Unit 4: Saving User Data

Lesson 9: Preferences and settings

9.0: Data storage

Contents:

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

Your data storage options include the following:

- Shared preferences: Store private primitive data in key-value pairs. This is 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.
- Room persistence library: Part of the Android Architecture Component libraries. Room caches an SQLite database locally, and automatically syncs changes to a network database
- Cloud backup: Back up your app and user data in the cloud.
- 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.
- Custom data store: Configure the Preference APIs to store preferences in a storage location you provide.

Shared preferences

Using shared preferences is a way to read and write key-value pairs of information persistently to and from a file. Shared Preferences is covered in its own chapter.

Note: The Preference APIs are used to create and store user settings for your app. Although the Preference APIs use shared preferences to store their data, they are not the same thing. You will learn more about settings and preferences in a later chapter.

Files

Android uses a file system that's similar to disk-based file systems on other platforms such as Linux. File-based operations should be familiar to anyone who has used Linux file I/O or the `java.io` package.
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 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.

Today, some devices divide the permanent storage space into "internal" and "external" partitions, so even without a removable storage medium, there are always two storage spaces and the API behavior is the same whether the external storage is removable or not. The following lists summarize the facts about each storage space.

Internal storage

You don't need any permissions to save files on the internal storage. Your app 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 totaling 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

Use external storage for files that should be permanently stored, even if your app is uninstalled, and be available freely to other users and apps, such as pictures, drawings, or documents made by your app.

Some private files that are of no value to other apps can also be stored on external storage. Such files might be additional downloaded app resources, or temporary media files. Make sure you delete those 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 Android manifest. 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 by the Android system for various purposes. There are public 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 topic is not Android-specific, so it's 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 SQLite Primer chapter gives you an overview on using an SQLite database.

Room persistence library

The Room persistence library provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.

The library helps you create a cache of your app's data on a device that's running your app. This cache, which serves as your app's single source of truth, allows users to view a consistent copy of key information within your app, regardless of whether users have an internet connection.

You will learn more about Room in the Room, LiveData, and ViewModel chapter. For more information see the Room training guide.

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 Learn More section 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 often invest significant time and effort creating data and setting preferences within apps. Preserving that data for users if they replace a broken device or upgrade to a new one is an important part of ensuring a great user experience.

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

For apps whose target SDK version is Android 6.0 (API level 23) and higher, devices running Android 6.0 and higher automatically backup app data to the cloud. The system performs this automatic backup for nearly all app data by default, and does so without you writing any additional app code.

When a user installs your app on a new device, or re-installs your app on one (for example, after a factory reset), the system automatically restores the app data from the cloud. The automatic backup feature preserves the data your app creates on a user device by uploading it to the user's Google Drive account and encrypting it. There is no charge to you or the user for data storage, and the saved data does not count towards the user's personal Google Drive quota. Each app can store up to 25MB. Once its backed-up data reaches 25MB, the app no longer sends data to the cloud. If the system performs a data restore, it uses the last data snapshot that the app had 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 API 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.

Custom data store for preferences

By default, the `Preference` class stores its values into the `SharedPreferences` interface, which is the recommended way to persist user preferences. However, providing a custom data store to your preferences can be useful if your app stores the preferences in a cloud or local database, or if the preferences are device-specific. On devices running Android 8.0 (API level 26) or higher, you can achieve this by providing any `Preference` object with your implementation of the `PreferenceDataStore` interface.
See Setting up a custom data store for more information.

Learn more

Files:

- Save files on device storage
- `getExternalFilesDir()` documentation and code samples
- `getExternalStoragePublicDirectory()` documentation and code samples
- `java.io.File` class
- Oracle's Java I/O Tutorial

Backup:

- Data backup overview
- Back up user data with Auto Backup
- Back up key-value pairs with Android Backup Service

Shared preferences:

- Save key-value data
- Shared preferences guide
- `SharedPreferences` reference

Firebase:

- Firebase home
- Firebase Realtime Database
- Add Firebase to Your Android Project

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

9.1: Shared preferences

Contents:

Shared preferences allow you to store small amounts of primitive data as key/value pairs in a file on the device. To get a handle to a preference file, and to read, write, and manage preference data, use the `SharedPreferences` class. The Android framework manages the shared preferences file itself. The file is accessible to all the components of your app, but it is not accessible to other apps.
For managing large amounts of data, use an SQLite database or other suitable storage option, which is discussed in a later 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 stopped and restarted, or if the device is rebooted.	Preserves state data across activity instances in the same user session.
Used for data that should be remembered across user sessions, such as a user's preferred settings or their game score.	Used for data that should not be remembered across sessions, such as the currently selected tab, or any current state of an activity.
Represented by a small number of key/value pairs.	Represented by a small number of key/value pairs.
Data is private to the app.	Data is private to the app.
Common use is to store user preferences.	Common use is to recreate state after the device has been rotated.

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

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 the `SharedPreferences.Editor` interface.

1. 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.
2. Add key/value pairs to the editor using the "put" method appropriate for the data type, for example, `putInt()` or `putString()`. These methods will overwrite previously existing values of an existing key.
3. 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 such as `getInt()` or `getString()` 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.

```
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 as a class member variable in an object such as an activity 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);
```

Related practical

The related practical is 9.1: Shared preferences.

Learn more

Android developer documentation:

- Data and file storage overview
- Save key-value data
- `SharedPreferences`
- `SharedPreferences.Editor`

Stack Overflow:

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

9.2: 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

Apps 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 how often the app syncs data with the cloud.

The controls that belong in the app's settings should capture user preferences that affect most users or provide critical support to a minority of users. For example, notification settings affect all users, while a currency setting for a foreign market provides critical support for the users in that market.

Most settings are accessed infrequently, because once users change a setting, they rarely need to go back and change it again. If users need to access a control or preference frequently, consider moving the control or preference to the app bar options menu, or to a side navigation menu such as a navigation drawer.

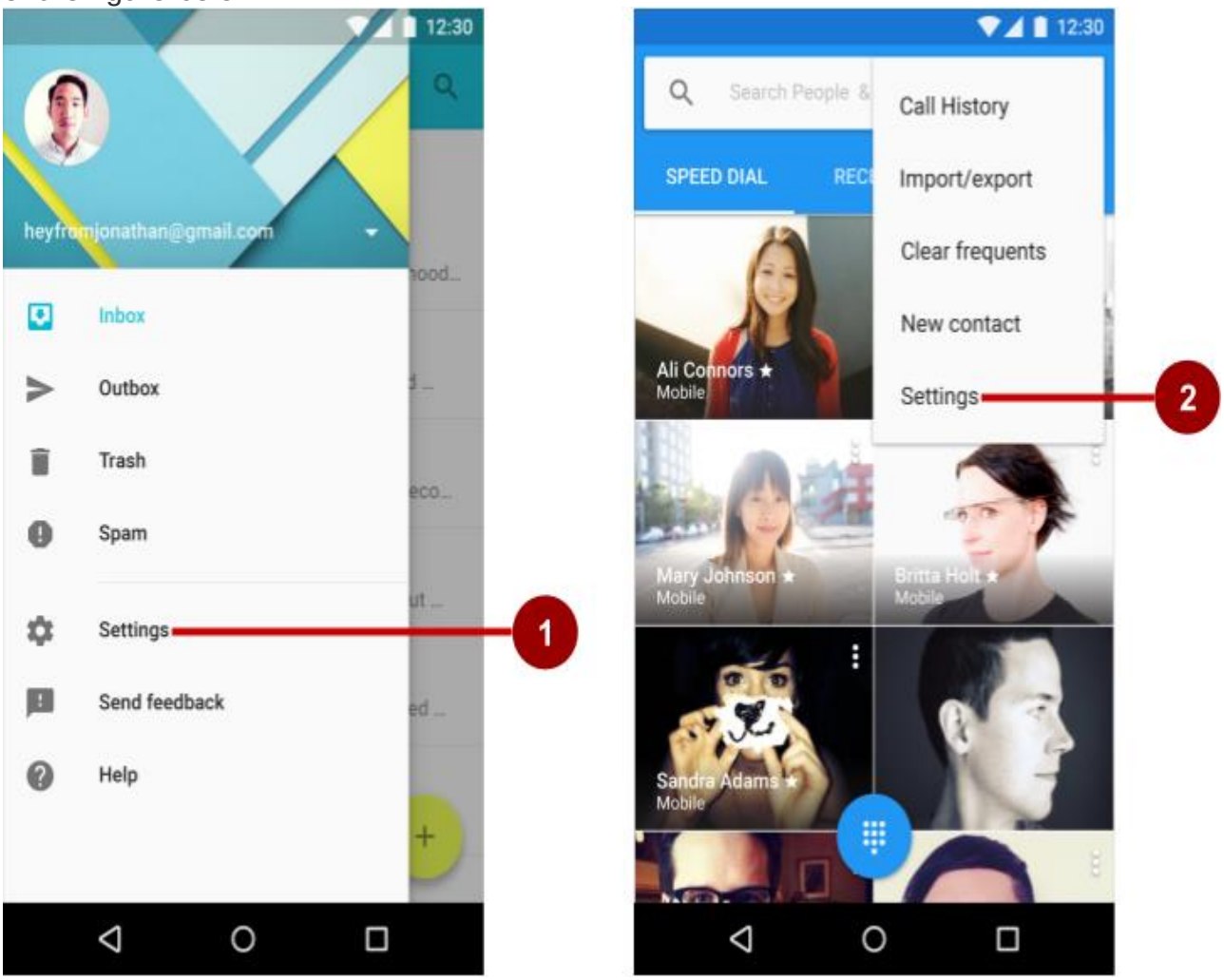
For your setting controls, set defaults that are familiar to users and make the app experience better. The initial default value for a setting should do the following:

- Represent the value most users would choose. For example, in the Contacts app, **All contacts** is the default for "Contacts to display".
- Use less battery power. For example, in the Android Settings app, Bluetooth is off until the user turns it on.
- Pose the least risk to security and data loss. For example, in the Gmail app, the 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 the setting to a separately accessed **Help** screen.

Providing navigation to settings

Users should be able to navigate to app settings by tapping **Settings**, which should be located in side navigation, such as a navigation drawer, as shown on the left side of the figure below, or in the options menu in the app bar, as shown on the right side of the figure below.



In the figure above:

1. Settings in side navigation (a navigation drawer)
2. Settings in the options menu of the app bar

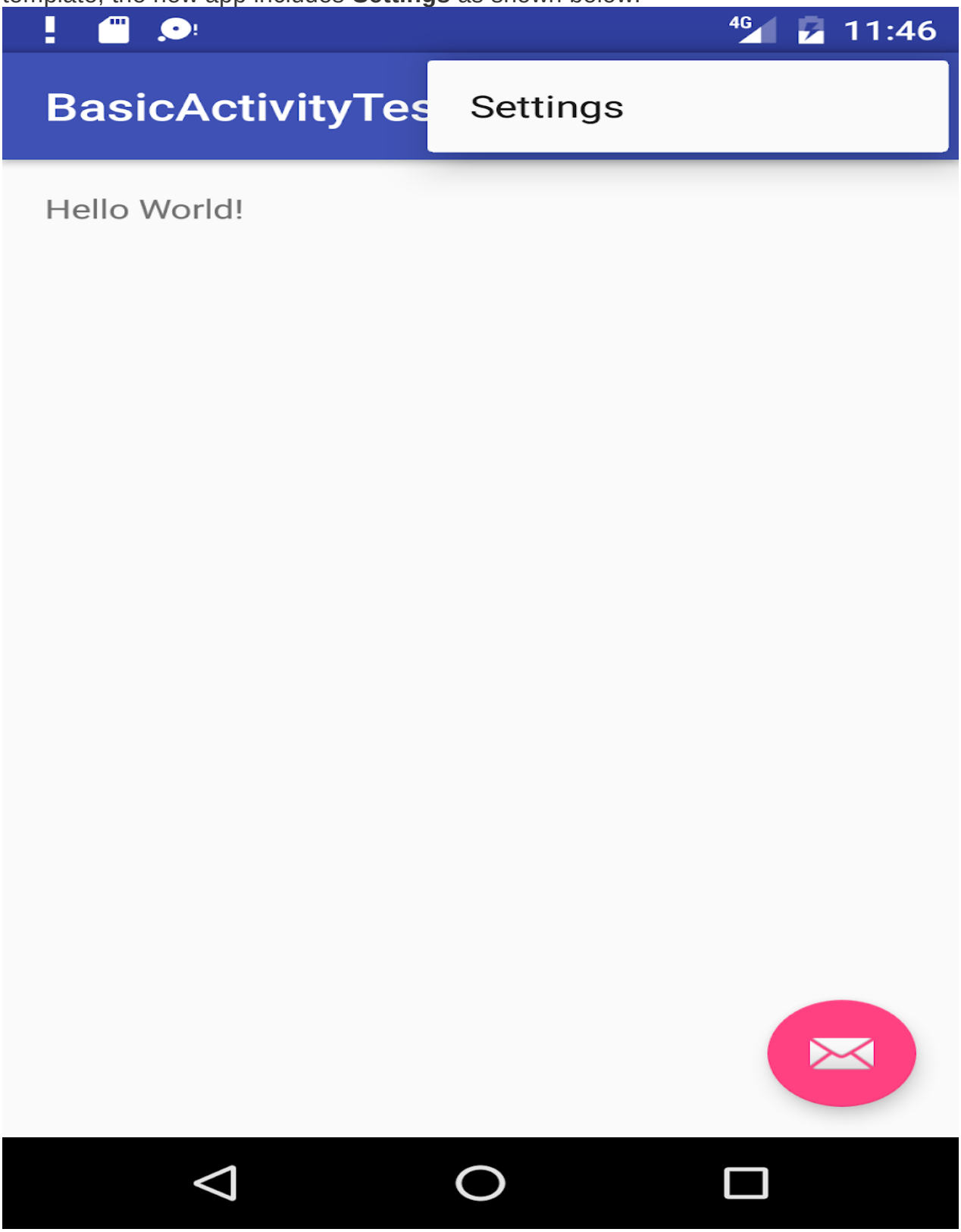
Follow these design guidelines for navigating to settings:

- 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 phone or tablet using the Basic Activity

template, the new app includes **Settings** as shown below:

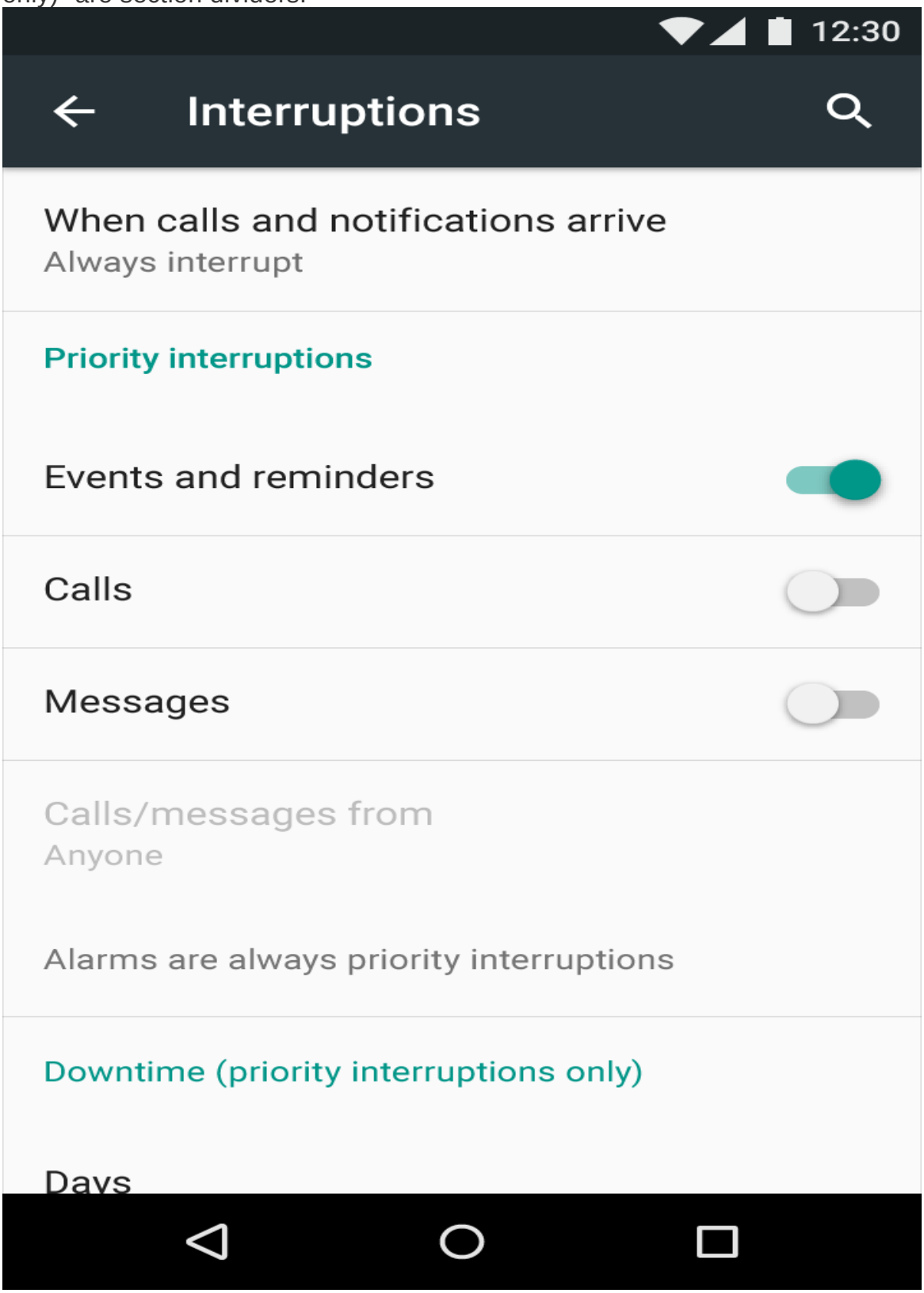


The settings UI

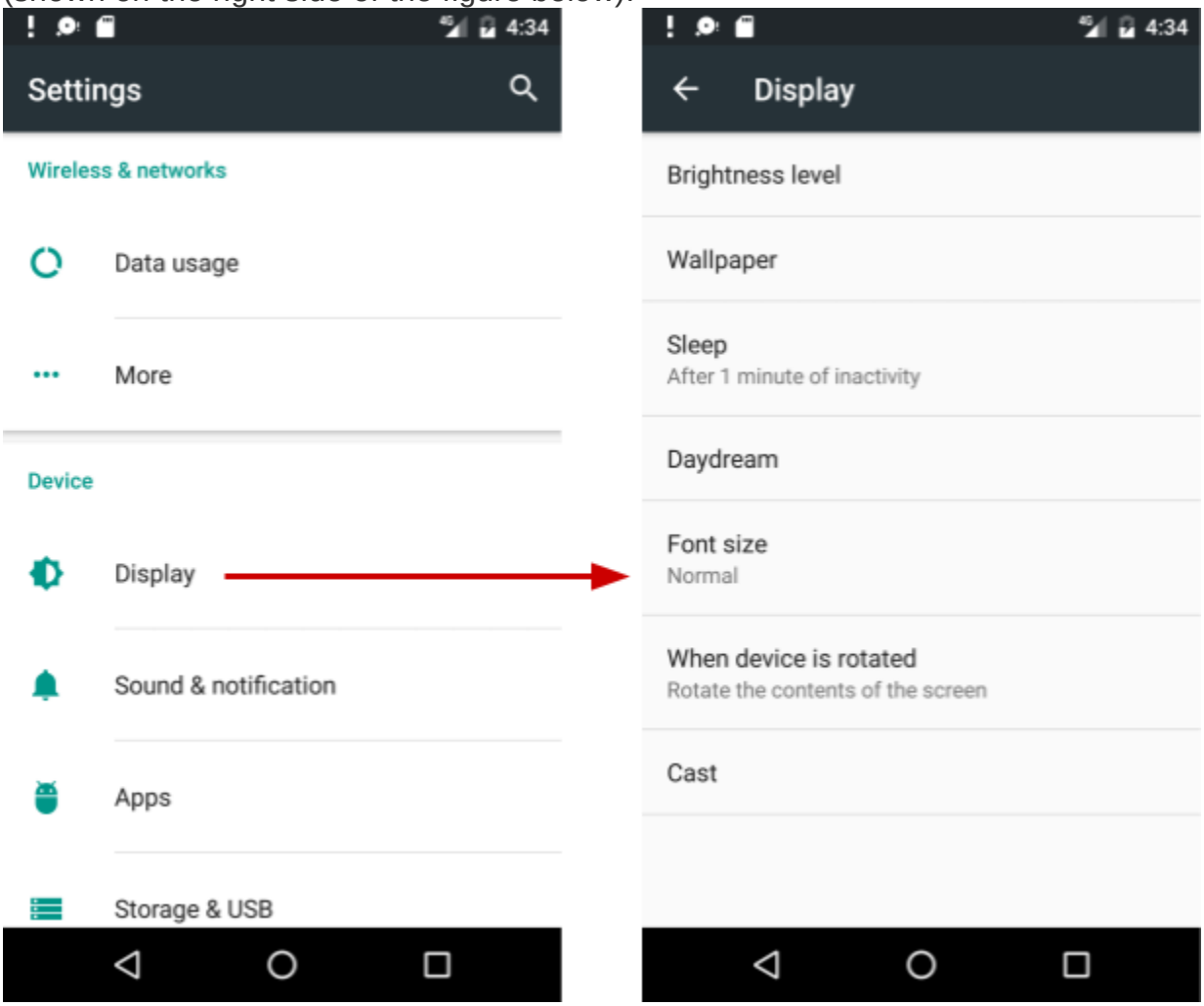
Settings should be well-organized, predictable, and contain a manageable number of options. A user should be able to quickly understand all available settings and their current values. Follow these design guidelines:

- **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.



- **16 or more settings**: Group related settings into separate 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` elements. `Preference` provides the `View` to be displayed for each setting, and associates with it a `SharedPreferences` interface to store and 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.

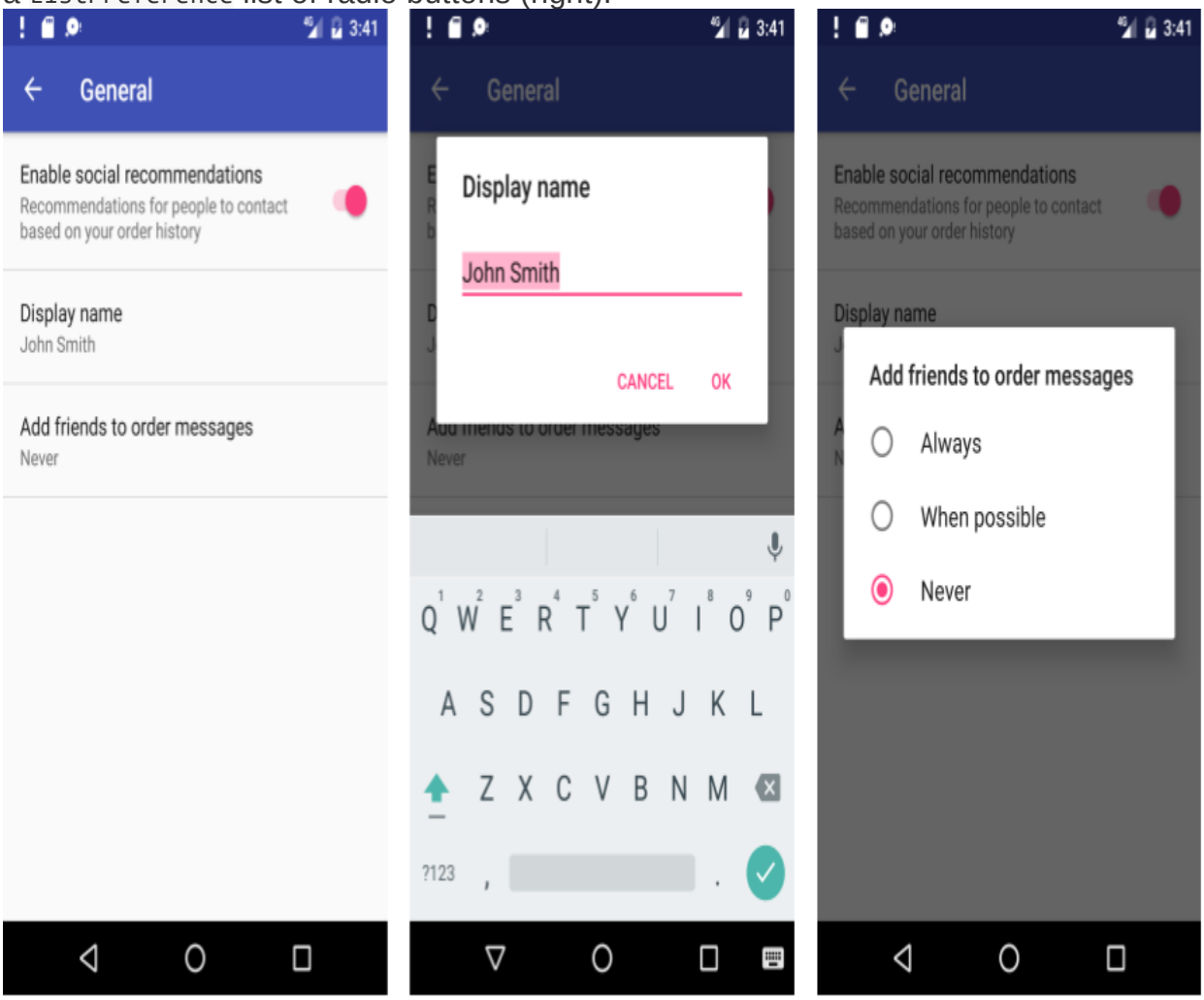
For example, to provide dividers with headings between groups of settings (as shown in the previous figure for 7-15 settings), place each group of `Preference` objects inside a `PreferenceCategory`. To use separate screens for groups, place each group of `Preference` objects inside a `PreferenceScreen`.
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 option that can be toggled (such as on/off or true/false).
- `EditTextPreference`: Creates an item that opens a dialog with an `EditText` element. The saved value is a `String`.
- `RingtonePreference`: Lets the user 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 `SwitchPreference` toggle switch (at the top of the screen on the left side), an `EditTextPreference` text entry field (center), and a `ListPreference` list of radio buttons (right):



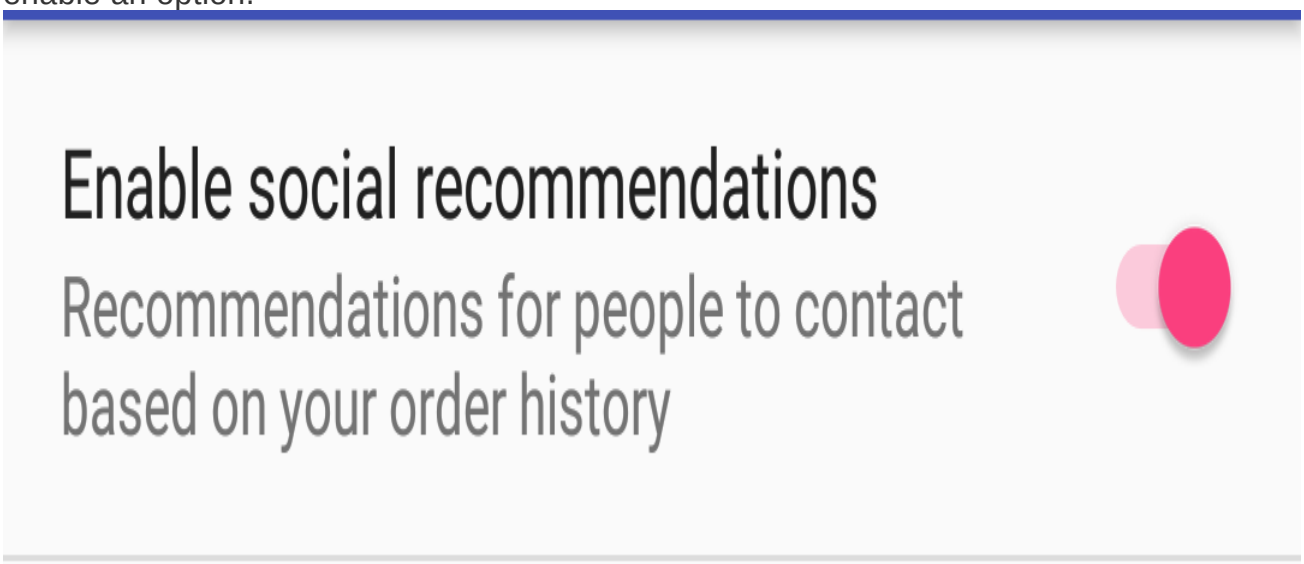
The root of the settings hierarchy is a `PreferenceScreen` layout:

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

    <android.support.v7.preference.SwitchPreferenceCompat
        android:defaultValue="true"
        android:key="example_switch"
        android:summary="@string/switch_summary"
        android:title="@string/switch_title" />

</PreferenceScreen>
```

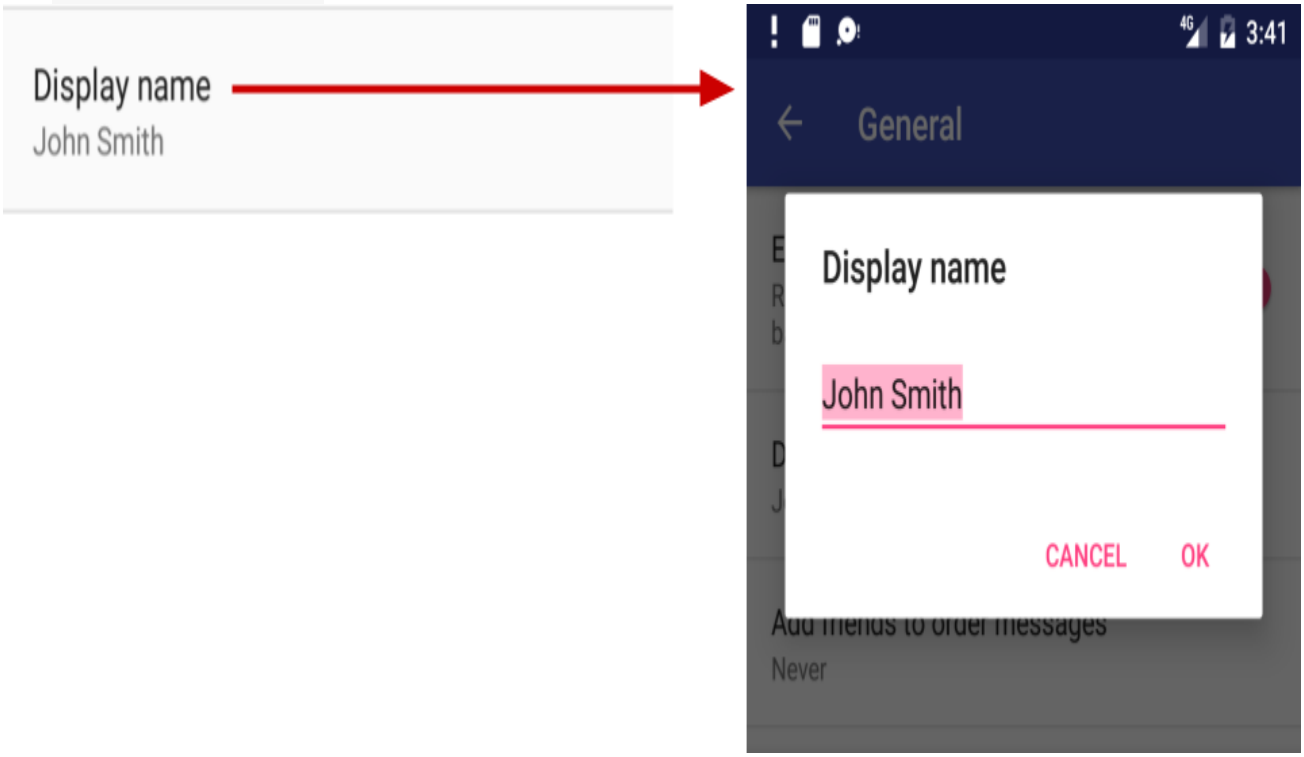
Inside the above layout is a `SwitchPreference` that shows a toggle switch to disable or enable an option.



The setting has the following attributes:

- `android:defaultValue`: The option is enabled (set to `true`) by default.
- `android:key`: The key to use for storing the setting value. Each setting (`Preference`) has a corresponding key-value pair that the system uses to save the setting in a default `SharedPreferences` file for your app's settings.
- `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.

An `EditTextPreference` shows a text field for the user to enter text.

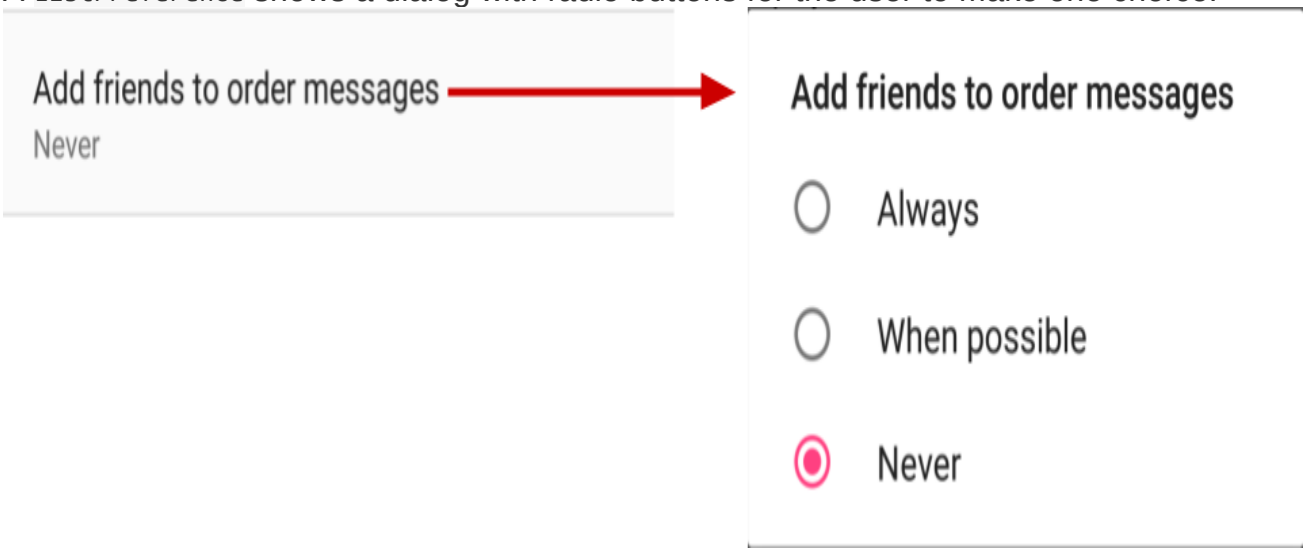


- Use `EditText` attributes such as `android:capitalize` and `android:maxLines` to define the text field's appearance and input control.
- The default setting is the `pref_default_display_name` string resource.

Example:

```
<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" />
```

A `ListPreference` shows 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"`.

Example:

```
<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_list_title" />
```

Save the XML file in the `res/xml` directory. Although you can name the file anything you want, it is 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:

```
<android.support.v7.preference.SwitchPreferenceCompat
```

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`, and 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.

`A Fragment` like `PreferenceFragment` provides a more flexible architecture for your app, compared to using an `Activity` 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:

```
implementation 'com.android.support:preference-v7:26.1.0'
```

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">
   <!-- Other items -->
   <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. In the `Fragment`, 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`.

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`.

To set Up navigation for the `SettingsActivity`, be sure to declare the `SettingsActivity` 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 practical, 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"
```

4. From the `onCreate()` method in `MainActivity`—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:

7. The app `context`, such as `this`.
8. The resource ID (`preferences`) for the settings layout XML file which includes the default values set by Step 1 above.
9. 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 `MainActivity` starts without overriding the user's saved settings 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 `MainActivity` `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 another practical, 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";
 }
```

7. 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) {
    // ... Rest of onCreate code.
    mPreferences = this.getActivity()
            .getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
    Preference preference =
            this.findPreference("example_switch");
    preference.setSummary(mPreferences.getString("summary",
            getString(R.string.option_on)));
 }
```

18. 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 listens to a change in the switch setting using `onPreferenceChange()`, and returns `true`. It then determines the new `boolean` value (`newValue`) for the setting after the change (`true` or `false`).

If `true`, the code edits the `SharedPreferences` file (as described in a previous practical) using `SharedPreferences.Editor`, putting the new value (`string.option.on`) as a string in the summary using `putString()` and applies the change using `apply()`. If `false`, the code does the same thing with the new value (`string.option.off`) as a string in the summary.

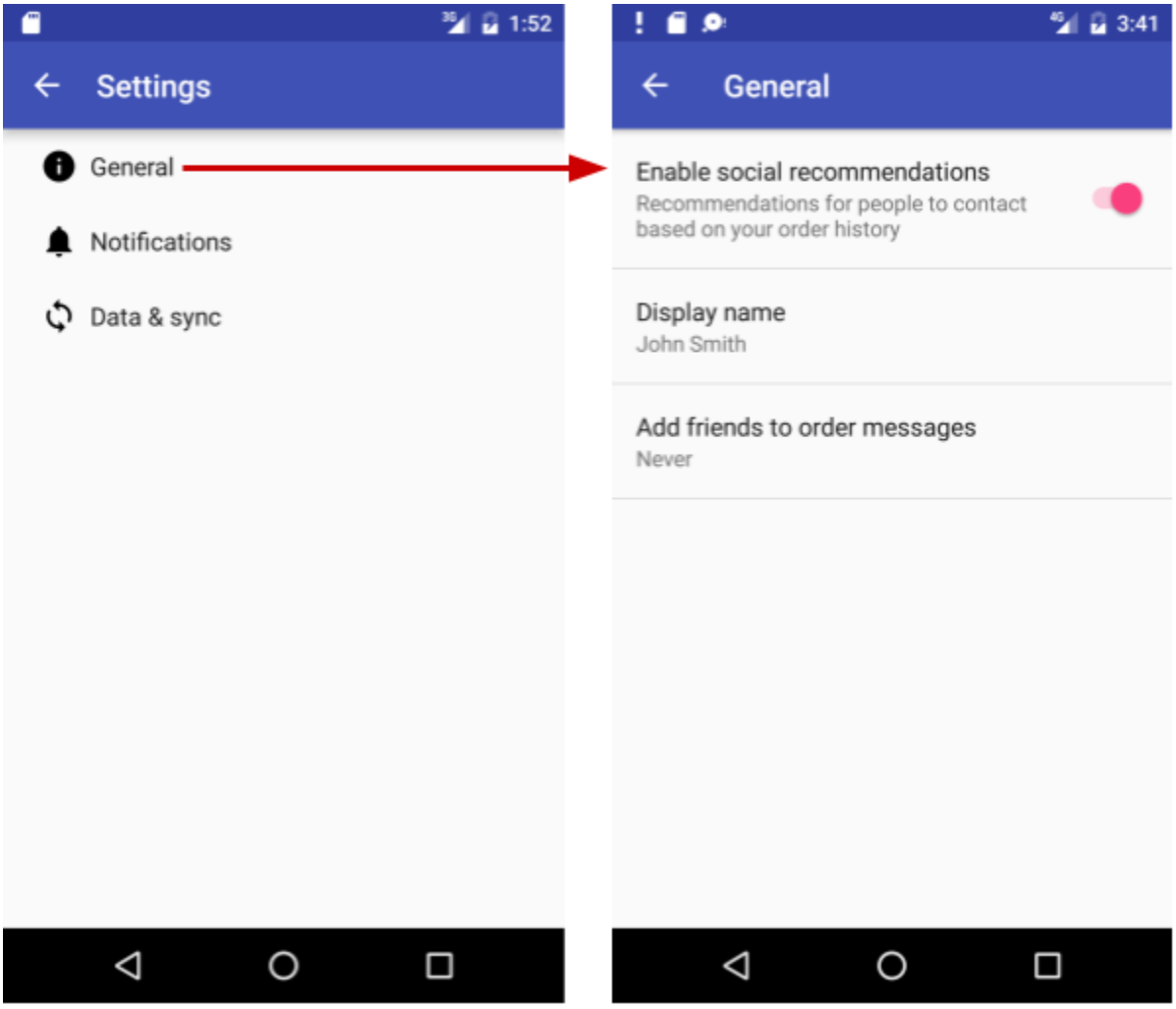
https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

Using the Settings Activity template

If you need to build several 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 populated with settings you can customize for an app, and provides a different layout for phones and tablets:

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

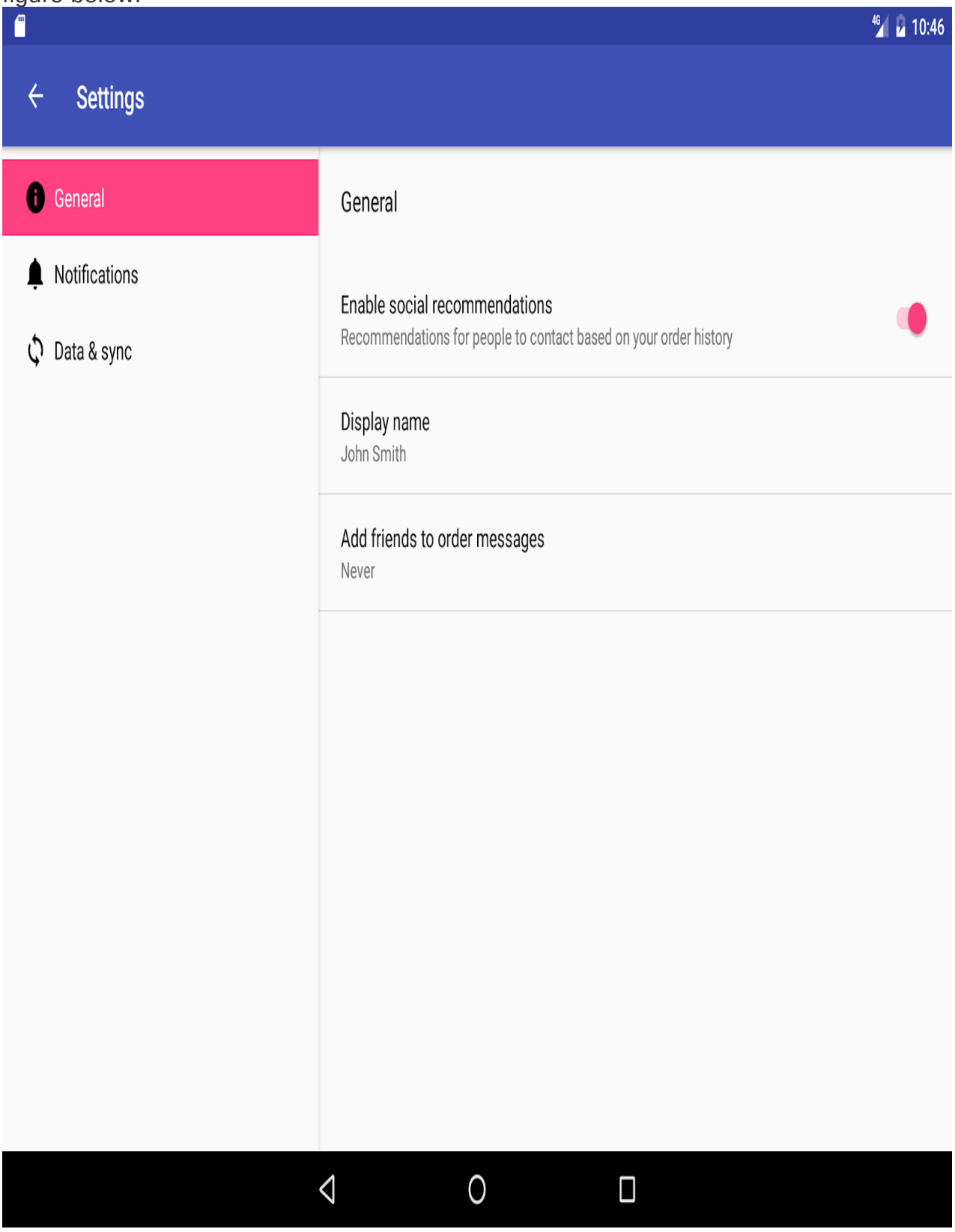


https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

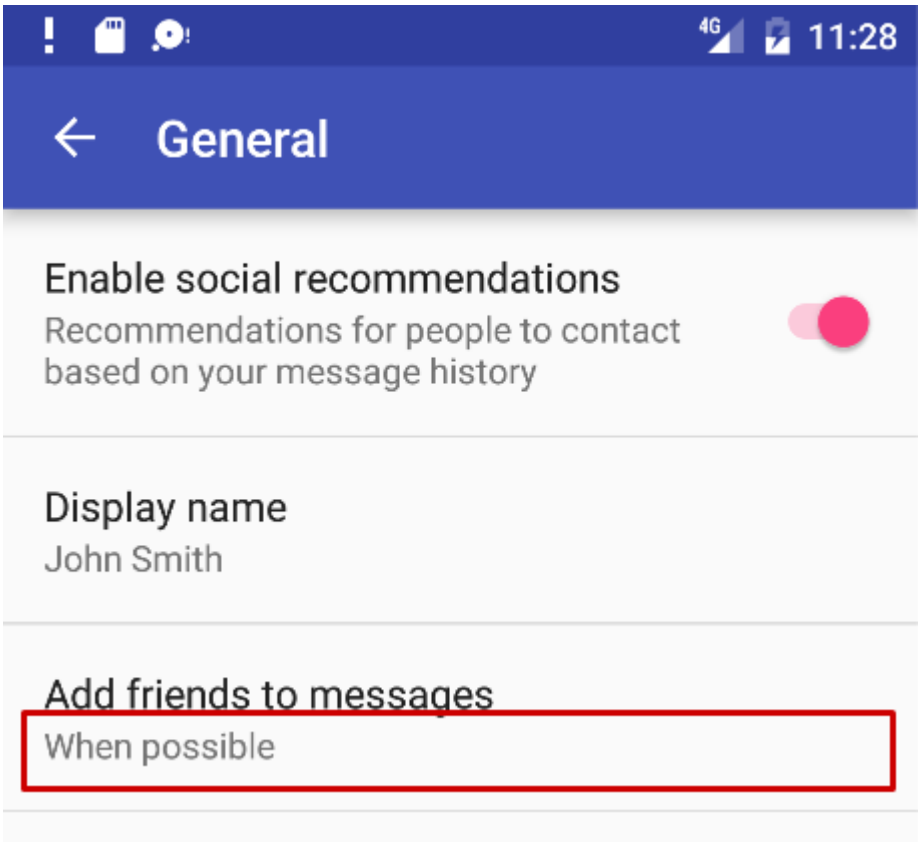
- *Tablets*: A master/detail screen layout with a header link for each group on the left (master) side, and the group of settings on the right (detail) side, as shown in the

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

figure below.



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:



In general, you need not change the Settings Activity template code in order to customize `SettingsActivity` 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 `res > xml` folder in the **Project > Android** pane.

You use the Settings Activity template code as-is. To make it work for your app, add code to `MainActivity` 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. Select **app** at the top of the **Project > Android** pane, and 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 the **Up** app bar button in `SettingsActivity` returns the user to `MainActivity`. Choosing the parent `Activity` automatically updates the `AndroidManifest.xml` file to support **Up** navigation.

The Settings Activity template creates XML files in the **res > xml** folder of the **Project > Android** pane, 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`, 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 option that can be toggled (such as on/off or true/false).
- `EditTextPreference`: A dialog with an `EditText`.
- `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` folder of the **Project > Android** pane, 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* folder, which you can use as is. This 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* folder, 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$GeneralPreference
Fragment"
      android:icon="@drawable/ic_info_black_24dp"
      android:title="@string/pref_header_general" />

   <header

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

   <header

android:fragment="com.example.android.droidcafe.SettingsActivity$DataSyncPreferenc
eFragment"
      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 a Fragment

The Settings Activity template provides an `Activity` (`SettingsActivity`) that extends `PreferenceActivity` to create a two-pane layout to support large screens, and *also* includes a `Fragment` for each group of settings. This is a useful pattern if you have multiple groups of settings and need to support tablet-sized screens as well as phones.

The following shows how to use an `Activity` that extends `PreferenceActivity` to host a `Fragment` (`PreferenceFragment`) that displays a group of settings. The `Activity` can host more than one `Fragment`, 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 {
   // ... SettingsActivity code

   public static class GeneralPreferenceFragment extends
                                              PreferenceFragment {
       @Override
       public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            addPreferencesFromResource(R.xml.pref_general);
            // ... onCreate code
   }
   public static class NotificationPreferenceFragment extends
                                              PreferenceFragment {
       // ... Similar code as in above fragment
}
```

Related practical

The related practical is 9.2: App settings.

Learn more

Android Studio documentation: Meet Android Studio

Android developer documentation:

- Settings (overview)
- `Preference`
- `PreferenceFragment`
- `PreferenceFragmentCompat`
- `Fragment`
- `SharedPreferences`
- Save key-value data
- Support different screen sizes

Material Design specification: Android settings

Stack Overflow:

- How does one get dimens.xml into Android Studio?
- Determine if the device is a smartphone or tablet?