Activity template in your app project in Android Studio, follow these steps:

1. Choose **New > Activity > Settings Activity**.
2. In the dialog that appears, accept the Activity Name (**SettingsActivity** is the suggested name) and the Title (**Settings**).
3. Click the three dots at the end of the Hierarchical Parent field and choose the parent activity (usually **MainActivity**), so that Up navigation in the Settings Activity returns the user to the MainActivity. Choosing the parent activity automatically updates the AndroidManifest.xml file to support Up navigation.

The Settings Activity template creates the XML files in the **res > xml** directory, which you can add to or customize for the settings you want:

- pref_data_sync.xml: PreferenceScreen layout for "Data & Sync" settings.
- pref_general.xml: PreferenceScreen layout for "General" settings.
- pref_headers.xml: Layout of headers for the Settings main screen.
- pref_notification.xml: PreferenceScreen layout for "Notifications" settings.

  The above XML layouts use various subclasses of the Preference class rather than View objects, and direct subclasses provide containers for layouts involving multiple settings. For example, PreferenceScreen represents a top- level Preference that is the root of a Preference hierarchy. The above files use PreferenceScreen at the top of each screen of settings. Other Preference subclasses for settings provide the appropriate UI for users to change the setting. For example:

  - CheckBoxPreference: A checkbox for a setting that is either enabled or disabled.
  - ListPreference: A dialog with a list of radio buttons.
  - SwitchPreference: A two-state toggleable option (such as on/off or true/false).
  - EditTextPreference: A dialog with an EditText widget.
  - RingtonePreference: A dialog with ringtones on the device.

The Settings Activity template also provides the following:

- String resources in the strings.xml file in the **res > values** directory, which you can customize for the settings you want.

  All strings used in the Settings Activity, such as the titles for settings, string arrays for lists, and descriptions for settings, are defined as string resources at the end of this file. They are marked by comments such as `<!-- Strings related to Settings -->` and `<!-- Example General settings -->`.

  **Tip**: You can edit these strings to customize the settings you need for your app.

- SettingsActivity in the **java > com.example.android.*projectname*** directory, which you can use as is.

  The activity that displays the settings. `SettingsActivity` extends `AppCompatPreferenceActivity` for maintaining compatibility with older versions of Android.

- AppCompatPreferenceActivity in the **java > com.example.android.*projectname*** directory, which you use as is.

  This activity is a helper class that SettingsActivity uses to maintain backwards compatibility with previous versions of Android.

## Using preference headers

The Settings Activity template shows preference headers on the main screen that separate the settings into categories (**General**, **Notifications**, and **Data & sync**). The user taps a heading to access the settings under that heading. On larger tablet displays (see previous figure), the headers appear in the left pane and the settings for each header appears in the right pane.

To implement the headers, the template provides the pref_headers.xml file:

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
   <header
      android:fragment=
           "com.example.android.droidcafe.SettingsActivity$GeneralPreferenceFragment"
      android:icon="@drawable/ic_info_black_24dp"
      android:title="@string/pref_header_general" />

   <header
      android:fragment=
      "com.example.android.droidcafe.SettingsActivity$NotificationPreferenceFragment"
      android:icon="@drawable/ic_notifications_black_24dp"
      android:title="@string/pref_header_notifications" />

   <header
      android:fragment=
          "com.example.android.droidcafe.SettingsActivity$DataSyncPreferenceFragment"
      android:icon="@drawable/ic_sync_black_24dp"
      android:title="@string/pref_header_data_sync" />
</preference-headers>
```

The XML headers file lists each preferences category and declares the fragment that contains the corresponding preferences.

To display the headers, the template uses the following `onBuildHeaders()` method:

```
@Override
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public void onBuildHeaders(List<Header> target) {
        loadHeadersFromResource(R.xml.pref_headers, target);
}
```

The above code snippet uses the `loadHeadersFromResource()` method of the PreferenceActivity class to load the headers from the XML resource (pref_headers.xml). The TargetApi annotation tells Android Studio's Lint code scanning tool that the class or method is targeting a particular API level regardless of what is specified as the min SDK level in manifest. Lint would otherwise produce errors and warnings when using new functionality that is not available in the target API level.

## Using PreferenceActivity with fragments

The Settings Activity template provides an activity (SettingsActivity) that extends PreferenceActivity to create a two-pane layout to support large screens, and *also* includes fragments within the activity to display lists of settings. This is a useful pattern if you have multiple groups of settings and need to support tablet-sized screens as well as smartphones.

The following shows how to use an activity that extends `PreferenceActivity` to host one or more fragments (PreferenceFragment) that display app settings. The activity can host multiple fragments, such as `GeneralPreferenceFragment` and `NotificationPreferenceFragment`, and each fragment definition uses `addPreferencesFromResource` to load the settings from the XML preferences file:

```
public class SettingsActivity extends AppCompatPreferenceActivity {
   ...
   public static class GeneralPreferenceFragment extends
                                              PreferenceFragment {
   @Override
   public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            addPreferencesFromResource(R.xml.pref_general);

   ...
   }
   public static class NotificationPreferenceFragment extends
                                              PreferenceFragment {
   ...
}
```

## Module Reference

ANDROID ★ DEVELOPER ★ FUNDAMENTALS
Android Developer Fundamentals Course
Learn to develop Android Applications
Developed by Google Developer Training Team