Android Saving Files on Internal and External Storage

Android allows us to store files in its file system which is quite similar to any other Linux filesystem that you must have experience with. Using the `java.io` file input/output APIs we can start reading and writing files to the Android filesystem. This is super useful when you want a store files (but not relational data or some sort of key/value cache pairs) on the device. Files like audio, video, images, documents, etc. all makes sense to store in the file system when required.

All Android devices have two types of file storage options – internal and external. We'll discuss both of them in this article.

> What's the one thing every developer wants? More screens! Enhance your coding experience with an external monitor to increase screen real estate.

Internal Storage

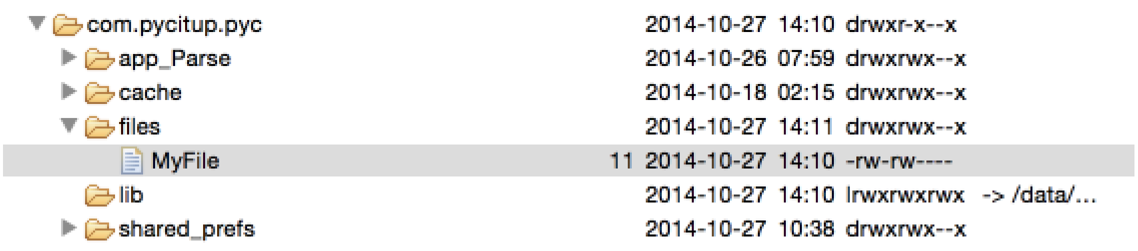
This area of storage is sort of private to the application. It is always available to the application and gets purged when the app is uninstalled by the user. Let's quickly see how to write a file to the internal storage:

```
// Create a file in the Internal Storage

String fileName = "MyFile";
String content = "hello world";

FileOutputStream outputStream = null;
try {
    outputStream = openFileOutput(fileName, Context.MODE_PRIV
    outputStream.write(content.getBytes());
    outputStream.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

This should instantly create a file called `MyFile` with the content `hello world`. The internal storage directory is usually found in a special location specified by our app's package name. In my case it is `/data/data/[package name]` and the files created are stored in a directory called `files` in that directory.



```
▼ com.pycitup.pyc                2014-10-27 14:10 drwxr-x--x
  ▶ app_Parse                    2014-10-26 07:59 drwxrwx--x
  ▶ cache                        2014-10-18 02:15 drwxrwx--x
  ▼ files                        2014-10-27 14:11 drwxrwx--x
      MyFile                  11 2014-10-27 14:10 -rw-rw----
    lib                          2014-10-27 14:10 lrwxrwxrwx  -> /data/...
  ▶ shared_prefs                 2014-10-27 10:38 drwxrwx--x
```

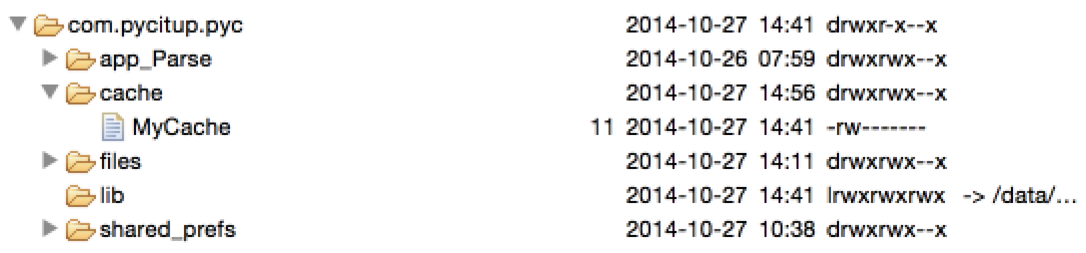
Note: The constant `Context.MODE_PRIVATE` makes the file inaccessible to other apps. If you want other apps to access your files then it might be better to save them on External storage which we'll cover in a bit.

In order to cache some data, i.e., create cache files rather than storing data in files persistently, we can open a `File` object representing the cache directory in the internal storage by using the `getCacheDir()` method. Let's see some code to create cache files.

```
String content = "hello world";
File file;
FileOutputStream outputStream;
try {
    // file = File.createTempFile("MyCache", null, getCacheDi
    file = new File(getCacheDir(), "MyCache");

    outputStream = new FileOutputStream(file);
    outputStream.write(content.getBytes());
    outputStream.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

This creates cache files in a directory called `cache`.



```
▼ com.pycitup.pyc                2014-10-27 14:41 drwxr-x--x
  ▶ app_Parse                    2014-10-26 07:59 drwxrwx--x
  ▼ cache                        2014-10-27 14:56 drwxrwx--x
      MyCache                 11 2014-10-27 14:41 -rw-------
  ▶ files                        2014-10-27 14:11 drwxrwx--x
    lib                          2014-10-27 14:41 lrwxrwxrwx  -> /data/...
  ▶ shared_prefs                 2014-10-27 10:38 drwxrwx--x
```

When the device is low on internal storage space, cache files get deleted to recover space. Although according to the documentation we shouldn't rely on the auto cleaning mechanism. Instead we should delete cache files ourselves and make sure that they do not consume more than 1MB.

If we use `getFilesDir()` instead of `getCacheDir()` then it'll store files in the `files` directory making the code similar in effect to the previous example. Also note the commented line that uses `File.createTempFile()` to create the file. You can use that version too if you want but keep in mind that the file generated will have random characters suffixed. This is done to maintain uniqueness in the name of the files by this method. So one way to keep track of it is to store the file names in a DB and then reference but for a simple task that can be an overkill.

If you quickly want to read your files then here's the code to do that exact job:

```
BufferedReader input = null;
File file = null;
try {
    file = new File(getCacheDir(), "MyCache"); // Pass getFil

    input = new BufferedReader(new InputStreamReader(new File
    String line;
    StringBuffer buffer = new StringBuffer();
    while ((line = input.readLine()) != null) {
        buffer.append(line);
    }

    Log.d(TAG, buffer.toString());
} catch (IOException e) {
    e.printStackTrace();
}
```

External Storage

Android devices supports another type of storage called external storage where apps can save files. It can be either removable like an SD card or non-removable in which case it is internal. Files in this storage are world readable which means other applications have access to them and the user can transfer them to their computer by connecting with a USB. Before writing to this volume we must check that it is available as it can become unavailable if the SD card is removed or mounted to the user's computer.

Using `getExternalStorageState()` we can get the current state of the primary external storage device. If it's equal to `Environment.MEDIA_MOUNTED` then we'll have read/write access and if equal to `Environment.MEDIA_MOUNTED_READ_ONLY` then we have only read access.

```
Environment.getExternalStorageState().equals(Environment.MEDIA
```

Although external storage is accessible and modifiable by the user and other apps, you can save the files in two ways – public and private. Before we discuss these two types note that in order to read and/or write files to the external storage we'll need the `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` system permissions. The latter covers both read and write. Although kicking off with Android 4.4 Kitkat (API 19), reading and writing private files in the private directories of the external store do not require permissions in AndroidManifest.xml.

Public files

Files that'll remain on the storage even after the application is uninstalled by the user like media (photos, videos, etc.) or other downloaded files. There are 2 methods that we can use to get the public external storage directory for placing files:

- `getExternalStorageDirectory()` – This returns the primary (top-level or root) external storage directory.
- `getExternalStoragePublicDirectorty()` – This returns a top level public external storage directory for shoving files of a particular type based on the argument passed. So basically the external storage has directories like Music, Podcasts, Pictures, etc. whose paths can be determined and returned via this function by passing the appropriate environment constants.

Let’s quickly save a file in the external storage directory:

```
String content = "hello world";
File file;
FileOutputStream outputStream;
try {
    file = new File(Environment.getExternalStorageDirectory()

    outputStream = new FileOutputStream(file);
    outputStream.write(content.getBytes());
    outputStream.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

As simple as that! You can try `Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS)` instead of `Environment.getExternalStorageDirectory()` to save the file in the `Downloads` directory of your external storage.

This SO answer has some extra details regarding these methods worth reading and knowing.

Private files

These should be files belonging to your app that’ll get deleted once the user uninstalls the app. These files will be accessible to other apps but are such that don’t provide any value to other apps or even to the user outside the context of the app like certain media (audio, images, etc.) files downloaded by a game.

Saving files to the directories appropriate for holding your private files is super easy. You can try the previous examples but instead use `getExternalFilesDir()` to get the appropriate directory path. Again Environment constants like `DIRECTORY_MUSIC` or `DIRECTORY_PICTURES` can be passed or you can also pass `null` to return the root directory for your app’s private directory on the volume. The path should be `Android/data/[package name]/files/` on the external storage. You can check the new files created in Android File Transfer, `adb shell` or ES file explorer.

Similar to `getCacheDir()` in terms of internal storage, we also have `getExternalCacheDir()` to save cache files in our private external store.

Multiple External Storages

There are devices that allocates a portion of its internal memory to use as external storage as well as offers an SD card slot that can also act as an external storage. Which one is primary or secondary is decided by the device manufacturer. In such cases to store private files you can get a list of all the directory paths using the `getExternalFilesDirs()` method but as for the public files the documentation lacks information currently which means it might not be possible at all. `Environment.getExternalStorageDirectory()` returns the "primary" external storage. Due to this information being devoid, an issue has already been created.

There are ways to get around the limitation. In order to learn that as well as gather a lot more information around this behaviour just google something like "android get access to secondary storage" or "android multiple storage". I found this interesting link that suggests Google trying to make SD card sort of useless (or atleast hard to work with) with the KitKat (4.4) and above.

Bonus

Deleting Files

Removing files created on the internal or external storage is super easy. Just call the `delete()` method on the `File` object.

```
file.delete();
```

Also the `Context` class provides us with the `deleteFile()` method that we can use to locate and delete files in our internal storage directory.

```
deleteFile("MyFile"); // In the Activity
```

Outside the Activity class you'll have to keep a reference to the Context class and call the method on that like this – `mContext.deleteFile(fileName);` .

Note: On the app's uninstallation the internal storage is purged as well as the private files created on the external storage using `getExternalFilesDir()` are removed.

Querying for Total and Free Space

On a file object you can query the `getTotalSpace()` method to get the entire size of the storage while `getFreeSpace()` to get the available free space. The returned value of both of them is in bytes that can be converted to a higher unit using this handy little function:

```
// Reference: http://stackoverflow.com/a/5599842

public static String readableFileSize(long size) {
    if(size <= 0) return "0";
    final String[] units = new String[] { "B", "KB", "MB", "GB
    int digitGroups = (int) (Math.log10(size)/Math.log10(1024)
    return new DecimalFormat("#,##0.#").format(size/Math.pow(1
}
```

Checking for space:

```
File file = new File(Environment.getExternalStoragePublicDirec
Log.d(TAG, readableFileSize(file.getTotalSpace())); // dumps "
Log.d(TAG, readableFileSize(file.getFreeSpace())); // dumps "1
```

Hiding Files from Media Scanner

Creating an empty file with the name `.nomedia` (not the dot prefix) inside external storage directories will deter the media scanner from reading your media files and providing them to other apps through the `MediaStore` content provider. If you really want to hide files from the user and other applications then use the internal storaget that we discussed earlier.