Mobile Application Development - Part 5

Persistence I - Preferences

In this section we will:

- Look at preferences and how to write a preferences XML file
- Look at how to access preferences from Java code
- Look at saving instance state

Persistence

To ***persist*** data means to preserve data across multiple executions of the application. For example the first time you run an app you might enter your username and password to log on to a web server. The next time you run the app, you'll probably want your login details to be remembered. So we need some method of ***persisting*** data between individual uses of the app. There are a number of different mechanisms we can use, including:

- Preferences
- Using the ***savedInstanceState*** Bundle
- File I/O
- On-device databases, e.g. SQLite
- Storing data on a web server

Today we will look at ***preferences*** and in the next topic we will start to look at ***file I/O***. We will look at web communication a bit later in the unit. On-board databases will be covered next year.

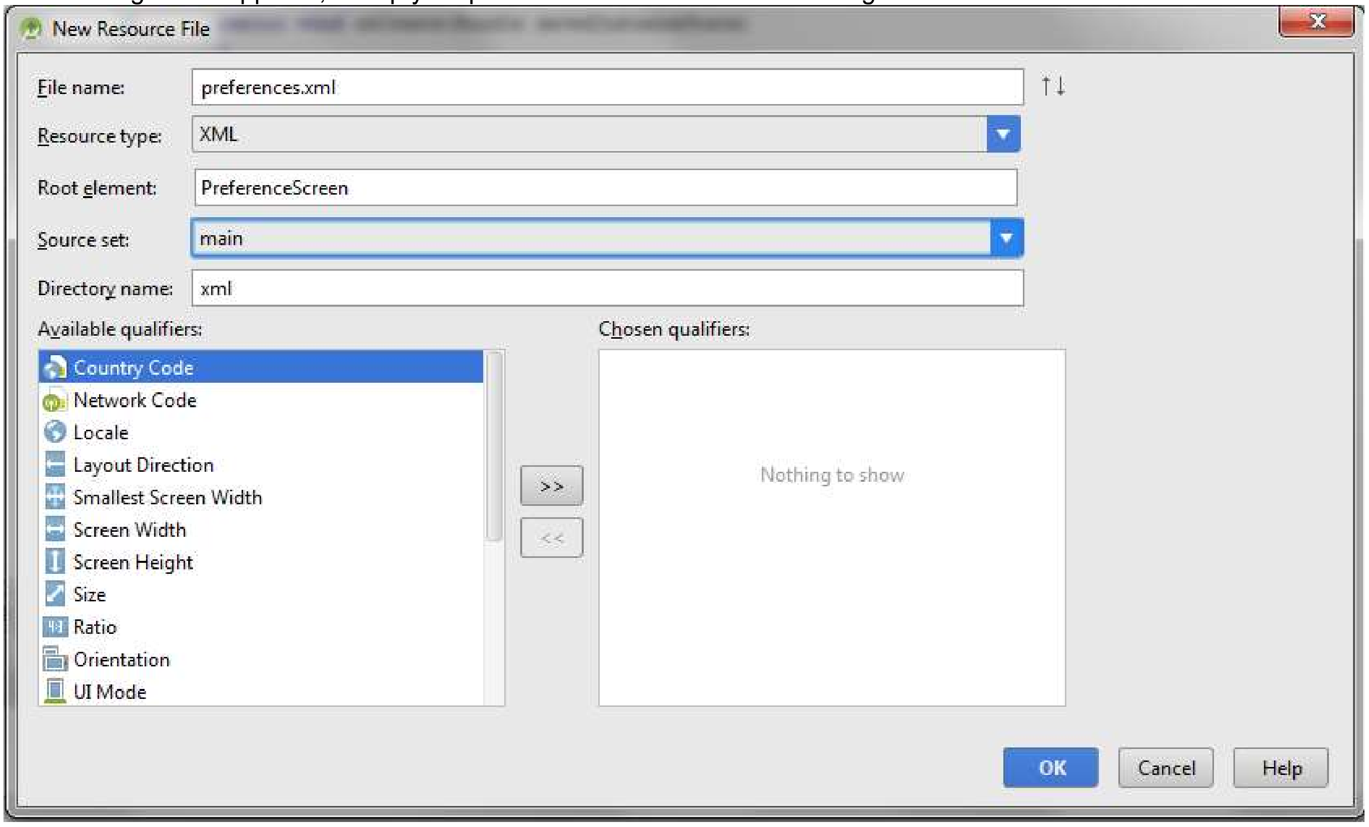
Preferences

Frequently a user might want to select ***preferences*** in an app. For example, they might want to select their preferred language or login details. Android makes this easy for us through the use of ***preferences***. Preferences are selected by the user in a ***PreferenceScreen*** which can be defined in an XML file much like layouts can. In Java code, you can then write a ***PreferenceActivity***: a special subclass of Activity designed specifically for displaying a PreferenceScreen.

Adding a preferences XML file to your project

- Open a new resource file as normal (highlight the ***res*** folder and choose ***File-New-Android Resource file).***

- In the dialog which appears, set up your preference file as shown in the diagram below:



New Resource File

File name: preferences.xml

Resource type: XML

Root element: PreferenceScreen

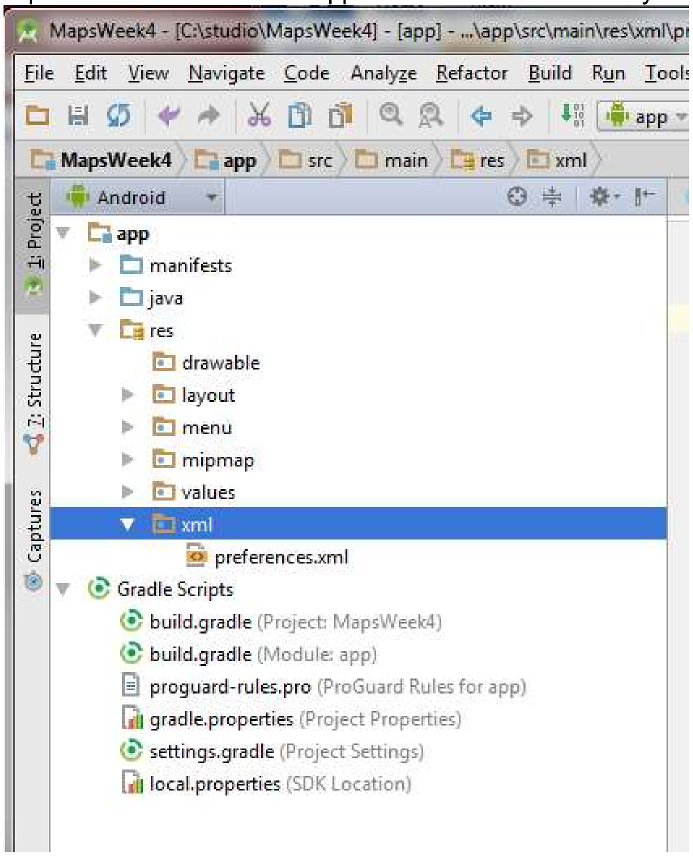
Source set: main

Directory name: xml

Available qualifiers: Country Code, Network Code, Locale, Layout Direction, Smallest Screen Width, Screen Width, Screen Height, Size, Ratio, Orientation, UI Mode

Chosen qualifiers: Nothing to show

- A preferences file will then appear in the ***xml*** section of your resources:



MapsWeek4 - [C:\studio\MapsWeek4] - [app] - ...\app\src\main\res\xml\pr

```
app
  manifests
  java
  res
    drawable
    layout
    menu
    mipmap
    values
    xml
      preferences.xml
Gradle Scripts
  build.gradle (Project: MapsWeek4)
  build.gradle (Module: app)
  proguard-rules.pro (ProGuard Rules for app)
  gradle.properties (Project Properties)
  settings.gradle (Project Settings)
  local.properties (SDK Location)
```

Defining a PreferenceScreen in XML

The first thing we will look at is how to define a set of preferences in XML. There are a number of types of preference, we will look at:

- EditTextPreference
- CheckboxPreference
- ListPreference

EditTextPreference

The ***EditTextPreference*** is one of the simplest types of preference. It allows the user to store a single value in a text field: fo
example, if you wanted to store the default location of a map (as latitude and longitude) you could use two EditTextPreferences, one for the latitude and one for the longitude. Here is an example of a preferences XML file with two EditTextPreferences:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
<EditTextPreference android:key="lat" android:defaultValue="50.9" android:title="Latitude" android:sum
<EditTextPreference android:key="lon" android:defaultValue="-1.4" android:title="Longitude" android:su
</PreferenceScreen>
```

You can hopefully see that each preference is defined by an <EditTextPreference> tag and that each preference has a number of attributes:

- The ***key***. Uniquely identifies a preference. We use this to access ***this particular preference*** in code.
- The ***defaultValue***. Default value of the preference; will appear within the edit text the first time a user accesses the preferences.
- The ***title***. Used to label the preference on the preference screen.
- The ***summary***. Longer text which appears under the title, to explain in more detail what the preference is for.

CheckBoxPreference

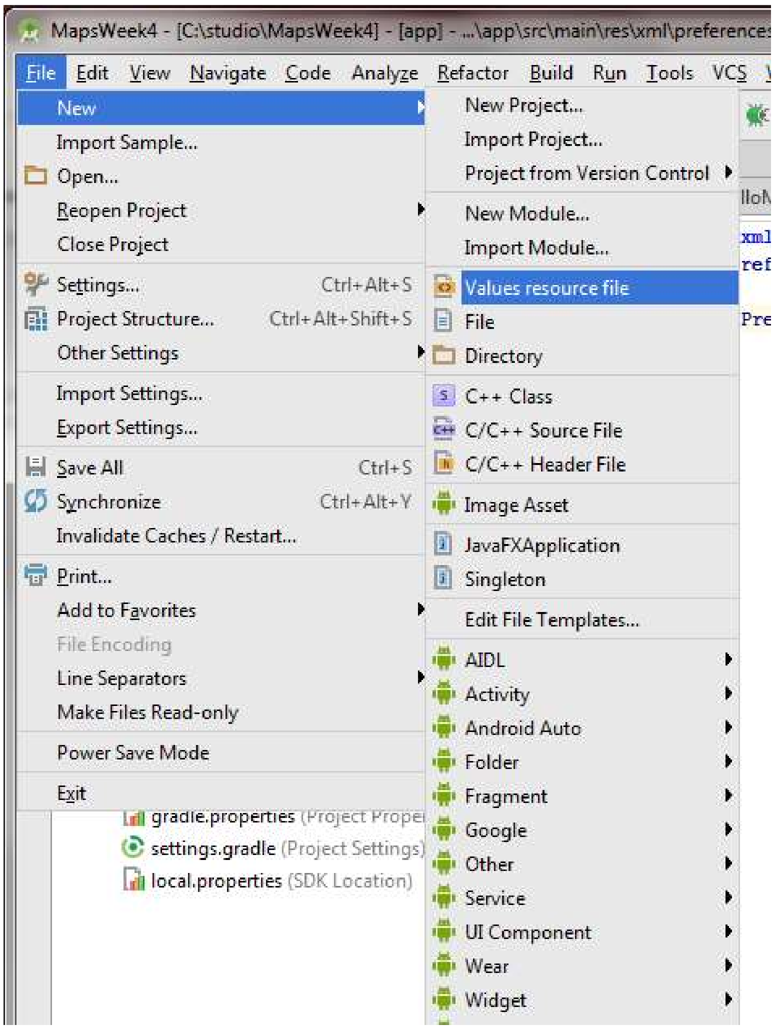
The ***CheckBoxPreference*** is also quite easy to understand. A CheckBoxPreference is a boolean value, which can be set to either true (if the checkbox is selected) or false (if it isn't). It is typically used for "on/off" values, for example in an app which accesses the internet to automatically download data about surrounding cinemas (their location, films showing, etc), the user might want to turn downloading on or off depending on whether they are connected to a wifi network. The XML is also quite simple:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
<CheckBoxPreference android:key="autodownload" android:defaultValue="true" android:title="Auto-downloa
android:summary="Auto-download data about surrounding cinemas?" />
</PreferenceScreen>
```

It can be seen that the XML is almost the same: the only differences are that it's a CheckBoxPreference and not an EditTextPreference, and that the default value is now a boolean meaning the preference will automatically be selected the first time the user runs the app.

ListPreference

The ***ListPreference*** is a bit more complex as it allows the user to pick from a list of values, typically presented as a set of radio buttons. The list of values has to be defined somewhere, and that place is the ***arrays*** resource file (array.xml). In a similar manner to strings.xml, array.xml is a convenient place to put all ***arrays*** that might be used in the UI. You can create the array.xml file by adding a ***values XML file*** to your project: highlight the ***values*** subfolder of ***res*** and then select ***File-New-Values resource file***:



A ListPreference contains an ***array*** of values so we store all the values in the array.xml file. Here is an example of an array.xml file which could be used for a ListPreference:

```
<resources>
    <string-array name="pizzaFlavours">
        <item>Cheese, Tomato and Mushroom</item>
        <item>Ham and Pineapple</item>
        <item>Hot n'Spicy</item>
        <item>Seafood</item>
    </string-array>
    <string-array name="pizzaFlavourCodes">
        <item>CTM</item>
        <item>HP</item>
        <item>HS</item>
        <item>SF</item>
    </string-array>
</resources>
```

Note how this arrays XML file actually defines ***two*** arrays, with the <string-array> tag. (string-array as they are arrays of strings).

- The ***pizzaFlavours*** array defines the text that will appear on the screen in the ListPreference;
- The ***pizzaFlavourCodes*** array defines the ***values*** that will be passed through to the application, i.e. the values that the Java code would actually see. Note how these are two- or three-letter codes. These are easier to process in our app than text descriptions, which are prone to typos when testing for in code (missing spaces, punctuation, etc).

If we now look at the preferences XML file for a ListPreference, note how this links to the arrays defined above:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
<ListPreference android:key="pizza" android:title="Pizza flavour" android:summary="Choose your pizza f
android:entries="@array/pizzaFlavours" android:entryValues="@array/pizzaFlavourCodes" />
```

```
</PreferenceScreen>
```

Note how we define the ListPreference. Many of the attributes (key, title, summary) are as before; however note ***android:entries*** and ***android:entryValues***. The former is a link to the array containing the entries that will appear on the screen (***@array/pizzaFlavours***) and the latter is a link to the actual values that will be carried through to the Java code (***@array/pizzaFlavourCodes***).

Note also how we refer to array XML resources in a similar way to strings; we use ***@array/arrayname*** to reference an array with a ***name*** of arrayname. Also note that in a real application, the ***android:title*** and ***android:summary*** would probably refer to entries in the strings.xml file, rather than hardcoded strings.

Defining a Preference Activity

As seen above, to actually show these preferences in a real app, we need to use a ***PreferenceActivity***. This is very simple t use, here is an example:

```
import android.preference.PreferenceActivity;
import android.os.Bundle;

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

All the work in setting up the preferences is done in the XML file so the activity code is extremely simple. We simply call the superclass version of ***onCreate()*** and then add the preferences from the external resource referenced by R.xml.***preferences*** this corresponds to an XML file called ***preferences***.xml.

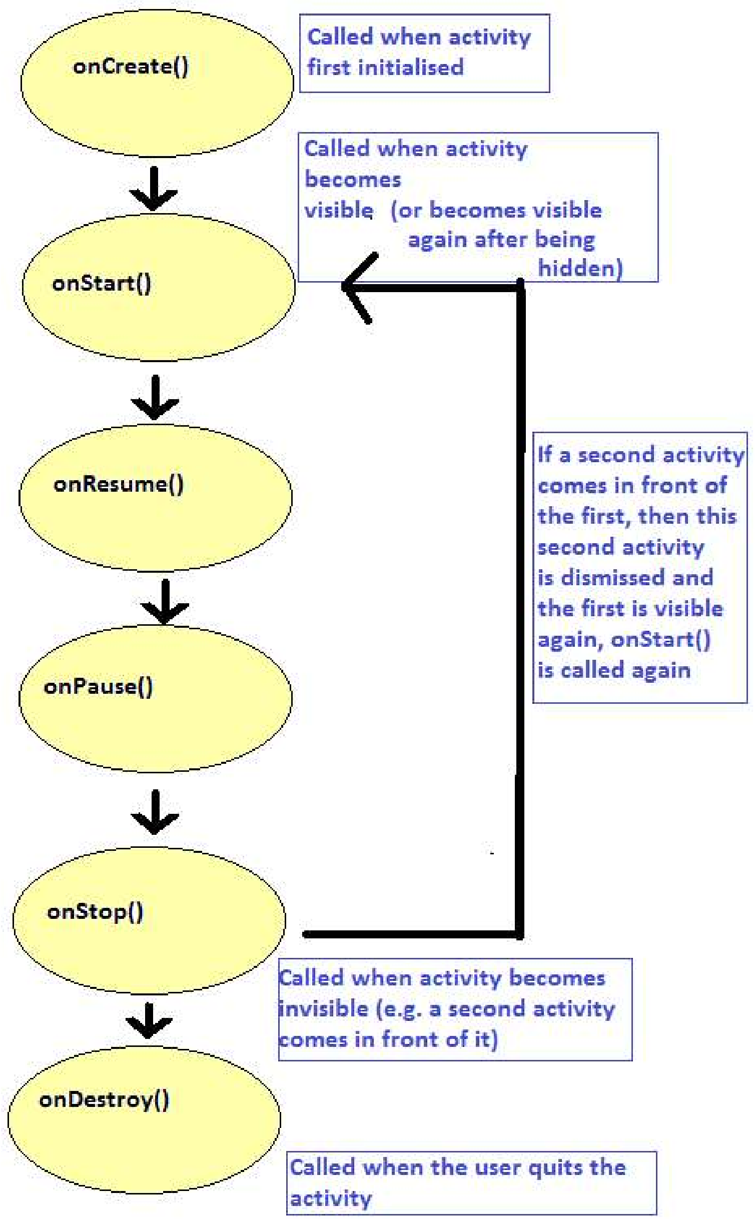
(Actually, though, the ***PreferenceActivity*** and the ***addPreferencesFromResource()*** method are ***deprecated*** (considered outdated) in Android 4.x. You will probably notice this when you try out the code, as Android Studio crosses out the ***addPreferencesFromResource()*** method. It is now recommended to use a preference ***fragment*** instead; however, we have not covered fragments yet and so we will use this code for now. When we come on to fragments, we will revisit preferences again and see how to set up a preference fragment.)

The Android Activity Lifecycle

To understand how to read preferences correctly you need to have some understanding of the Android activity lifecycle.

Mobile apps are not quite like standard desktop applications. In particular, ***the application may be interrupted***, most commonly by the user answering a phone call. Also, users commonly switch from one app to another. Unless the user quits the app using the "back" button, when this happens ***the original app is still runningin the background***.

To manage these possiblities, Android activities have a defined ***lifecycle***. The idea is to code activities to respond to the different points in the lifecycle. The lifecycle consists of a series of methods which run one after the other, as follows. When coding your app, you override whichever methods you want specific behaviour to occur at. With practically every activity, this includes ***onCreate()***, but your activity might need to override the others too.


onCreate()
Called when activity first initialised
onStart()
Called when activity becomes visible (or becomes visible again after being hidden)
onResume()
If a second activity comes in front of the first, then this second activity is dismissed and the first is visible again, onStart() is called again
onPause()
onStop()
Called when activity becomes invisible (e.g. a second activity comes in front of it)
onDestroy()
Called when the user quits the activity

It is described in full on the Android website, here.

The original article describes the lifecycle in full but to summarise, the following methods run when certain events occur:

- ***onCreate()***: runs when the application is first launched. Note also that certain actions, such as changing the device orientation, can force ***onCreate()*** to run.
- ***onStart()***: runs when the activity becomes visible. This can either be when it first becomes visible, or becomes visible again after being hidden. For example, it will be called after you return to the current Activity after using another application, or after closing a second activity within the same application, e.g. a preferences screen, causing the origina activity to become visible again.
- ***onStop()***: runs when the activity is hidden, e.g. you open another application which hides the current app, or you run a second activity within the same application, e.g. a preferences screen, which hides the current activity.
- ***onDestroy()***: runs when the user dismisses the application. This will happen when the user presses the Back button from the main activity. It can also be called under other circumstances, e.g. Android kills the activity due to low memory, or when the user rotates the device.
- ***onPause()***: when an activity not occupying the whole of the screen comes in front of the current activity. In this case, the current activity is not completely hidden, so ***onStop()*** is not called. ***However***, as shown on the diagram above, onPause() also runs immediately before onStop() when the activity becomes completely hidden.
- ***onResume()***: when the current activity becomes focused again after ***onPause()***.. In this case, the current activity was never completely hidden, so ***onStart()*** is not called. ***However***, as shown on the diagram above, onResume() also runs immediately after onStart() when the activity becomes visible after being completely hidden.

The lifecycle can have important consequences for development. For example, in a mapping app you might want to stop GPS communication when the activity becomes invisible and start it again when it becomes visible again (to save battery), in which case you would stop the GPS in ***onPause()*** and start it in ***onResume()***. If you did this in ***onDestroy()*** and ***onCreate()*** instead, the GPS would still be running if the activity was running but invisible, which for a mapping app would be unnecessary.

Accessing Preferences from the App

The other thing we need to look at is ***how to access the values of preferences from your Java code***. This can be accomplished by means of the ***PreferenceManager*** class, which controlls access to the preferences, and ***SharedPreferences*** which represents a set of preferences. Here is how you would read preferences. This code should be in your ***main activity***, not the preferences activity.

```
public void onResume()
{
    super.onResume();
    SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(getApplicationContext());
    double lat = Double.parseDouble ( prefs.getString("lat", "50.9") );
    double lon = Double.parseDouble ( prefs.getString("lon", "-1.4") );
    boolean autodownload = prefs.getBoolean("autodownload", true);
    String pizzaCode = prefs.getString("pizza", "NONE");

    // do something with the preference data...
}
```

- First of all, note how the code is in the ***onResume()*** method of your main activity. Why is this? Remember from the section above on the Android activity lifecycle that the ***onResume()*** method is called when the activity re-appears, having been hidden. When preferences are changed via a PreferenceActivity, the user will most likely press the Back button to dismiss the PreferenceActivity, resulting in the main activity appearing again and therefore its ***onResume()*** method being called - so ***onResume()*** is a good place to put code to read preferences.
- Note how we obtain a ***SharedPreferences*** object by using ***PreferenceManager.getDefaultSharedPreferences()***. Thi takes as a parameter a ***Context*** object. This object represents the context of the preferences; we can have preference which relate to the application as a whole or to an individual activity. Here we use ***getApplicationContext()*** to indicate that we want the preferences relating to the application as a whole, i.e. they can be accessed across all activities. Note that the Activity class is a subclass of ***Context***, as is Application, the class representing the application as a whole. Thi means that the return value of ***getApplicationContext()*** could be either an Activity or an Application. In the case of ***getApplicationContext()*** it is the Application. Many UI elements, such as dialog boxes, require a Context of some kind (usually, but not always, an Activity) as a parameter in the constructor, which represents the "parent" element of that UI element.
- Having got hold of the ***SharedPreferences***, we then use ***get*** methods to obtain an individual preference. These ***get*** methods use the preference's ***key*** to look up that particular preference. Notice also how the ***get*** methods take a seconc parameter, which is the default value to use if that preference does not exist. Note also how the ***EditTextPreference***s and the ***ListPreference*** return Strings, while the ***CheckBoxPreference*** returns a boolean.

Preferences outside a Preference Screen

It is possible for an app to have ***additional*** preferences, outside those defined in the XML for the PreferenceScreen. One example might be an app which is recording a trail of your position on the Earth using GPS; you might want to use this to record a walking or cycling route, for example. If the app is still recording when the app is closed down, you might want to save the recording status so that the next time the app starts up (which could be after powering the phone down and switching it back on again), the app can read the recording status from the preferences so that it continues recording if it was recording last time it was used. However, unlike preferences in a PreferenceScreen, these additional preferences will not automatically be saved when the app is destroyed. You instead need to make use of a ***SharedPreferences.Editor*** to save the preferences. For example:

```
public void onDestroy()
{
    super.onDestroy();
    SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(getApplicationContext());
    SharedPreferences.Editor editor = prefs.edit();
    editor.putBoolean ("isRecording", isRecording);
    editor.commit();
}
```

Note how the code is in the ***onDestroy()*** method; remember that ***onDestroy()*** is called when the app is destroyed, e.g. pressing the "back" button from the main activity or powering the phone down. Note also how we:

- Obtain a SharedPreferences.Editor object using the ***edit()*** method of SharedPreferences;
- Use an appropriate ***put*** method to set the preference. The first parameter is the preference key (which we would use again when reading the preference back) and the second parameter is the variable we want to save in the preferences. Here, ***isRecording*** would probably be an attribute of the Activity which represents whether we are recording a track.
- Finally we ***commit()*** the changes to our preferences.

With these sorts of additional preferences not controlled via a preference screen, we'd probably want to read them back in ***onCreate()***, rather than ***onResume()***, as we only need to save the data in preferences in between executions of the activity. Remember from the Part 1 section on the activity lifecycle that the activity remains in memory when ***onPause()*** is called (e.g when another activity comes in front of the original activity) so we can just save application state in attributes of the Activity object. It is only when the Activity is actually destroyed that we need to save the activity's state in preferences.

Saving Instance State

(ref: Recreating an Activity on the Android site)

The technique above is necessary to save the activity's state where the user deliberately closes down the activity. However occasionally the system destroys your activity and restarts it again, for example if memory is low or if the user rotates the screen. In these cases, we can use the simpler method of ***saving instance state*** to save the activity's data. To do this you need to implement an ***onSaveInstanceState()*** method:

```
public void onSaveInstanceState (Bundle savedInstanceState)
{
    savedInstanceState.putBoolean("isRecording", isRecording);
}
```

Note how the method is automatically passed a ***Bundle*** object (a collection of indices and values); you use appropriate ***put*** methods to save data in this Bundle. You can then read it back in the ***onCreate()*** method; remember that ***onCreate()*** takes one parameter ***savedInstanceState***, of data type ***Bundle***. This is used to restore instance state.

```
public void onCreate (Bundle savedInstanceState)
{
    super.onCreate (savedInstanceState);
    if (savedInstanceState != null)
    {
        isRecording = savedInstanceState.getBoolean ("isRecording");
    }
}
```

Hopefully this code is self-explanatory: we read back the same boolean that we saved in the Bundle. Note though that we need to test for ***savedInstanceState*** being null; if the activity is started without any saved instance state (e.g. for the first time) the Bundle will be null and we will get a NullPointerException error if we try to use it.

Exercise

The idea of this exercise is to develop a version of your mapping app which uses preferences, rather than a custom activity. Add an extra menu item to launch the preferences, and add a PreferenceActivity which includes the following:

1. Current Latitude and Longitude of the map (EditTextPreference; incorporate the code provided above)
2. Default zoom level for the map (EditTextPreference)

Add code in the ***onResume()*** to get these working. Once this is working, add the following further preference:

1. Map style - Regular or hike/bike map(use a ListPreference)

Make sure the preferences are read when the main activity is started, so that the app immediately changes when the PreferenceActivity is dismissed. Also save the selected mapping provider when the app is destroyed, so that when the user starts the app again, the same mapping provider will be used as last time.