CSCD 372 – Android Mobile Development

Module 9: Preferences and Files

Preferences API

- For storing user settings to the Android File System
  - the SharedPreferences class interfaces to the data file
  - data is stored as key-value pairs
  - the API handles most of the details of presenting app preferences (defined in an XML file) to the user, and letting them modify them
  - the Settings Activity Wizard was recently updated to generate non-deprecated code, but it is overly complicated – for now, I prefer to add Preferences from scratch
  - here I show an approach for adding Preferences to an existing project using a SettingsFragment
- SharedPreferences can also be used directly
  - you could use this to share data between apps, but such use has been deprecated due to security concerns
  - in particular, file creation modes other than MODE_PRIVATE have been deprecated
  - example of direct use within a single app follows ...

Using SharedPreferences

- **Direct use of SharedPreferences:**
  - contains get and set methods for boolean, float, int, long, and string and Set<String>
  - getAll() returns all key/value pairs in a Map object
  - contains() searches for the existence of a particular key
  - the file is located at: /data/data/com.../shared_prefs/

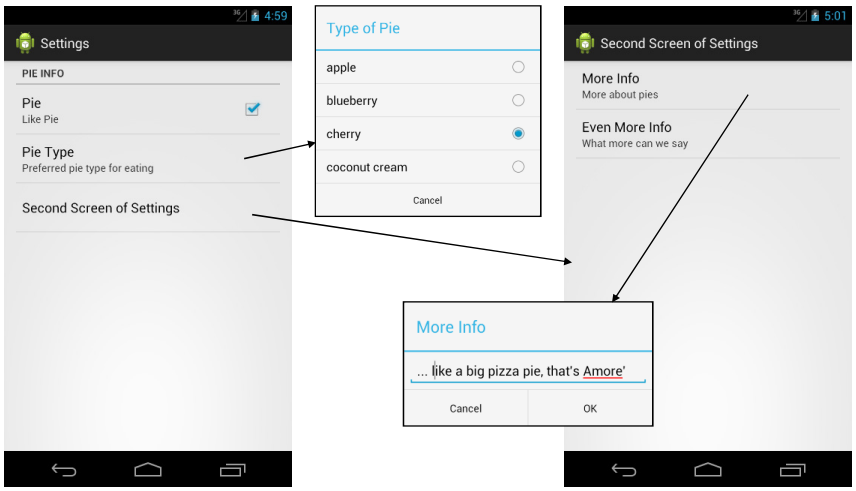
```
// filename and parameter keys:
private static final String SETTINGS = "SETTINGS" ;
public static final String FIRST_USE = "FIRST_USE" ;

SharedPreferences prefs = getSharedPreferences(SETTINGS,
                          Context.MODE_PRIVATE) ;
...
// default value if never been set is true:
boolean firstUse = prefs.getBoolean(FIRST_USE, true) ;
if (firstUse) {
   // say hello or do something special for the first run, then ...
   prefs.edit().putBoolean(FIRST_USE,false).commit() ;
}
```


Using the Preferences API

- **Possible approaches:**
  - Use PreferenceActivity: this is what the Settings Activity Wizard currently generates, but a lot of goo is generated
  - Use a generic Activity with a placeholder in its layout for a PreferenceFragment: this is what we cover here
  - Integrate a PreferenceFragment into your existing layout (optional lab)
- **PreferencesFragment Overview:**
  - Create a preferences.xml file in res/xml folder (New→Resource File, type = XML, Root element = PreferenceScreen)
  - Create a SettingsFragment class (Blank, no options), that implements OnSharedPreferenceChangeListener with overrides for: onCreate(), onResume(), onPause(), and (probably) onSharedPreferenceChanged()
  - Create an Empty Activity (e.g., SettingsActivity) with only a static <Fragment> placeholder in its associated layout file
  - Respond to the Main Activity Settings menu event by creating an Intent for SettingsActivity.class and starting the Activity
  - Android will then handle all interactions with the user

Preferences Example



preferences.xml Example

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory
        android:title="Pie Info">
      <CheckBoxPreference
          android:key="pie"
          android:title="Pie"
          android:summary="Like Pie"
          android:defaultValue="true" />
      <ListPreference
          android:dependency="pie"
          android:key="pie_type"
          android:title="Pie Type"
          android:summary="Preferred pie type %s"
          android:dialogTitle="Type of Pie"
          android:entries="@array/pie_array"
          android:entryValues="@array/pie_array"
          android:defaultValue="apple" />
    </PreferenceCategory>
    ...
```

Google recommends updating these to reflect the choice (we'll do that in the lab exercise)

For ListPreference only, %s will fill in the current value, as long as there is one (which is a good reason to specify a default) Note that says the current *Value*, which comes from entryValues

preferences.xml Example

```
...
<PreferenceScreen
       android:key="second_preferencescreen"
       android:title="Second Screen of Settings">
   <EditTextPreference
       android:key="more_info"
       android:title="More Info"
       android:summary="More about pies"
       android:defaultValue="" />
   <EditTextPreference
       android:key="even_more_info"
       android:title="Even More Info"
       android:summary="What more can we say"
       android:defaultValue="" />
   </PreferenceScreen>
</PreferenceScreen>
```

- All standard preferences only manage string values
  - although you can put numbers in programmatically with setXXX, for standard preferences managed by the API, you'll have to convert from strings when retrieving the value – this restriction doesn't apply to custom preferences
  - but you can restrict the user input to numeric entry using:

```
android:inputType="number" (or "numberSigned","numberDecimal", ...)
android:numeric="integer" (or "signed" or "decimal")
```

PreferenceFragment Example

```
public class SettingsFragment extends PreferenceFragment
             implements OnSharedPreferenceChangeListener
{
   @Override
   public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState) ;
        // load the XML definition of our preferences
        addPreferencesFromResource(R.xml.preferences) ;
   }
   ...
```

- At this point we have enough for Android to launch and manage our preferences
  - except we still need to respond to the Settings menu event by launching an activity that uses this Fragment (or loads the Fragment into a placeholder)
  - and we'll need to add to the onCreate method above if we want the summaries (or titles) to reflect the current values (shortly)
- We'll also need to read the preferences file at some point
  - you likely want to do this on your MainActivity (or MainFragment) onResume(): see slide 3, but use PreferenceManager.getDefaultSharedPreferences()
  - we may also want to know as soon as any preference is changed...

Registering a Change Listener

- **OnSharedPreferenceChangeListener**
  - (probably in your Settings Fragment class)
  - this will be called when the user *changes* a preference
  - we must register a listener, and google recommends registering and unregistering in onResume() and onPause()

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

Change Listener

- **Typically we use this** to update the preference summary
  - and perhaps update some other feature of the app
- **If you don't care about summaries, and only want to process changes after the Preference Activity ends**
  - then you won't need this change listener
  - but wait (you ask) how does my main activity (or main fragment) know the settings have changed? It can simply call PreferenceManager. getDefaultSharedPreferences() in onResume()

```
@Override
public void onSharedPreferenceChanged(SharedPreferences
                              sharedPreferences, String key) {
   Preference pref = findPreference(key) ;
   switch(key) {
      // this one is custom and supports ints
      case "numcoils":
         pref.setSummary(Integer.
              toString(sharedPreferences.getInt(key, 11)));
         // and maybe do some more stuff
         break ;
      default:
         pref.setSummary(sharedPreferences.getString(key,""));
   }
}
```

Initializing the Summaries

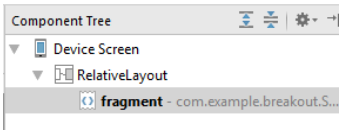
- If we want the summaries (or titles) to show the current values ...
  - the preceding change listener only updates them after the user changes them
  - but we probably also want them to display the latest values when the preference screen is launched (beats me why that isn't automatic)
  - maybe cheat and call the change listeners yourself from the Fragment's onCreate() ...

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState) ;
   addPreferencesFromResource(R.xml.preferences);

   SharedPreferences prefs =
            getPreferenceScreen().getSharedPreferences() ;
   onSharedPreferenceChanged(prefs, "numcoils") ;
   onSharedPreferenceChanged(prefs, "spring") ;
   onSharedPreferenceChanged(prefs, "shape") ;
}
```


Showing the PreferenceFragment

- Here we embed the PreferenceFragment in another Activity
  - in the lab we may do a replace transaction on a fragment placeholder instead
- Start with New→Activity→Empty Activity
  - put just a <fragment> widget on the layout
  - and select your SettingsFragment class from the list
  - the class needs only an onCreate method
  - (you can delete any generated menu_settings.xml)




```
public class SettingsActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState) ;
      setContentView(R.layout.activity_settings) ;
   }
}
```

Launching the Preference Activity

- We launch that activity in response to the Settings menu selection in the Main Activity:

```
@Override
public boolean onOptionsItemSelected(MenuItem item)
{
   switch (item.getItemId()) {
      case R.id.action_settings:
         Intent intent = new Intent(this, SettingsActivity.class) ;
         startActivity(intent) ;
         return true ;
      default:
         return super.onOptionsItemSelected(item) ;
   }
}
```

- See PreferenceEx, which combines this with a manually managed SharedPreferences file in order to detect the first run of the app

Static File Input

- Any file can be placed in the assets directory
  - the contents are encoded into the APK; they are NOT expanded into separate files in the Android File System
    - as a consequence you can only open them for reading
    - cannot write to them at run-time (but you can copy them to a local file)
  - to create an asset directory from the Project/Android view:
    - right-click on app, then: New→Folder→Assets Folder
  - The first step in accessing such files is to obtain a reference to the AssetManager (getAssets)
    - as we saw in an earlier module, you can then get an AssetFileDescriptor (openFd method), to which you can attach a FileReader
    - or you can simply open an InputStream (open method)
- A static file can also be placed in the res/raw directory
  - to create a raw directory from the Project/Android view:
    - right-click on res, then: New→Directory (and call it "raw")
  - to open these, call openRawResource(), passing the R.raw.<filename> resource ID (lower case filenames only)
  - this returns an InputStream that you can use to read the file

Internal Storage File I/O

- **Internal storage is private to your app**
  - **it cannot be accessed by other apps, or the user (unless the phone is rooted)**
  - **these files are removed if the app is de-installed**
- **To open a file for output**
  - **call openFileOutput() with a filename and a mode:**
    - MODE_PRIVATE will create or replace the file
    - MODE_APPEND will append to the end of the file
    - MODE_WORLD_READABLE and MODE_WORLD_WRITEABLE have been deprecated
  - **returns a FileOutputStream**
    - write to the file with write(), close the file with close()
- **To open a file for input**
  - **call openFileInput() with a filename**
  - **returns a FileInputStream**
    - see the available(), read(), skip(), and close() methods

Other Useful File Methods

- **These are all methods of Context**
  - **your Activity is a subclass of Context**
- **getFilesDir()**
  - **gets the absolute path to the filesystem directory where your internal files are saved**
- **getDir()**
  - **creates (or opens an existing) directory within your internal storage space**
- **deleteFile()**
  - **deletes a file saved on the internal storage**
- **fileList()**
  - **returns an array of files currently saved by your application**

External Storage

- External Storage may be a removable SD card
  - or a non-removable storage (that is accessible from a computer when the device is plugged in via USB)
- Permission is required (this is now a *dangerous permission)*:

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

  - or READ_EXTERNAL_STORAGE
  - if you need to both read and write, then you need request only WRITE_EXTERNAL_STORAGE (implies read access as well)
  - beginning with Android 4.4 (API 19), these permissions are *not* required if you're reading or writing only files that are private to your app...
- App-Specific (Private) External Storage
  - if you're handling files that are not intended for other apps, use a private storage directory (on the external storage) by calling getExternalFilesDir()

```
File file = new File(getExternalFilesDir(null), "MyFile.txt") ;
```

  - these files are not typically visible to the user as media files
  - these files WILL be deleted when the app is uninstalled
  - the storage may or may not be available (see getExternalStorageState – next slide)

Shared External Storage

- Shared External Storage (Mod10TextFileIO):
  - use: File path = Environment.*getExternalStorageDirectory*() ;
  - these files will NOT be deleted on uninstall
  - may or may not be visible to the user without root privileges
- Before accessing either private or shared external storage, check for availability:

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

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

Shared Media Files

- Consider saving your files to a standard media-type directory, so that the media scanner can properly categorize them:
  - call getExternalStoragePublicDirectory() to obtain one of those directories:
  - DIRECTORY_ALARMS, DIRECTORY_DCIM, DIRECTORY_DOCUMENTS (API 19), DIRECTORY_DOWNLOADS, DIRECTORY_MOVIES, DIRECTORY_MUSIC, DIRECTORY_NOTIFICATIONS, DIRECTORY_PICTURES, DIRECTORY_PODCASTS, DIRECTORY_RINGTONES
  - these are also visible to the user when connected via USB
- Example method that creates a directory for a new photo album in the public pictures directory:

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for public pictures
    File file = new
       File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName) ;
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Failed to create a directory") ;
    }
    return file ;
}
```

Additional Data Support

- Android provides full support for SQLite databases
  - any databases you create will be accessible by name to any class in the application, but not outside the application
  - we're not covering this because CSCD 327 is not a pre-req for this class, other than to relay this:
  - the recommended method to create a SQLite database is to create a subclass of SQLiteOpenHelper
    - then override the onCreate() method, in which you can execute a SQLite command to create tables in the database
  - call getReadableDatabase() or getWritableDatabase() to read or write to the database
    - both return a SQLiteDatabase object on which you can execute SQLite queries using the query() methods
- Android also includes support for parsing XML and Json
  - XmlPullParser (also Java SAX and DOM APIs)
  - JsonReader, JsonWriter