Embedded Systems

CS379 Embedded Systems

File Storage and Preferences

C.-Z. Yang
http://syslab.cse.yzu.edu.tw/~czyang

9.1 Storing Data

- Applications often store information about a user's preferences in order to provide a more sophisticated level of personalization and responsiveness
- Advance Android applications are frequently data-driven and can require the management of a larger and more complex volume of data.
- The Android platform allows data files to be saved on the device's internal memory and on an external storage media.

9.2 Shared Preferences

- The Android SDK provides SharedPreferences for the most primitive type of data storage.
  - A SharedPreferences file is stored internally and managed by the framework.
  - By default, all data stored in internal storage are private.
  - Files stored in external storage can be made accessible to other applications.

SharedPreferences

- The term *Shared Preferences* refers to the storage of a limited set of primitive data used to make persistent changes in an Android application.
- It is a simple way to read and write key-value pairs of data.
- A key-value pair is a data representation model that uses a set of key identifiers along with associated data values.
  - The model is frequently used in hash tables and configuration files.

Adding preferences support

1. Retrieve an instance of a SharedPreferences object.
2. Create a SharedPreferences.Editor to modify preference content.
3. Make changes to the preferences using the Editor.
4. Commit your changes.

- To get a SharedPreferences object for your application, use one of two methods:
  - getSharedPreferences() - Use this if you need multiple preferences files identified by name, which you specify with the first parameter.
  - getPreferences() - Use this if you need only one preferences file for your Activity. Because this will be the only preferences file for your Activity, you don't supply a name.
- To write values:
  - Call edit() to get a SharedPreferences.Editor.
  - Add values with methods such as putBoolean() and putString().
  - Commit the new values with commit()

- An example that saves a preference for silent keypress mode in a calculator:

```
public class Calc extends Activity {
   public static final String PREFS_NAME = "MyPrefsFile";

   @Override
   protected void onCreate(Bundle state){
     super.onCreate(state);
     . . .

     // Restore preferences
     SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
     boolean silent = settings.getBoolean("silentMode", false);
     setSilent(silent);
   }
```

Default argument

```
    @Override
    protected void onStop(){
       super.onStop();

      // We need an Editor object to make preference changes.
      // All objects are from android.context.Context
      SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
      SharedPreferences.Editor editor = settings.edit();
      editor.putBoolean("silentMode", mSilentMode);

      // Commit the edits!
      editor.commit();
    }
}
```

MODE_PRIVATE

Creating Private Preferences

- Individual activities can have their own private preferences.
  - These preferences are for the specific Activity only and are not shared with other activities within the application.
- Retrieves the activity’s private preferences

```
import android.content.SharedPreferences;
...
SharedPreferences settingsActivity = getPreferences(MODE_PRIVATE);
```

Creating Shared Preferences

- Creating shared preferences is similar.
  - Name the preference set
  - Use a different call to get the preference instance

```
import android.content.SharedPreferences;
...
SharedPreferences settings =
    getSharedPreferences("MyCustomSharedPreferences", 0);
```

- Once the key is used to identify which key-value pair is marked for removal, commit() can be called to perform the final changes.

```
editor.remove(KEYNAME1);
editor.remove(KEYNAME2);
editor.commit();
```

- All key-value data sets within the shared preferences file can be easily cleared using the clear() method.

```
editor.clear();
editor.commit();
```

9.3 File Storage

- In Android, a file-based storage option allows data to be written to an actual file structure
  - This storage method requires more control regarding read and write permissions
- Internal storage allows data to be stored directly onto the device's memory
  - This storage is always available, assuming there is space
- External storage may not always be obtainable on a device.

Internal and External Storage

- There are significant differences in how external and internal storage is utilized in an application
- Internal storage files can be configured to be readable and writeable by the application
  - Typically, internal storage is utilized when processing an image, video and audio elements, and large data files
  - By default, files saved to internal storage are private to the application

Internal and External Storage

- External storage is publicly shared storage, which means that it can be made available to external applications
  - Unlike internal storage, once an application is uninstalled, external storage files will continue to exist

FileOutputStream

- A file output stream is an output stream for writing data to a File or to a FileDescriptor.

```
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();
}
```

FileInputStream

- A FileInputStream obtains input bytes from a file in a file system.

```
FileInputStream in = null;
StringBuffer data = new StringBuffer();
try {
   in = openFileInput("test.txt");

   BufferedReader reader = new BufferedReader( new InputStreamReader(in, "utf-8"));
   String line;
   while ((line = reader.readLine()) != null) {
      data.append(line);
   }
} catch (Exception e) {
   ;
} finally {
   try {
      in.close();
   } catch (Exception e) {
      ;
   }
}
```

Obtain Permissions for External Storage

- To write to the external storage, you must request the WRITE_EXTERNAL_STORAGE permission in your manifest file:

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

Caution:

- Since API 19 (Android 4.4) you must explicitly specify the READ_EXTERNAL_STORAGE permission.
  - Before API level 19, this permission is not enforced and all apps still have access to read from external storage.
- To ensure that your app continues to work as expected, you should declare this permission now, before the change takes effect.

Save a File on External Storage

- Because the external storage may be unavailable, 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.

Availability of external storage

- For example

```
/* 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;
}
```

External Storage

- Two categories of files: public and private
- Public files
  - Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain available to the user.
  - For example, photos captured by your app or other downloaded files.
- Private files
  - Files that rightfully belong to your app and should be deleted when the user uninstalls your app.

Saving public files

- Use the getExternalStoragePublicDirectory() method to get a File representing the appropriate directory on the external storage.

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for the user's public pictures directory.
    File file = new File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

Saving private files

- you can acquire the appropriate directory by calling getExternalFilesDir() and passing it a name indicating the type of directory you'd like.

```
public File getAlbumStorageDir(Context context, String albumName) {
    // Get the directory for the app's private pictures directory.
    File file = new File(context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

Query Free 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.

Delete a File

- 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);
```

Concluding Remarks