Settings

Applications often include settings that allow users to modify app features and behaviors. For example, some apps allow users to specify whether notifications are enabled or specify how often the application syncs data with the cloud.

If you want to provide settings for your app, you should use Android's Preference (/reference/android/preference/Preference.html) APIs to build an interface that's consistent with the user experience in other Android apps (including the system settings). This document describes how to build your app settings using Preference (/reference/android/preference/Preference.html) APIs.

Settings Design

For information about how to design your settings, read the Settings (/design/patterns/settings.html) design guide.

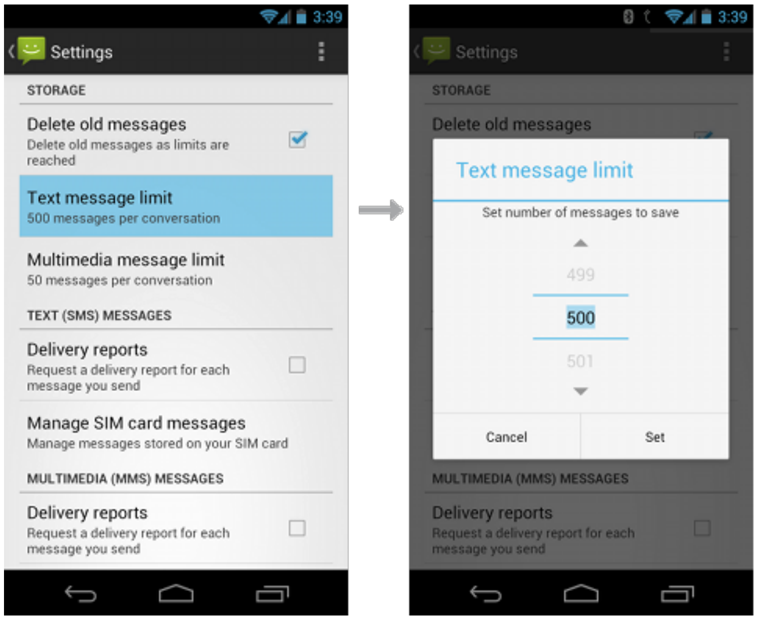
3:39
Settings
STORAGE
Delete old messages
Delete old messages as limits are reached
Text message limit
500 messages per conversation
Multimedia message limit
50 messages per conversation
TEXT (SMS) MESSAGES
Delivery reports
Request a delivery report for each message you send
Manage SIM card messages
Manage messages stored on your SIM card
MULTIMEDIA (MMS) MESSAGES
Delivery reports
Request a delivery report for each message you send
Text message limit
Set number of messages to save
499
500
501
Cancel
Set

Figure 1. Screenshots from the Android Messaging app's settings. Selecting an item defined by a Preference (/reference/android/preference/Preference.html) opens an interface to change the setting.

KEY CLASSES

- Preference
- PreferenceActivity
- PreferenceFragment

Overview

Instead of using View (/reference/android/view/View.html) objects to build the user interface, settings are built using various subclasses of the Preference (/reference/android/preference/Preference.html) class that you declare in an XML file.

A Preference (/reference/android/preference/Preference.html) object is the building block for a single setting. Each Preference (/reference/android/preference/Preference.html) appears as an item in a list and provides the appropriate UI for users to modify the setting. For example, a CheckBoxPreference (/reference/android/preference/CheckBoxPreference.html) creates a list item that shows a checkbox, and a ListPreference (/reference/android/preference/ListPreference.html) creates an item that opens a dialog with a list of choices.

SEE ALSO

Settings design guide

Each Preference (/reference/android/preference/Preference.html) you add has a corresponding key-value pair that the system uses to save the setting in a default SharedPreferences (/reference/android/content/SharedPreferences.html) file for your app's settings. When the user changes a setting, the system updates the corresponding value in the SharedPreferences (/reference/android/content/SharedPreferences.html) file for you. The only time you should directly interact with the associated SharedPreferences (/reference/android/content/SharedPreferences.html) file is when you need to read the value in order to determine your app's behavior based on the user's setting.

The value saved in SharedPreferences (/reference/android/content/SharedPreferences.html) for each setting can be one of the following data types:

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

Because your app's settings UI is built using Preference (/reference/android/preference/Preference.html) objects instead of View (/reference/android/view/View.html) objects, you need to use a specialized Activity (/reference/android/app/Activity.html) or Fragment (/reference/android/app/Fragment.html) subclass to display the list settings:

- If your app supports versions of Android older than 3.0 (API level 10 and lower), you must build the activity as an extension of the PreferenceActivity class.
- On Android 3.0 and later, you should instead use a traditional Activity that hosts a PreferenceFragment that displays your app settings. However, you can also use PreferenceActivity to create a two-pane layout for large screens when you have multiple groups of settings.

How to set up your PreferenceActivity (/reference/android/preference/PreferenceActivity.html) and instances of PreferenceFragment (/reference/android/preference/PreferenceFragment.html) is discussed in the sections about Creating a Preference Activity (#Activity) and Using Preference Fragments (#Fragment).

Preferences

Every setting for your app is represented by a specific subclass of the Preference (/reference/android/preference/Preference.html) class. Each subclass includes a set of core properties that allow you to specify things such as a title for the setting and the default value. Each subclass also provides its own specialized properties and user interface. For instance, figure 1 shows a screenshot from the Messaging app's settings. Each list item in the settings screen is backed by a different Preference (/reference/android/preference/Preference.html) object.

A few of the most common preferences are:

CheckBoxPreference
Shows an item with a checkbox for a setting that is either enabled or disabled. The saved value is a

boolean (`true` if it's checked).

`ListPreference`
: Opens a dialog with a list of radio buttons. The saved value can be any one of the supported value types (listed above).

`EditTextPreference`
: Opens a dialog with an `EditText` widget. The saved value is a `String`.

See the `Preference` (/reference/android/preference/Preference.html) class for a list of all other subclasses and their corresponding properties.

Of course, the built-in classes don't accommodate every need and your application might require something more specialized. For example, the platform currently does not provide a `Preference` (/reference/android/preference/Preference.html) class for picking a number or a date. So you might need to define your own `Preference` (/reference/android/preference/Preference.html) subclass. For help doing so, see the section about Building a Custom Preference (#Custom).

Defining Preferences in XML

Although you can instantiate new `Preference` (/reference/android/preference/Preference.html) objects at runtime, you should define your list of settings in XML with a hierarchy of `Preference` (/reference/android/preference/Preference.html) objects. Using an XML file to define your collection of settings is preferred because the file provides an easy-to-read structure that's simple to update. Also, your app's settings are generally pre-determined, although you can still modify the collection at runtime.

Each `Preference` (/reference/android/preference/Preference.html) subclass can be declared with an XML element that matches the class name, such as `<CheckBoxPreference>`.

You must save the XML file in the `res/xml/` directory. Although you can name the file anything you want, it's traditionally named `preferences.xml`. You usually need only one file, because branches in the hierarchy (that open their own list of settings) are declared using nested instances of `PreferenceScreen` (/reference/android/preference/PreferenceScreen.html).

> **Note**: If you want to create a multi-pane layout for your settings, then you need separate XML files for each fragment.

The root node for the XML file must be a `<PreferenceScreen>` (/reference/android/preference/PreferenceScreen.html) element. Within this element is where you add each `Preference` (/reference/android/preference/Preference.html). Each child you add within the `<PreferenceScreen>` (/reference/android/preference/PreferenceScreen.html) element appears as a single item in the list of settings.

For example:

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

In this example, there's a `CheckBoxPreference` (/reference/android/preference/CheckBoxPreference.html) and a `ListPreference` (/reference/android/preference/ListPreference.html). Both items include the following three attributes:

`android:key`
: This attribute is required for preferences that persist a data value. It specifies the unique key (a string) the system uses when saving this setting's value in the `SharedPreferences`.

  The only instances in which this attribute is *not required* is when the preference is a `PreferenceCategory` (/reference/android/preference/PreferenceCategory.html) or `PreferenceScreen` (/reference/android/preference/PreferenceScreen.html), or the preference specifies an `Intent` (/reference/android/content/Intent.html) to invoke (with an `<intent>` (#Intents) element) or a `Fragment` (/reference/android/app/Fragment.html) to display (with an `android:fragment` (/reference/android/preference/Preference.html#attr_android:fragment) attribute).

`android:title`
: This provides a user-visible name for the setting.

`android:defaultValue`
: This specifies the initial value that the system should set in the `SharedPreferences` file. You should supply a default value for all settings.

For information about all other supported attributes, see the `Preference` (/reference/android/preference/Preference.html) (and respective subclass) documentation.

When your list of settings exceeds about 10 items, you might want to add titles to define groups of settings or display those groups in a separate screen. These options are described in the following sections.

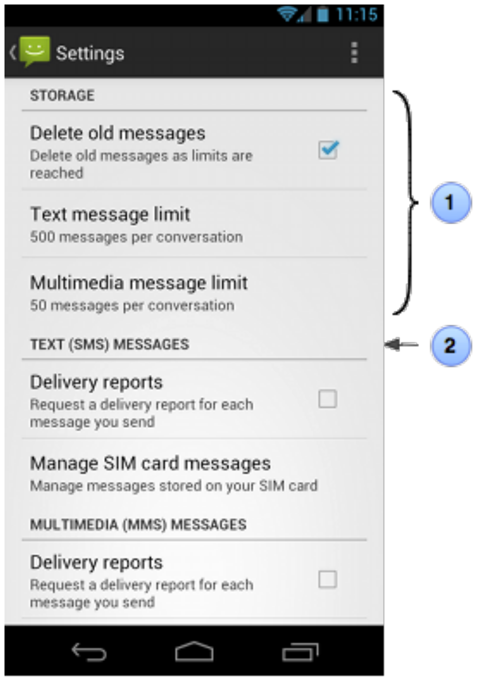
11:15
Settings
STORAGE
Delete old messages
Delete old messages as limits are reached
Text message limit
500 messages per conversation
Multimedia message limit
50 messages per conversation
TEXT (SMS) MESSAGES
Delivery reports
Request a delivery report for each message you send
Manage SIM card messages
Manage messages stored on your SIM card
MULTIMEDIA (MMS) MESSAGES
Delivery reports
Request a delivery report for each message you send
1
2

Figure 2. Setting categories with titles.
1. The category is specified by the

Creating setting groups

If you present a list of 10 or more settings, users may have difficulty scanning, comprehending, and processing them. You can remedy this by dividing some or all of the settings into groups, effectively turning one long list into multiple shorter lists. A group of related settings can be presented in one of two ways:

- Using titles
- Using subscreens

You can use one or both of these grouping techniques to organize your app's settings. When deciding which to use and how to divide your settings, you should follow the guidelines in Android Design's Settings (/design/patterns/settings.html) guide.

Using titles

If you want to provide dividers with headings between groups of settings (as shown in figure 2), place each group of `Preference` (/reference/android/preference/Preference.html) objects inside a `PreferenceCategory` (/reference/android/preference/PreferenceCategory.html).

`<PreferenceCategory>` (/reference/android/preference/PreferenceCategory.html) element.
2. The title is specified with the `android:title` attribute.

For example:

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

Using subscreens

If you want to place groups of settings into a subscreen (as shown in figure 3), place the group of `Preference` (/reference/android/preference/Preference.html) objects inside a `PreferenceScreen` (/reference/android/preference/PreferenceScreen.html).

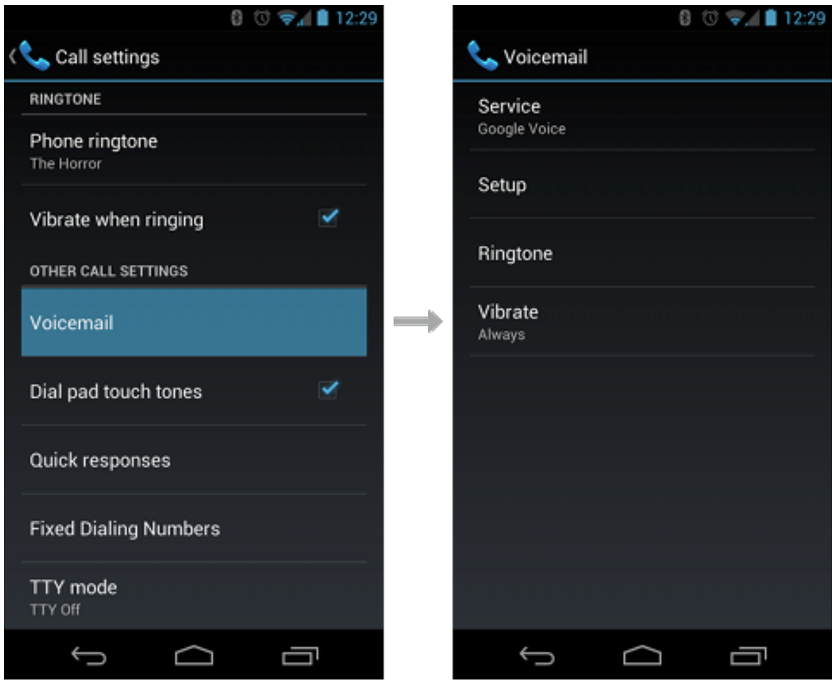

Figure 3. Setting subscreens. The `<PreferenceScreen>` element creates an item that, when selected, opens a separate list to display the nested settings.

For example:

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

Using intents

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

For example, here's how you can use a preference item to open a web page:

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

You can create both implicit and explicit intents using the following attributes:

`android:action`
: The action to assign, as per the setAction() method.

`android:data`
: The data to assign, as per the setData() method.

`android:mimeType`
: The MIME type to assign, as per the setType() method.

`android:targetClass`
: The class part of the component name, as per the setComponent() method.

`android:targetPackage`
: The package part of the component name, as per the setComponent() method.

Creating a Preference Activity

To display your settings in an activity, extend the PreferenceActivity (/reference/android/preference/PreferenceActivity.html) class. This is an extension of the traditional Activity (/reference/android/app/Activity.html) class that displays a list of settings based on a hierarchy of Preference (/reference/android/preference/Preference.html) objects. The PreferenceActivity (/reference/android/preference/PreferenceActivity.html) automatically persists the settings associated with each Preference (/reference/android/preference/Preference.html) when the user makes a change.

> **Note:** If you're developing your application for Android 3.0 and higher, you should instead use PreferenceFragment (/reference/android/preference/PreferenceFragment.html). Go to the next section about Using Preference Fragments (#Fragment).

The most important thing to remember is that you do not load a layout of views during the onCreate() (/reference/android/preference/PreferenceActivity.html#onCreate(android.os.Bundle)) callback. Instead, you call addPreferencesFromResource() (/reference/android/preference/PreferenceActivity.html#addPreferencesFromResource(int)) to add the preferences you've declared in an XML file to the activity. For example, here's the bare minimum code required for a functional PreferenceActivity (/reference/android/preference/PreferenceActivity.html):

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

This is actually enough code for some apps, because as soon as the user modifies a preference, the system saves the changes to a default `SharedPreferences` (/reference/android/content/SharedPreferences.html) file that your other application components can read when you need to check the user's settings. Many apps, however, require a little more code in order to listen for changes that occur to the preferences. For information about listening to changes in the `SharedPreferences` (/reference/android/content/SharedPreferences.html) file, see the section about Reading Preferences (#ReadingPrefs).

Using Preference Fragments

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

Fragments (/guide/components/fragments.html) provide a more flexible architecture for your application, compared to using activities alone, no matter what kind of activity you're building. As such, we suggest you use `PreferenceFragment` (/reference/android/preference/PreferenceFragment.html) to control the display of your settings instead of `PreferenceActivity` (/reference/android/preference/PreferenceActivity.html) when possible.

Your implementation of `PreferenceFragment` (/reference/android/preference/PreferenceFragment.html) can be as simple as defining the `onCreate()` (/reference/android/preference/PreferenceFragment.html#onCreate(android.os.Bundle)) method to load a preferences file with `addPreferencesFromResource()` (/reference/android/preference/PreferenceFragment.html#addPreferencesFromResource(int)). For example:

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

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

You can then add this fragment to an `Activity` (/reference/android/app/Activity.html) just as you would for any other `Fragment` (/reference/android/app/Fragment.html). For example:

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

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

Note: A PreferenceFragment (/reference/android/preference/PreferenceFragment.html) doesn't have a its own Context (/reference/android/content/Context.html) object. If you need a Context (/reference/android/content/Context.html) object, you can call getActivity() (/reference/android/app/Fragment.html#getActivity()). However, be careful to call getActivity() (/reference/android/app/Fragment.html#getActivity()) only when the fragment is attached to an activity. When the fragment is not yet attached, or was detached during the end of its lifecycle, getActivity() (/reference/android/app/Fragment.html#getActivity()) will return null.

Setting Default Values

The preferences you create probably define some important behaviors for your application, so it's necessary that you initialize the associated SharedPreferences (/reference/android/content/SharedPreferences.html) file with default values for each Preference (/reference/android/preference/Preference.html) when the user first opens your application.

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

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

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

Then, from the onCreate() (/reference/android/app/Activity.html#onCreate(android.os.Bundle)) method in your application's main activity—and in any other activity through which the user may enter your application for the first time—call setDefaultValues() (/reference/android/preference/PreferenceManager.html#setDefaultValues(android.content.Context, int, boolean)):

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

Calling this during onCreate() (/reference/android/app/Activity.html#onCreate(android.os.Bundle)) ensures that your application is properly initialized with default settings, which your application might need to read in order to determine some behaviors (such as whether to download data while on a cellular network).

This method takes three arguments:

- Your application `Context`.
- The resource ID for the preference XML file for which you want to set the default values.
- 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` (/reference/android/preference/PreferenceManager.html#KEY_HAS_SET_DEFAULT_VALUES) in the default value shared preferences file is false).

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

Using Preference Headers

In rare cases, you might want to design your settings such that the first screen displays only a list of subscreens (#Subscreens) (such as in the system Settings app, as shown in figures 4 and 5). When you're developing such a design for Android 3.0 and higher, you should use a new "headers" feature in Android 3.0, instead of building subscreens with nested `PreferenceScreen` (/reference/android/preference/PreferenceScreen.html) elements.

To build your settings with headers, you need to:

1. Separate each group of settings into separate instances of `PreferenceFragment`. That is, each group of settings needs a separate XML file.
2. Create an XML headers file that lists each settings group and declares which fragment contains the corresponding list of settings.
3. Extend the `PreferenceActivity` class to host your settings.
4. Implement the `onBuildHeaders()` callback to specify the headers file.

A great benefit to using this design is that `PreferenceActivity` (/reference/android/preference/PreferenceActivity.html) automatically presents the two-pane layout shown in figure 4 when running on large screens.

Even if your application supports versions of Android older than 3.0, you can build your application to use `PreferenceFragment` (/reference/android/preference/PreferenceFragment.html) for a two-pane presentation on newer devices while still supporting a traditional multi-screen hierarchy on older devices (see the section about Supporting older versions with preference headers (#BackCompatHeaders)).

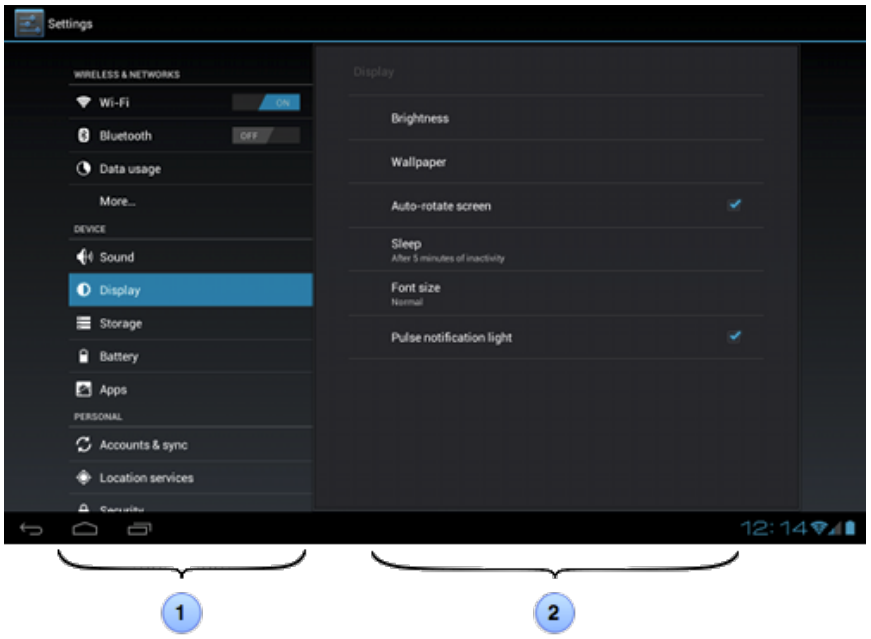

Figure 4. Two-pane layout with headers.
1. The headers are defined with an XML headers file.
2. Each group of settings is defined by a `PreferenceFragment` (/reference/android/preference/PreferenceFragment.html) that's specified by a `<header>` element in the headers file.

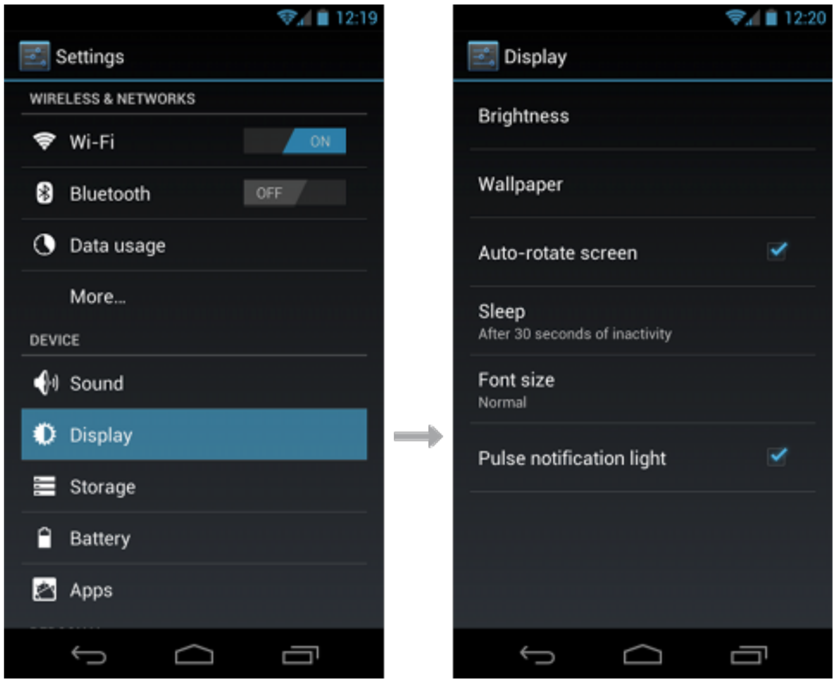

Figure 5. A handset device with setting headers. When an item is selected, the associated `PreferenceFragment` (/reference/android/preference/PreferenceFragment.html) replaces the headers.

Creating the headers file

Each group of settings in your list of headers is specified by a single `<header>` element inside a root `<preference-headers>` element. For example:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android'
    <header
        android:fragment="com.example.prefs.SettingsActivity$SettingsFragment(
        android:title="@string/prefs_category_one"
        android:summary="@string/prefs_summ_category_one" />
    <header
        android:fragment="com.example.prefs.SettingsActivity$SettingsFragment
        android:title="@string/prefs_category_two"
        android:summary="@string/prefs_summ_category_two" >
        <!-- key/value pairs can be included as arguments for the fragment. -
        <extra android:name="someKey" android:value="someHeaderValue" />
    </header>
</preference-headers>
```

With the `android:fragment` attribute, each header declares an instance of `PreferenceFragment` (/reference/android/preference/PreferenceFragment.html) that should open when the user selects the header.

The <`extras`> element allows you to pass key-value pairs to the fragment in a `Bundle` (/reference/android/os/Bundle.html). The fragment can retrieve the arguments by calling `getArguments()` (/reference/android/app/Fragment.html#getArguments()). You might pass arguments to the fragment for a variety of reasons, but one good reason is to reuse the same subclass of `PreferenceFragment` (/reference/android/preference/PreferenceFragment.html) for each group and use the argument to specify which preferences XML file the fragment should load.

For example, here's a fragment that can be reused for multiple settings groups, when each header defines an <`extra`> argument with the `"settings"` key:

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

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

Displaying the headers

To display the preference headers, you must implement the `onBuildHeaders()` (/reference/android/preference/PreferenceActivity.html#onBuildHeaders(java.util.List<android.preference.PreferenceActivity.Header>)) callback method and call `loadHeadersFromResource()` (/reference/android/preference/PreferenceActivity.html#loadHeadersFromResource(int, java.util.List<android.preference.PreferenceActivity.Header>)). For example:

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

When the user selects an item from the list of headers, the system opens the associated PreferenceFragment (/reference/android/preference/PreferenceFragment.html).

> **Note**: When using preference headers, your subclass of PreferenceActivity (/reference/android/preference/PreferenceActivity.html) doesn't need to implement the onCreate() (/reference/android/preference/PreferenceActivity.html#onCreate(android.os.Bundle)) method, because the only required task for the activity is to load the headers.

Supporting older versions with preference headers

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

Instead of opening a new PreferenceScreen (/reference/android/preference/PreferenceScreen.html), however, each of the <Preference> (/reference/android/preference/Preference.html) elements sends an Intent (/reference/android/content/Intent.html) to the PreferenceActivity (/reference/android/preference/PreferenceActivity.html) that specifies which preference XML file to load.

For example, here's an XML file for preference headers that is used on Android 3.0 and higher (res/xml/preference_headers.xml):

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

And here is a preference file that provides the same headers for versions older than Android 3.0 (res/xml/preference_headers_legacy.xml):

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

Because support for `<preference-headers>` was added in Android 3.0, the system calls `onBuildHeaders()` (/reference/android/preference/PreferenceActivity.html#onBuildHeaders(java.util.List<android.preference.PreferenceActivity.Header>)) in your `PreferenceActivity` (/reference/android/preference/PreferenceActivity.html) only when running on Androd 3.0 or higher. In order to load the "legacy" headers file (`preference_headers_legacy.xml`), you must check the Android version and, if the version is older than Android 3.0 (`HONEYCOMB` (/reference/android/os/Build.VERSION_CODES.html#HONEYCOMB)), call `addPreferencesFromResource()` (/reference/android/preference/PreferenceActivity.html#addPreferencesFromResource(int)) to load the legacy header file. For example:

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

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

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

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

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

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

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

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

Beware that consecutive calls to `addPreferencesFromResource()` (/reference/android/preference/PreferenceActivity.html#addPreferencesFromResource(int)) will stack all the preferences in a single list, so be sure that it's only called once by chaining the conditions with else-if statements.

Reading Preferences

By default, all your app's preferences are saved to a file that's accessible from anywhere within your application by calling the static method `PreferenceManager.getDefaultSharedPreferences()` (/reference/android/preference/PreferenceManager.html#getDefaultSharedPreferences(android.content.Context)). This returns the `SharedPreferences` (/reference/android/content/SharedPreferences.html) object containing all the key-value pairs that are associated with the `Preference` (/reference/android/preference/Preference.html) objects used in your `PreferenceActivity` (/reference/android/preference/PreferenceActivity.html).

For example, here's how you can read one of the preference values from any other activity in your application:

```
SharedPreferences sharedPref = PreferenceManager.getDefaultSharedPreferences(t
String syncConnPref = sharedPref.getString(SettingsActivity.KEY_PREF_SYNC_CONN
```

Listening for preference changes

There are several reasons you might want to be notified as soon as the use 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` (/reference/android/content/SharedPreferences.OnSharedPreferenceChangeListener.html) interface and register the listener for the `SharedPreferences` (/reference/android/content/SharedPreferences.html) object by calling `registerOnSharedPreferenceChangeListener()` (/reference/android/content/SharedPreferences.html#registerOnSharedPreferenceChangeListener(android.content.SharedPreferences.OnSharedPreferenceChangeListener)).

The interface has only one callback method, `onSharedPreferenceChanged()` (/reference/android/content/SharedPreferences.OnSharedPreferenceChangeListener.html#onSharedPreferenceChanged(android.content.SharedPreferences, java.lang.String)), and you might find it easiest to implement the interface as a part of your activity. For example:

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

    public void onSharedPreferenceChanged(SharedPreferences sharedPreferences
        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()` (/reference/android/preference/PreferenceActivity.html#findPreference(java.lang.CharSequence)) to get the `Preference` (/reference/android/preference/Preference.html) object that was changed so it can modify the item's summary to be a description of the user's selection. That is, when the setting is a `ListPreference` (/reference/android/preference/ListPreference.html) or other multiple choice setting, you should call `setSummary()` (/reference/android/preference/Preference.html#setSummary(int)) when the setting changes to display the current status (such as the Sleep setting shown in figure 5).

> **Note**: As described in the Android Design document about Settings (/design/patterns/settings.html), we recommend that you update the summary for a `ListPreference` (/reference/android/preference/ListPreference.html) each time the user changes the preference in order to describe the current setting.

For proper lifecycle management in the activity, we recommend that you register and unregister your `SharedPreferences.OnSharedPreferenceChangeListener` (/reference/android/content/SharedPreferences.OnSharedPreferenceChangeListener.html) during the `onResume()` (/reference/android/app/Activity.html#onResume()) and `onPause()` (/reference/android/app/Activity.html#onPause()) callbacks, respectively:

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

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

Managing Network Usage

Beginning with Android 4.0, the system's Settings application allows users to see how much network data their applications are using while in the foreground and background. Users can then disable the use of background data for individual apps. In order to avoid users disabling your app's access to data from the background, you should use the data connection efficiently and allow users to refine your app's data usage through your application settings.

For example, you might allow the user to control how often your app syncs data, whether your app performs uploads/downloads only when on Wi-Fi, whether your app uses data while roaming, etc. With these controls available to them, users are much less likely to disable your app's access to data when they approach the limits they set in the system Settings, because they can instead precisely control how much data your app uses.

Once you've added the necessary preferences in your `PreferenceActivity` (/reference/android/preference/PreferenceActivity.html) to control your app's data habits, you should add an intent filter for `ACTION_MANAGE_NETWORK_USAGE` (/reference/android/content/Intent.html#ACTION_MANAGE_NETWORK_USAGE) in your manifest file. For example:

```
<activity android:name="SettingsActivity" ... >
    <intent-filter>
       <action android:name="android.intent.action.MANAGE_NETWORK_USAGE" />
       <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

This intent filter indicates to the system that this is the activity that controls your application's data usage. Thus, when the user inspects how much data your app is using from the system's Settings app, a *View application settings* button is available that launches your `PreferenceActivity` (/reference/android/preference/PreferenceActivity.html) so the user can refine how much data your app uses.

Building a Custom Preference

The Android framework includes a variety of `Preference` (/reference/android/preference/Preference.html) subclasses that allow you to build a UI for several different types of settings. However, you might discover a setting you need for which there's no built-in solution, such as a number picker or date picker. In such a case, you'll need to create a custom preference by extending the `Preference` (/reference/android/preference/Preference.html) class or one of the other subclasses.

When you extend the `Preference` (/reference/android/preference/Preference.html) class, there are a few important things you need to do:

- Specify the user interface that appears when the user selects the settings.
- Save the setting's value when appropriate.
- Initialize the `Preference` with the current (or default) value when it comes into view.
- Provide the default value when requested by the system.
- If the `Preference` provides its own UI (such as a dialog), save and restore the state to handle lifecycle changes (such as when the user rotates the screen).

The following sections describe how to accomplish each of these tasks.

Specifying the user interface

If you directly extend the `Preference` (/reference/android/preference/Preference.html) class, you need to implement `onClick()` (/reference/android/preference/Preference.html#onClick()) to define the action that occurs when the user selects the item. However, most custom settings extend `DialogPreference` (/reference/android/preference/DialogPreference.html) to show a dialog, which simplifies the procedure. When you extend `DialogPreference` (/reference/android/preference/DialogPreference.html), you must call `setDialogLayoutResource()` (/reference/android/preference/DialogPreference.html#setDialogLayoutResource(int)) during in the class constructor to specify the layout for the dialog.

For example, here's the constructor for a custom DialogPreference (/reference/android/preference/DialogPreference.html) that declares the layout and specifies the text for the default positive and negative dialog buttons:

```
public class NumberPickerPreference extends DialogPreference {
    public NumberPickerPreference(Context context, AttributeSet attrs) {
        super(context, attrs);

        setDialogLayoutResource(R.layout.numberpicker_dialog);
        setPositiveButtonText(android.R.string.ok);
        setNegativeButtonText(android.R.string.cancel);

        setDialogIcon(null);
    }
    ...
}
```

Saving the setting's value

You can save a value for the setting at any time by calling one of the Preference (/reference/android/preference/Preference.html) class's `persist*()` methods, such as persistInt() (/reference/android/preference/Preference.html#persistInt(int)) if the setting's value is an integer or persistBoolean() (/reference/android/preference/Preference.html#persistBoolean(boolean)) to save a boolean.

Note: Each Preference (/reference/android/preference/Preference.html) can save only one data type, so you must use the `persist*()` method appropriate for the data type used by your custom Preference (/reference/android/preference/Preference.html).

When you choose to persist the setting can depend on which Preference (/reference/android/preference/Preference.html) class you extend. If you extend DialogPreference (/reference/android/preference/DialogPreference.html), then you should persist the value only when the dialog closes due to a positive result (the user selects the "OK" button).

When a DialogPreference (/reference/android/preference/DialogPreference.html) closes, the system calls the onDialogClosed() (/reference/android/preference/DialogPreference.html#onDialogClosed(boolean)) method. The method includes a boolean argument that specifies whether the user result is "positive"—if the value is `true`, then the user selected the positive button and you should save the new value. For example:

```
@Override
protected void onDialogClosed(boolean positiveResult) {
    // When the user selects "OK", persist the new value
    if (positiveResult) {
        persistInt(mNewValue);
    }
}
```

In this example, `mNewValue` is a class member that holds the setting's current value. Calling persistInt() (/reference/android/preference/Preference.html#persistInt(int)) saves the value to the SharedPreferences (/reference/android/content/SharedPreferences.html) file (automatically using the key that's specified in the XML file for this Preference (/reference/android/preference/Preference.html)).

Initializing the current value

When the system adds your Preference (/reference/android/preference/Preference.html) to the screen, it calls onSetInitialValue() (/reference/android/preference/Preference.html#onSetInitialValue(boolean, java.lang.Object)) to notify you whether the setting has a persisted value. If there is no persisted value, this call provides you the default value.

The onSetInitialValue() (/reference/android/preference/Preference.html#onSetInitialValue(boolean, java.lang.Object)) method passes a boolean, restorePersistedValue, to indicate whether a value has already been persisted for the setting. If it is true, then you should retrieve the persisted value by calling one of the Preference (/reference/android/preference/Preference.html) class's getPersisted*() methods, such as getPersistedInt() (/reference/android/preference/Preference.html#getPersistedInt(int)) for an integer value. You'll usually want to retrieve the persisted value so you can properly update the UI to reflect the previously saved value.

If restorePersistedValue is false, then you should use the default value that is passed in the second argument.

```
@Override
protected void onSetInitialValue(boolean restorePersistedValue, Object defaul
    if (restorePersistedValue) {
        // Restore existing state
        mCurrentValue = this.getPersistedInt(DEFAULT_VALUE);
    } else {
        // Set default state from the XML attribute
        mCurrentValue = (Integer) defaultValue;
        persistInt(mCurrentValue);
    }
}
```

Each getPersisted*() method takes an argument that specifies the default value to use in case there is actually no persisted value or the key does not exist. In the example above, a local constant is used to specify the default value in case getPersistedInt() (/reference/android/preference/Preference.html#getPersistedInt(int)) can't return a persisted value.

Caution: You **cannot** use the defaultValue as the default value in the getPersisted*() method, because its value is always null when restorePersistedValue is true.

Providing a default value

If the instance of your Preference (/reference/android/preference/Preference.html) class specifies a default value (with the android:defaultValue attribute), then the system calls onGetDefaultValue() (/reference/android/preference/Preference.html#onGetDefaultValue(android.content.res.TypedArray, int)) when it instantiates the object in order to retrieve the value. You must implement this method in order for the system to save the default value in the SharedPreferences (/reference/android/content/SharedPreferences.html). For example:

```
@Override
protected Object onGetDefaultValue(TypedArray a, int index) {
    return a.getInteger(index, DEFAULT_VALUE);
}
```

The method arguments provide everything you need: the array of attributes and the index position of the `android:defaultValue`, which you must retrieve. The reason you must implement this method to extract the default value from the attribute is because you must specify a local default value for the attribute in case the value is undefined.

Saving and restoring the Preference's state

Just like a `View` (/reference/android/view/View.html) in a layout, your `Preference` (/reference/android/preference/Preference.html) subclass is responsible for saving and restoring its state in case the activity or fragment is restarted (such as when the user rotates the screen). To properly save and restore the state of your `Preference` (/reference/android/preference/Preference.html) class, you must implement the lifecycle callback methods `onSaveInstanceState()` (/reference/android/preference/Preference.html#onSaveInstanceState()) and `onRestoreInstanceState()` (/reference/android/preference/Preference.html#onRestoreInstanceState(android.os.Parcelable)).

The state of your `Preference` (/reference/android/preference/Preference.html) is defined by an object that implements the `Parcelable` (/reference/android/os/Parcelable.html) interface. The Android framework provides such an object for you as a starting point to define your state object: the `Preference.BaseSavedState` (/reference/android/preference/Preference.BaseSavedState.html) class.

To define how your `Preference` (/reference/android/preference/Preference.html) class saves its state, you should extend the `Preference.BaseSavedState` (/reference/android/preference/Preference.BaseSavedState.html) class. You need to override just a few methods and define the `CREATOR` (/reference/android/preference/Preference.BaseSavedState.html#CREATOR) object.

For most apps, you can copy the following implementation and simply change the lines that handle the `value` if your `Preference` (/reference/android/preference/Preference.html) subclass saves a data type other than an integer.

```
private static class SavedState extends BaseSavedState {
    // Member that holds the setting's value
    // Change this data type to match the type saved by your Preference
    int value;

    public SavedState(Parcelable superState) {
        super(superState);
    }

    public SavedState(Parcel source) {
        super(source);
        // Get the current preference's value
        value = source.readInt();  // Change this to read the appropriate data
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        super.writeToParcel(dest, flags);
        // Write the preference's value
        dest.writeInt(value);  // Change this to write the appropriate data ty
    }

    // Standard creator object using an instance of this class
    public static final Parcelable.Creator<SavedState> CREATOR =
            new Parcelable.Creator<SavedState>() {

        public SavedState createFromParcel(Parcel in) {
            return new SavedState(in);
        }

        public SavedState[] newArray(int size) {
            return new SavedState[size];
        }
    };
}
```

With the above implementation of `Preference.BaseSavedState` (/reference/android/preference/Preference.BaseSavedState.html) added to your app (usually as a subclass of your `Preference` (/reference/android/preference/Preference.html) subclass), you then need to implement the `onSaveInstanceState()` (/reference/android/preference/Preference.html#onSaveInstanceState()) and `onRestoreInstanceState()` (/reference/android/preference/Preference.html#onRestoreInstanceState(android.os.Parcelable)) methods for your `Preference` (/reference/android/preference/Preference.html) subclass.

For example:

```
@Override
protected Parcelable onSaveInstanceState() {
    final Parcelable superState = super.onSaveInstanceState();
    // Check whether this Preference is persistent (continually saved)
    if (isPersistent()) {
        // No need to save instance state since it's persistent, use superclas
        return superState;
    }

    // Create instance of custom BaseSavedState
    final SavedState myState = new SavedState(superState);
    // Set the state's value with the class member that holds current setting
    myState.value = mNewValue;
    return myState;
}

@Override
protected void onRestoreInstanceState(Parcelable state) {
    // Check whether we saved the state in onSaveInstanceState
    if (state == null || !state.getClass().equals(SavedState.class)) {
        // Didn't save the state, so call superclass
        super.onRestoreInstanceState(state);
        return;
    }

    // Cast state to custom BaseSavedState and pass to superclass
    SavedState myState = (SavedState) state;
    super.onRestoreInstanceState(myState.getSuperState());
    // Set this Preference's widget to reflect the restored state
    mNumberPicker.setValue(myState.value);
}
```